IA321 - Meta Learning

Ilias Harkati, Armand de Villeroché, Florian Perrocheau, Alberic de Foucauld

24 mars 2023

Table des matières

1 Introduction **2**
1.1 Contexte : description de notre compétition pour le meilleure algo . . . 2
1.2 Meta-learning/Optimisation d'Hyperparamètres & Sélection d'algorithme . . . 2
1.3 Objectif et Plan d'action . . . 2

2 Problèmes **3**
2.1 Problème 1 : Optimisation des hyperparamètres d'un SGDClassifier . . . 3
2.2 Problème 2 : Optimisation des hyperparamètres d'un ResNet18 . . . 3

3 Algorithmes : Descriptions des algos et Explications de chaque partie avec bibliographie **3**
3.1 PSO . . . 3
3.2 CMA-ES . . . 4
3.3 BO . . . 4
3.4 TPE . . . 4
3.5 Reinforcement Learning et Proximal Policy Optimisation . . . 5
3.5.1 Le cadre du Reinforcement Learning . . . 5
3.5.2 Proximal Policy Optimisation . . . 5
3.6 Learning rate scheduler . . . 6
3.7 Environnements de RL . . . 7

4 Résultats **7**
4.1 Description du jeu de données (CIFAR10) et Plan d'expérience . . . 7
4.2 Baseline case : Estimation des performances de notre algo de référence : ResNet18 sur 10% de CIFAR10 . . . 8
4.3 Problème 1 : Optimisation des hyperparamètres d'un SGD Classifier . . . 9
4.4 Problème 2 : Optimisation des hyperparamètres d'un ResNet18 . . . 10
4.5 Comparer deux approches pour l'algorithme PPO . . . 11
4.6 Comparer l'apprentissage de schedulers par PPO et SAC . . . 12
4.6.1 Environnement du modèle d'agent scheduler . . . 12

5 Conclusion **14**

1 Introduction

1.1 Contexte : description de notre compétition pour le meilleure algo

À l'heure où les modèles de Deep Learning deviennent de plus en plus performants, et par la même occasion de plus en plus lourds en mémoire et gourmands en temps de calcul et en paramètres, de nouvelles approches sont explorées par les chercheurs et les entreprises. Tirant parti des récentes avancées en terme d'architectures ou de modélisation, l'objectif est désormais de surmonter ces contraintes de temps de calcul, d'accès à des jeux de données réduits qui deviennent des enjeux importants dans la réduction de l'empreinte climatique d'un modèle.

C'est dans ce cadre qu'on se propose d'étudier le meta-learning, et plus particulièrement l'optimisation d'hyperparamètres, au travers de deux problèmes d'optimisation contraints en ressources.

1.2 Meta-learning/Optimisation d'Hyperparamètres & Sélection d'algorithme

Le meta learning est un domaine de l'apprentissage automatique ; c'est un terme générique qui regroupe l'ensemble des tâches d'apprentissage machine, non pas destinées à entraîner des modèles sur des données, mais qui sont à destination des modèles d'intérêt eux-mêmes. Aussi, le meta-learning est souvent associé à l'idée d'apprentissage de l'apprentissage, où le but est d'utiliser les techniques d'apprentissage machine pour faire apprendre à des modèles.

Parmi les enjeux du meta-learning il y a entre autres le fait d'élaborer des méthodes d'apprentissage automatique et des modèles capable d'apprendre avec très peu d'exemples - comme sait le faire un être humain par exemple -, l'entraînement de modèles destinés à être les plus généraux possibles - c'est-à-dire d'être capable de s'adapter au plus de tâches possibles. Ou encore le transfert des connaissances d'un modèle déjà entraîné à un autre.

L'un des enjeux du meta-learning est l'élaboration de méthodes d'apprentissage automatique et de modèles ne nécessitant que très peu d'exemples en étant très généraux. Ainsi, ces meta-modèles sont capables de s'adapter à une grande variété de tâches et peuvent tirer parti du *Transfer Learning*, i.e récupérer les connaissances d'un modèle pré-entraîné sur une certaine tâche pour accélérer l'entraînement sur un problème de la même classe.

Un des cas particuliers important du meta-learning est l'optimisation d'hyperparamètres. En général, les modèles d'apprentissage automatique sont réglables à différents degrés : la profondeur d'un arbre, pour un arbre de décision, le degré d'un polynôme pour une régression polynomiale, la taille d'une forêt pour un random forest etc. Ces hyperparamètres peuvent avoir un impact important sur la qualité de l'apprentissage du modèle, sur l'adéquation de celui-ci aux données ou sur sa vitesse de convergence par exemple.

Optimiser ces hyperparamètres est donc un enjeu crucial pour bénéficier des meilleures capacités des modèles entraînés tout en s'assurant d'une convergence plus rapide qu'un processus aléatoire. C'est donc aussi un enjeu dans le déploiement industriel de l'apprentissage automatique. C'est à ce problème d'optimisation d'hyperparamètres que s'adressera ce projet, en comparant notamment différentes méthodes d'optimisation d'hyperparamètres.

1.3 Objectif et Plan d'action

Afin de tenter de traiter cette vaste problématique qu'est l'optimisation d'hyperparamètres nous l'étudierons au travers de différents problèmes que nous traiterons sous l'angle d'un environnement contraint en ressources.
Aussi, dans un premier temps, l'objectif sera d'optimiser les hyperparamètres d'un modèle relativement simple. Plusieurs approches et leur implémentations seront présentées et utilisées afin d'évaluer leur performances et leur pertinence dans le cas de ce problème simple.

Dans un second temps, l'objectif sera d'appliquer ces méthodes et d'autres mais cette fois afin de traiter un problème d'optimisation un peu plus complexe. Le champ de l'apprentissage par renforcement -*Reinforcement Learning* (RL)- sera notamment exploré afin d'évaluer sa pertinence dans le cadre de l'optimisation d'hyperparamètres.

2 Problèmes

Afin d'étudier la problématique introduite dans la partie précédente on se propose de tester deux problèmes différents - avec différentes approches qui seront présentées dans la partie résultat.

2.1 Problème 1 : Optimisation des hyperparamètres d'un SGDClassifier

On dispose d'un ResNet18 pré-entrainé et qu'on utilise pour extraire des features d'une base de données labélisée. Une fois ces features extraites on souhaite utiliser un modèle de support vector machine (SVM) pour catégoriser nos données. Dans ce problème on s'intéresse à l'impact que pourraient avoir deux hyperparamètres α et η_0 de notre SVM - implémenté ici par la classe SDGClassifier de la bibliothèque scikit learn et qui correspondent au facteur de régularisation et au learning rate initial - sur la capacité de classification de notre modèle.

2.2 Problème 2 : Optimisation des hyperparamètres d'un ResNet18

Dans ce problème on s'intéresse cette fois à l'entrainement d'un ResNet18 initialisé aléatoirement. On souhaite étudier la capacité d'apprentissage du modèle en fonction du *batch_size* et du *learning rate* choisis. Par fidélité aux contraintes du temps de calcul du monde professionnel, on se place dans une situation où notre temps d'apprentissage est contraint. L'objectif est alors de trouver pour quelles valeurs de ces hyperparamètres le modèle pourrait être entraîné le plus efficacement.

Dans ce problème on utilisera la base de données CIFAR-10, et un ResNet18 qui n'a pas été pré-entraîné.

3 Algorithmes : Descriptions des algos et Explications de chaque partie avec bibliographie

3.1 PSO

La méthode d'optimisation par essaims particulaires[1] (PSO) est une méthode d'optimisation d'une fonction objectif qui part de plusieurs solutions potentielles - les particules - et qui déplace ces particules dans l'espace des solutions selon une information globale et une information locale enregistrée par chaque particule, via des formules d'évolution. C'est à partir de ces informations locales et globales et du grand nombre de particules qu'émerge "l'intelligence" de l'essaim.

Les formules d'évolutions sont données par :

$$V_{k+1} = \omega V_k + b_1(P_i - X_k) + +b_2(P_g - X_k)$$

$$X_{k+1} = X_k + V_{k+1}$$

L'algorithme PSO étant une méthode d'optimisation, on peut l'utiliser dans le cas d'une recherche d'hyperparamètres d'un modèle d'apprentissage automatique. En effet, ces hyperparamètres sont souvent sous la forme d'un vecteur et il suffira alors de considérer la précision du modèle comme étant la fonction objéctif de notre méthode d'optimisation et les hyperparamètres comme les paramètres à optimiser de cette fonction objectif et simplement laisser tourner la méthode.

3.2 CMA-ES

L'algorithme CMA-ES (Covariance Matrix Adaptation Evolution Strategies) est une méthode d'optimisation de type stratégie évolutionnaire et qui ne requiert pas de connaître le gradient de la fonction à optimiser. Les stratégies évolutionnaires sont un type de méthode d'optimisation qui partent d'une population initiale de solutions, sélectionnent quelques unes de ces solutions selon un certain critère, puis produisent, à partir de ces solutions sélectionnées une nouvelle population. L'idée étant que de génération en génération les populations de solutions s'approchent de plus en plus en moyenne de l'optimum.

CMA-ES fonctionne selon ce principe : à chaque étape les meilleures solutions sont conservées, et à partir de celles-ci, une distribution gaussienne est entraînée sur les données pour qu'une nouvelle population de solutions soit échantillonnée à partir de celles-ci.

3.3 BO

Les algorithmes d'optimisation bayésienne (BO) [2] sont des algorithmes d'optimisation qui reposent sur l'estimation d'une distribution de la fonction objectif. Leur principe est d'essayer de construire une distribution qui maximise la vraisemblance de la sortie attendue de la fonction objectif sachant un certain paramètre avec lequel on l'évaluerait. Ces algorithmes utilisent les échantillonnages de la fonction objectif pour affiner itérativement la distribution a posteriori, ainsi plus l'algorithme réalise d'appels à celle-ci plus celle-ci est connue. Comme ces méthodes construisent un modèle de la fonction objectif elles sont dites basés sur un modèle - en anglais : model-based.

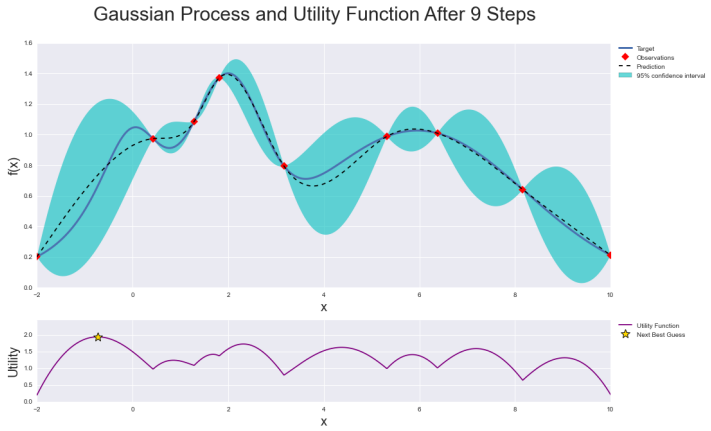

FIGURE 1 – Exemple de distribution estimée par un algorithme d'optimisation bayésienne

3.4 TPE

L'algorithme TPE (Tree-structured Parzen Estimator) [3] est une méthode d'optimisation séquentielle proche de l'optimisation bayésienne. À l'instar de cette dernière cet algorithme essaie de construire un modèle de la fonction objectif, et ce modèle est aussi une distribution ; mais cette fois c'est la vraisemblance d'une entrée de la fonction objectif sachant une certaine évaluation, qui est estimée.

Pour ce faire l'algorithme TPE utilise en fait deux distributions basées sur les solutions déjà échantillonnés. Une distribution pour les évaluations jugées bonnes (celles qui optimisent le mieux la fonction objectif), et une pour les évaluations jugées mauvaises. Ces distributions - sous la forme de mélanges gaussiens tronqués, et calculées à partir des échantillons connus - sont ensuite utilisées pour estimer à

quel point une entrée de la fonction objectif est "prometteuse". Cette valeur est calculée via le rapport entre la distribution des bonnes solutions et celle des mauvaises solutions.

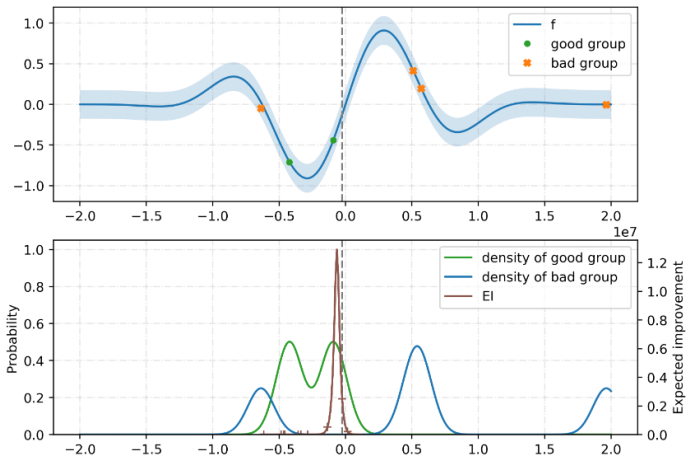

FIGURE 2 – Exemple des distributions estimées par TPE

3.5 Reinforcement Learning et Proximal Policy Optimisation

Cette partie vise à présenter l'algorithme de Proximal Policy Optimisation (PPO) dans le cadre d'une approche par renforcement. Après avoir décrit le cadre de l'apprentissage par renforcement, les méthodes de gradient de Policy sont présentées avant qu'une dernière partie détaille l'implémentation de l'algorithme PPO. Cette partie est très inspirée d'un travail réalisé précédemment dans l'année [4].

3.5.1 Le cadre du Reinforcement Learning

L'apprentissage par renforcement ou Reinforcement Learning (RL) est un processus de contrôle à temps discret dans lequel un agent interagit en boucle fermée avec son environnement. À chaque instant t, l'agent reçoit une observation de l'état o_t et applique une action a_t à l'environnement.

L'objectif d'un agent RL est de trouver une politique π qui maximise le retour estimé. Dans ce projet, la politique π est considérée comme une fonction stochastique $\pi(s; a) : S \text{ x } A \rightarrow [0; 1]$ qui donne la probabilité qu'une action spécifique a soit choisie compte tenu de l'état s .

3.5.2 Proximal Policy Optimisation

Proximal Policy Optimization est une avancée récente dans le domaine du RL qui obtient des performances remarquables. Il s'agit d'une extension de Trust Region Policy Optimization (TRPO), une approche qui garantit que chaque mise à jour de la policy n'est pas destructrice [1] et en même temps pas trop petite - ce qui conduirait à un entraînement prolongé. Dans la TRPO, la fonction objectif appelée objectif de substitution et notée L^{CPI} (pour une itération conservatrice de policy) est légèrement différente de celle des méthodes de Policy gradient, mais conduit à un problème d'optimisation identique. Le but est de :

$$\max_{\theta} L^{CPI}(\theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}((a_t|s_t))} \hat{A}_t \right] \tag{1}$$

$$\text{avec } \hat{\mathbb{E}}_t \left[KL[\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t)] \right] \leq \delta \tag{2}$$

1. Une approche de mise à jour de policy est dite non destructrice si elle n'écrase pas la policy précédente lors d'une itération

Utiliser la contrainte avec KL [2] est un moyen de maintenir basse la différence entre l'ancienne et la nouvelle distribution de la politique , ce qui permet de mettre en œuvre une *région de confiance* dans laquelle évolue la Policy.

Bien que TRPO évite les défauts des méthodes Actor-Critic, la contrainte de KL brute ajoute une surcharge supplémentaire au processus d'optimisation et peut conduire à des comportements d'apprentissage indésirables. PPO modifie la fonction objectif afin d'inclure implicitement la contrainte d'avoir de petites mises à jour de la policy dans le problème d'optimisation. La fonction objectif devient L^{CLIP} ; en définissant $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}((a_t|s_t))}$, on peut écrire :

$$L^{CLIP} = \hat{\mathbb{E}}_t \left[\min \left(\underbrace{r_t(\theta)\hat{A}_t}_{\text{objectif TRPO}} , \underbrace{\text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t}_{\text{objectif TRPO restraint}} \right) \right] \tag{3}$$

r_t est le rapport entre la probabilité sous l'ancienne policy et la probabilité sous la policy actuelle, de choisir une action spécifique dans un état. En d'autres termes, $r_t > 1$ pour une action spécifique signifie qu'elle est plus susceptible d'être entreprise dans le cadre de la nouvelle policy (par rapport à l'ancienne politique), et $r_t < 1$ signifie que l'action est moins susceptible d'être entreprise dans le cadre de la nouvelle policy. Plus important encore, les valeurs de r_t éloignées de 1 signifient que la probabilité d'entreprendre l'action donnée a beaucoup varié lors de la précédente mise à jour de la policy. C'est exactement ce que TRPO et PPO cherchent à éviter : de grands changements de policy en une seule mise à jour qui pourraient nuire à la policy apprise jusque là.

Ainsi, l'objectif de la mise à jour de la policy est de déplacer θ de manière à augmenter L^{CLIP}. Si l'avantage est positif, L^{CLIP} peut être augmenté en augmentant l'objectif TRPO (voir (3)), donc d'augmenter r_t. Toutefois, comme on prend le minimum des objectifs clippés et non clippés, L^{CLIP} ne peut pas dépasser $\hat{\mathbb{E}}_t[(1+\epsilon)\hat{A}_t]$. Cette valeur seuil empêche l'incitation à augmenter r_t au-delà de $1+\epsilon$. Inversement, pour les actions présentant un avantage négatif, le clip empêche de diminuer r_t en dessous de $1-\epsilon$. En bref, l'objectif de PPO est tel que la nouvelle policy ne bénéficie pas d'un changement trop important par rapport à la précédente.

L'algorithme PPO 2 est divisé en deux temps : le premier recueille des séquences d'épisodes et calcule l'estimation de l'avantage. Une fois qu'un lot d'épisodes a été collecté, un second temps exécute une montée de gradient sur le réseau de policy en utilisant l'objectif PPO.

3.6 Learning rate scheduler

Le learning rate est un paramètre important qui va pondérer la mise à jour des gradients lors de la phase de *back-propagation* [3].

Ainsi le learning rate détermine l'importance de la mise à jour des poids du réseau et donc la taille du pas effectué vers le minimum visé. Lors de l'entrainement d'un réseau de neurones, il est ainsi coutume de réduire le learning rate au fil des epochs[5] afin d'améliorer la convergence. Cette idée découle de l'intuition qu'avec un haut learning rate, le modèle possède une haute inertie et est donc incapable de converger vers les minima locaux de la fonction de perte. Cependant, si le learning rate est assez petit, le modèle aura une très faible inertie et pourra donc converger vers un minimum de la fonction de coût.

2. La divergence de Kullback-Leibler (KL) est une mesure de la distance statistique entre deux distributions de probabilité.

3. Une fois que le réseau de neurone a effectué la phase de *forward-propagation* ie a prédit un estimateur du label visé, l'erreur de prédiction est calculée et le gradient de cette erreur par rapport à tous les paramètres du modèle est calculée. Ensuite, chaque paramètre est mis à jour selon l'équation :

$$Param_i \longleftarrow Param_i - Learning\ rate * \nabla_{Param_i} Erreur\ prédict.$$

Un bon learning rate est choisi par compromis : il n'est pas trop petit pour que notre algorithme puisse converger rapidement, et il n'est pas trop grand pour que notre algorithme n'oscille pas autour d'un minimum sans pouvoir l'atteindre.

Un learning rate Scheduler est un objet prédéfini qui ajuste le learning rate entre les epochs ou les itérations au fur et à mesure que l'entraînement progresse. Les deux techniques les plus courantes de learning rate scheduler sont :

— Learning rate constant : on initialise le learning rate et on ne le modifie pas pendant l'entraînement

— Décroissance du learning rate : on sélectionne un learning initial qui est diminué progressivement suivant un planning de temps pré-établi.

Dans le cadre de ce projet, les deux techniques précédentes ont été explorées. Ainsi, garder un learning rate constant au cours de l'entrainement permettra d'implémenter simplement et de comparer plusieurs meta-modèles sur un même temps d'entraînement. Cette première approche sera présentée dans les sections 4.3 et 4.4.

Dans le cadre du meta-learning et de l'objectif que le meta-modèle puisse surpasser le modèle d'intérêt présenté en Section2.2 sur la même tâche [4], l'exploration de la seconde approche de learning rate scheduler a été menée. Ainsi, la question de la pertinence d'entraîner un meta-Agent qui puisse décroître le learning rate du modèle d'intérêt pendant l'apprentissage sera traité dans ce projet. L'idée de cette seconde approche est d'entraîner un agent à modifier le learning rate de l'entrainement plutôt que d'importer un learning rate scheduler qui va décroître le learning rate selon un comportement pré-établi à des instants pré-établis. [5]

3.7 Environnements de RL

Deux environnements ont été testés pour l'optimisation du learning rate et du batch size, sans scheduler.

Le premier environnement cherche à faire varier ces deux paramètres de façon continue. Il correspond à l'idée intuitive d'essayer des paramètres, de regarder si l'accuracy s'améliore ou pas, et de les modifier légèrement en fonction de l'amélioration. Ainsi, l'observation de l'environnement renvoie la valeur actuelle des paramètres. L'action qu'un agent peut prendre est une variation continue des paramètres. Finalement, la reward est la différence entre l'accuracy de l'étape précédente, avant d'appliquer l'action, et l'accuracy après avoir modifié les paramètres en fonction de l'action.

Le deuxième modèle prend une approche différente, inspirée de l'article [6]. L'approche proposée dans cet article a été simplifiée. L'observation de l'environement renvoie un contexte c constitué des hyperparamètres de l'épisode précédent, ceux de l'épisode d'avant, et de l'accuracy de l'épisode précédent. La policy choisit ensuite directement de nouveaux hyperparamètres. Finalement, la reward donné par l'environnement est l'accuracy calculée par les nouveaux paramètres.

4 Résultats

4.1 Description du jeu de données (CIFAR10) et Plan d'expérience

Lors de ce projet, le jeu de données utilisé pour la comparaison des performances de chacune de nos approches est le dataset CIFAR10 [7]. Ce dataset est composé de $60,000$ images colorées de dimension 32x32 de 10 classes d'images, soit $6,000$ images par classe. Le dataset est distribué entre $50,000$

4. Rappel : Le modèle d'intérêt est un modèle ResNet18 entraîné sur 10% du dataset CIFAR10 au cours de 10 epochs d'entraînement

5. Il faut bien noter ici que le méta-modèle agit ici **au cours** et **au coeur** de l'entrainement du modèle d'intérêt, et non à une échelle plus globale comme dans les Sections 4.3 et 4.4. Ces deux approches sont différentes et explore les deux techniques les plus courantes de learning rat scheduler.

images d'enraînement et 10,000 images pour l'évaluation. Les classes sont complètement mutuellement exclusives.

Un extrait du dataset CIFAR10 est présenté ci-dessous :

FIGURE 3 – Exemple d'images présentes dans le dataset CIFAR10

Comme cela est expliqué dans la section Section 4.2, seul un extrait de 10% de CIFAR10 va être utilisé pour l'évaluation de la précision de nos méthodes.

4.2 Baseline case : Estimation des performances de notre algo de référence : ResNet18 sur 10% de CIFAR10

Afin d'évaluer et de juger la performance de chacune des approches présentées précédemment, chaque modèle sera comparé à un modèle "témoin" dont les performances serviront de baseline pour ce projet. Ce modèle est l'architecture ResNet18 présentée par He *et al.* dans [8].

Comme décrit dans la section 4.1, le dataset CIFAR10 contient 50,000 exemples pour le training et 10,000 exemples pour la phase de test. Dans le cadre de ce projet, seule une partie de ce jeu de données sera utilisée lors de l'entraînement du meta-modèle et la considération du temps d'entrainement a poussé la restriction à 10% de CIFAR10. Pour évaluer la performance d'une approche meta-entraînée sur ces 10% de CIFAR10, un échantillon de CIFAR10 a été réalisé afin d'obtenir la performance de l'architecture ResNet18 sur différentes tailles d'échantillon, permettant de situer la performance de chaque approche. En effet, il est important de rappeler que chacune des approches développées dans ce rapport a été entraînée sur un échantillon de 10% de CIFAR10 et le but de ce projet est de montrer comment l'adoption de meta-modèles peut permettre d'atteindre des performances supérieures pour la même taille d'échantillon et équivalentes à un entraînement sur un échantillon plus important.

Comme décrit dans la Section 4.3, la mesure utilisée dans ce rapport est la précision. Ainsi, la figure Fig.4 présente les performances de l'architecture ResNet18 sur différentes tailles d'échantillon du dataset CIFAR10 [6].

6. L'entraînement a été réaslisé sur 10 epochs, avec un optimizer de Adam et un learning rate de répart égal à 10^{-3}

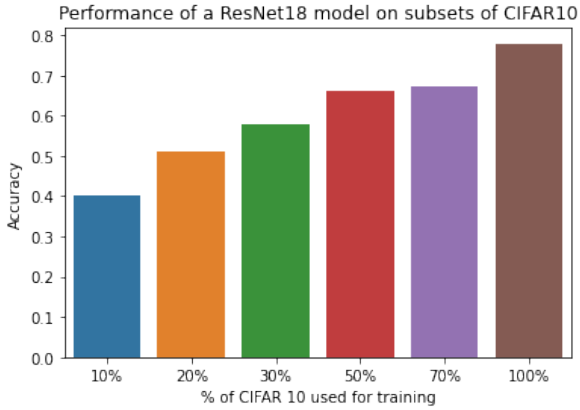

FIGURE 4 – Performance du ResNet18 sur différents échantillons de CIFAR10

Les résultats présentés sur la figure Fig.4 sont détaillés dans le tableau Table[3] :

Taille de l'échantillon (en %)	10%	20%	30%	50%	70%	100%
Taille du training set	5,000	10,000	15,000	25,000	35,000	50,000
Précision du modèle	0.401	0.512	0.580	0.6604	0.671	0.7791

TABLE 1 – Précision du modèle ResNet18 sur des échantillons de CIFAR10

4.3 Problème 1 : Optimisation des hyperparamètres d'un SGD Classifier

Méthode d'évaluation

Pour étudier cette problématique on travaillera sur la base de données MNIST [9], et on utilisera un ResNet18 pré-entraîné importé de la bibliothèque PyTorch.

De plus afin d'évaluer les différents modèles sur ce problème - CMAES et TPE pour ce problème, on fixera notre espace de recherche comme-suit :

$$\alpha \in [0, 1],\ \eta_0 \in [10^{-5}, 1]$$

, et on regardera l'accuracy (exprimée en pourcentage) maximum en moyenne de notre SGDClassifier ainsi que le coefficient directeur de la pente du run moyen pour voir si le modèle apprend.

Finalement pour comparer les modèles on s'autorisera 50 appels à la fonction objectif pour chaque algorithme, algorithmes que l'on lancera sur 10 *runs* afin d'avoir une performance moyenne - et que l'on comparera à une recherche aléatoire (RS).

Résultats :

Algorithmes	RandomSearch	CMAES	TPE
Accuracy max moyenne	63.89	64.12	64.27
Variance	0.12	0.02	0.05
Max	64.31	64.4	64.27
Min	63.07	63.93	63.81
Coefficient	-0.03	0.19	0.23

TABLE 2 – Comparaison de l'accuracy (exprimée en pourcentage) maximum en moyenne des différents algorithmes - temps (en nombre d'appel à la fonction objectif) : 50 - nombre de runs : 10

Interprétation :

Des trois algorithmes étudiés on observe, au travers des coefficients directeurs de leur accuracy moyenne, que CMAES et TPE semblent tous deux apprendre au fil du processus. Avec un apprentissage qui semble légèrement meilleur pour le TPE. Ce que corrobore la moyenne sur les runs des accuracy max sur chaque run.

4.4 Problème 2 : Optimisation des hyperparamètres d'un ResNet18

Méthode d'évaluation

Dans le cadre de l'évaluation de nos algorithmes pour notre problème notre espace de recherche doit être fixé aussi on se fixera comme bornes pour nos hyperparamètres $bs \in [20, 200]$ et $lr \in [10^{-2}, 10^{-5}]$. De plus comme nos hyperparamètres à tester ne sont ni du même type : entier et flottant, ni du même ordre de grandeur. On normalisera notre espace de recherche $\mathcal{X}$ pour nos algorithmes : celui-ci correspondra à la boîte $\mathcal{X} = [0, 1]^2$, après quoi les hyperparamètres seront remis aux bonnes valeurs dans la fonction objectif. La fonction de normalisation du batch size est linéaire, tandis que la fonction de normalisation du learning rate est exponentielle. Cela permet aux algorithme d'explorer beaucoup plus facilement les valeurs faible de learning rate, plutot que d'explorer seulement le haut des valeurs.

On se limitera également, dans l'évaluation de nos algorithmes, à une fraction seulement de notre base de données. Ainsi on se contentera d'entraîner notre ResNet18 sur 10 epochs sur 10% de la base MNIST. Et l'objectif sera de tenter de maximiser l'accuracy de notre réseau exprimée en pourcentage.

Enfin pour comparer les modèles entre eux on s'autorisera un budget identique de 32 appels à la fonction objectif pour chaque algorithme afin de mesurer leur performance sur un temps identique. Et on effectuera aussi 10 *runs* afin d'avoir une performance moyenne evaluée avec l'accuracy (exprimée en pourcentage) maximum en moyenne de notre SGDClassifier, ainsi qu'avec le coefficient directeur de la pente du run moyen.

Résultats :

Algorithmes	RS	PSO	CMAES	BO	TPE	PPO contexte	PPO continu
Accuracy max moyenne	57.46	58.82	**59.28**	59.08	57.80	53.35	54.05
Variance	3.22	1.41	4.11	1.49	3.90	1.24	2.26
Max	59.6	60.8	62.4	62.2	62.0	55.4	56.7
Max	54.4	56.8	56.0	57.4	55.2	52.0	51.2
Coefficient	-0.04	0.06	0.15	0.12	**0.26**	-0.04	-0.09

TABLE 3 – Comparaison de l'accuracy (exprimée en pourcentage) finale des différents algorithmes - temps (en nombre d'appel à la fonction objectif) : 32 - nombre de runs : 10

Interprétation :

On remarque que la méthode d'apprentissage par renforcement PPO, testée sur deux environnements différents, ne semble pas apprendre au regard de sa moyenne qui ne va pas au-delà de la recherche aléatoire (RS), mais surtout au regard du coefficient directeur quasiment nul.

A contrario la méthode CMAES semble avoir sur nos tests le mieux performé avec une accuracy maximum en moyenne de 59.28%, même si l'algorithme TPE semble lui avoir conduit à la meilleure amélioration moyenne du modèle avec un coefficient de 0.29. On peut en conclure qu'il a le mieux appris. Suivi de CMAES et de l'optimisation bayésienne. Un peu plus surprenant l'algorithme des essaims particulaires semble avoir peu appris ou pas appris du tout avec un coefficient en dessous de 1. Finalement, les algorithmes de RL donnent des résultas inférieurs aux autres. Ce n'est pas étonant puisque ces algorithmes doivent apprendre leur policy avant de pouvoir donner des résultats fiables, et donc n'ont pas le temps d'apprendre avec un budget d'appel aussi limité.

4.5 Comparer deux approches pour l'algorithme PPO

Cette section va présenter la comparaison des deux approches RL énoncées en Section 2.2. La performance [7] de deux agents dont la policy sera établie par l'algorithme PPO mais dont l'un considère le learning rate constant au cours des 10 epochs tandis que le second agent va agir comme un learning rate scheduler (cf.Section 3.6). A noter que le premier agent va apprendre à identifier le learning rate et le batch size optimal avant l'entraînement du ResNet18 sur 10 epochs, tandis que le second agent ne se concentre que sur le learning rate mais cherche la meilleure valeur de learning rate à chaque epoch de l'entraînement du ResNet18. Ces deux approches sont différentes et l'intérêt de cette section est l'évaluation de ces approches : vaut-il mieux apprendre plusieurs hyperparamètres (comme le learning rate et le batch-size) optimaux ou plutôt se concentrer sur l'un deux et optimiser sa valeur à chaque instant de l'apprentissage ?

Pour des raisons de clarté, le modèle apprenant le learning rate et le batch size sera indiqué par "_cst_lr" et le modèle de scheduler par l'indication "_lr_scheduler". Les deux types d'environnements énoncés en Section 3.7 sont présentés séparément pour la comparaison avec l'agent scheduler, qui lui évolue dans l'environnement décrit dans la section 4.6.1 . La figure Fig.5 présente la comparaison de l'agent scheduler avec l'agent évoluant dans l'environnement inspiré de [6] (cf.Section 3.7). La figure Fig.6 présente ainsi la comparaison du même agent scheduler mais ici, l'agent RL de la méthode "_cst_lr" évolue dans l'environnement a variations continue des paramètres, décrit en fin de section 3.7. Une figure récapitulative des trois courbes est présentée en Annexe

7. Rappelons que pour ce projet, la performance d'un modèle représente la précision atteinte par un ResNet18 entraîné sur 10 epochs et sur 10% de CIFAR10, et dont le comportement est influencé par les actions de ce méta-modèle.

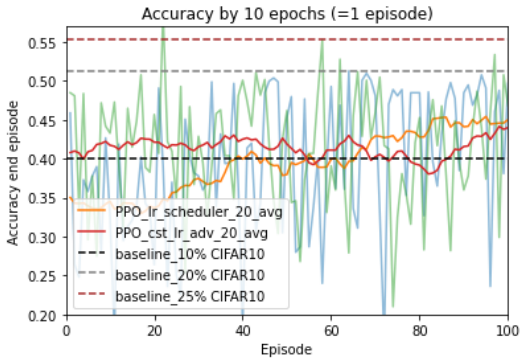

FIGURE 5 – Accuracy par épisode pour l'agent avec learning rate constant sur l'environement basé sur le contexte (rouge) et l'agent scheduler (orange)

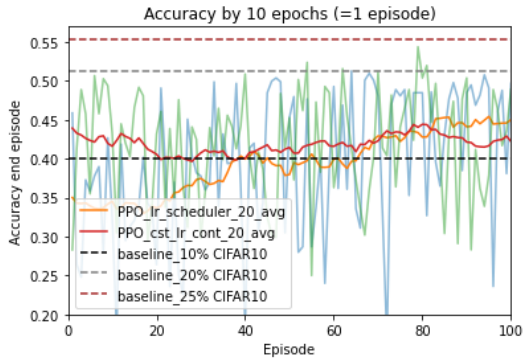

FIGURE 6 – Accuracy par épisode pour l'agent avec learning rate constant avec variation continue des paramètres (rouge) et l'agent scheduler (orange)

Ces figures montrent que les modèles n'apprenent peu en un temps aussi court. Cependant, on observe une amélioration, en particulier pour la méthode utilisant le scheduler. Tout les modèles permettent d'obtenir des accuracy plus élevé que la baseline en moyenne. Il est donc probable que un apprentissage plus long permettrai aux modèles d'obtenir des meilleurs résultat, et de gagner en stabilité.

4.6 Comparer l'apprentissage de schedulers par PPO et SAC

Cette partie va présenter la comparaison entre des performances obtenues par le méta-scheduler entraîné sous la méthode PPO et avec l'architecture SAC [8]. Rappelons que l'objectif de cette approche particulière de scheduler est l'entraînement d'un méta modèle pour ajuster le learning rate afin de maximiser la précision du ResNet18 après 10 epochs d'entrainement. La partie précédente (Section 4.5) a présenté les résultats d'un algorithme PPO appliqué au learning rate scheduler ajusté par le méta modèle et a détaillé la comparaison avec la même implémentation PPO appliquée au cadre du learning rate constant.

4.6.1 Environnement du modèle d'agent scheduler

Pour comparer la performance de ces deux approches (PPO et SAC), l'environnement suivant a été implémenté :

— S : Espace d'état fini et Continu : $s_t = [lr, accuracy] \in [10^{-5}, 10^{-2}]x[0, 1]$. Le $batch\_size$ est considérer constant et égal à 64 ici. [9]

— A : Espace d'action fini et Continu : $a_t = [lr_pred] \in [10^{-5}, 10^{-2}]$. L'agent RL impose le nouveau learning rate pour la prochaine epoch.

— R :S x A x $S \rightarrow R$: Fonction de récompense choisie $r_t = accuracy^{<t>} - accuracy^{<t-1>}$

Le choix de cette fonction de récompense encourage l'agent RL à améliorer l'accuracy du modèle à chaque epoch. De plus, la somme cumulée des *rewards* par épisode est égale à la précision du ResNet18 à la fin des 10 epochs d'entrainement, soit exactement la métrique qui nous interesse dans ce projet.

8. L'algorithme Soft Actor-Critic (SAC) permet de tirer partie des avantages d'algorithmes *on-policy* comme PPO et *off-policy* comme Deep Deterministic Policy Gradient. Le lecteur est invité à consulter l'Appendix 5 pour une présentation de l'algorithme SAC.

9. La valeur de 64 pour le batch_size permet un entrainement rapide de ResNet18 tout en concentrant l'apprentissage de l'agent RL sur la plannification du learning rate. L'accuracy est calculée à chaque epoch en évaluant le ResNet18 sur un set de test déterminé à chaque epoch.

Pour comparer la performance de chaque algorithme (PPO et SAC), chaque agent RL va être entraîné sur 100 episodes. Dans un premier temps, le choix de faire intervenir l'agent à chaque epoch d'entrainement du ResNet18 avait été fait. Cependant, les conseils présentées dans la littérature [5] sur l'utilisation des scheduler suggèrent qu'il est courant d'effectuer une mise-à-jour du learning rate uniquement lorsque l'erreur de validation augmente, ce qui correspond à une baisse de la précision. Ces deux approches ont été traitées dans cette partie et leur résultats seront présentés en Appendix **??**. La seconde approche présente des performances plus intéressantes qu'une optimisation du learning rate à chaque epoch, c'est donc sur cette observation que seront comparer PPO et SAC.

Les implémentations de PPO et de SAC choisie sont inspirées de celles présentées par Ding dans [10]. Ce choix est motivé par la volonté d'avoir accès à tout le code des algorithmes pour pouvoir utiliser tous les leviers possibles dans cette recherche de performance optimale.

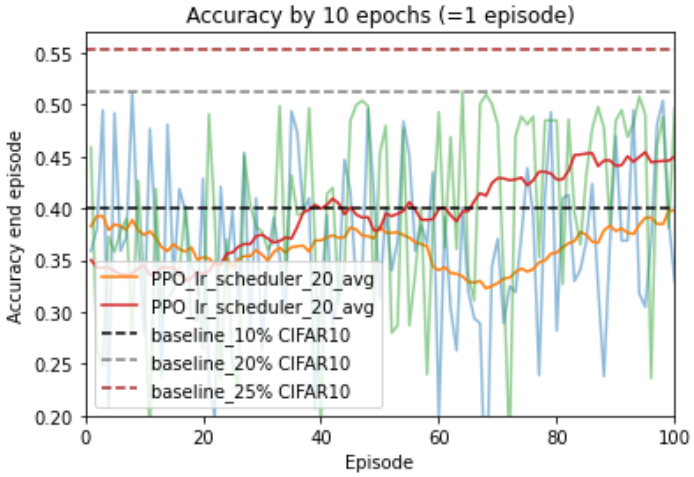

FIGURE 7 – Accuracy par épisode pour PPO (rouge) et SAC (orange)

La figure ci-dessus représente la précision en fin d'épisode i.e après 10 epochs d'entrainement - du ResNet18 sur 10% de CIFAR10, pour 100 épisodes d'entrainement. Une rapide recherche des hyperparamètres optimaux pour les méta-modèles a été effectué et est présenté en Appendix **??**. En effet, les implémentations utilisées étaient calibrés sur 1000 épisodes d'entrainement, et donc les 100 premiers épisodes ne représentaient que de l'exploration, sans apprentissage. Un effort a été fait pour adapter ce set-up à un entraînement sur 100 épisode.

D'après la figure Fig.7, force est de constater que le meta-modèle guidé par PPO apprend correctement sur 100 episode (la courbe rouge est croissante) et vient même dépasser la précision obtenue par le baseline modèle (courbe en pointillé noir) ! L'approche méta-learning a permis de surpasser le modèle ResNet18 seul pour un entraînement sur 10% de CIFAR10. Cependant, la courbe du meta-modèle SAC n'est pas satisfaisante sur ces 100 épisodes d'entraînement, en témoigne les oscillations sur la précision. Une revue des hyper-paramètres de cette implémentation SAC pourrait permettre de corriger ce manque d'apprentissage sur 100 épisode. [10]

La figure Fig.7 justifie l'intérêt porté dans l'approche méta-learning de ce problème d'optimisation des hyperparamètres. Le même modèle -ici un ResNet18- entraîné sur le même dataset - ici 10% de CIFAR10- mais guidé par un méta-modèle atteint une performance -ici on s'intéresse à la précision-

10. Cette recherche d'hyperparamètres est actuellement en cours d'exécution, toutefois limitée par la restriction GPU de Google Colab.

supérieure que pour un modèle seul ! De plus, les courbes présentées sont extrêmement prometteuses : en allongeant le temps d'entrainement par exemple sur 200 épisodes, il serait possible d'obtenir la même précision qu'un ResNet18 seul mais entraîné sur 20% des données ! Les problématiques soulevées en introduction sur les contraintes de dataset réduit ou sur le temps de calcul rencontrent ici une alternative robuste.

5 Conclusion

Les algorithmes donnant les meillerus résultats dans un laps de temps court sont les algorithmes classiques d'optimisation, et en particulier CMA-ES et BO. Ils permettent d'atteindre presque 60% d'accuracy, soit l'accuracy obtenue avec 30% des données sans avoir fait de recherche précise d'hyperparamètres, en utilisant seulement 10% du dataset.
Les algorithmes de RL sont beaucoup moins efficace en temps que les algorithmes classiques pour les problèmes d'optimisations. Cependant, leur avantage réside en l'optimisation d'un agent qui peut a priori optimiser des problèmes plus généraux, tels que les meilleurs hyperparamètres avec un modèle ou dataset légèrement différent, sans avoir a refaire la longue phase d'entrainement. Parmi les algorithmes étudié, l'approche ayant les meilleurs résultats est le scheduler du learning rate, entrainé par l'algorithme PPO.

Références

[1] J. et R.Eberhart, *Particle swarm optimization*, IEEE International Conference on Neural Networks, 1995. Proceedings, vol. 4,novembre 1995, p. 1942–1948 vol.4 (DOI 10.1109/icnn.1995.488968), 1995.

[2] J. S. et al., *Practical Bayesian Optimization of Machine Learning Algorithms*, 2012.

[3] J. B. et al., *Algorithms for Hyper-Parameter Optimization*, 2011.

[4] T. Chollet, A. de Foucauld, M. Sutour, P. Potel et A. Cor, « Multi-agent reinforcement learning on low-end hardware for game AI, » 2023.

[5] S. Sudhakar, 2017. adresse : `https://towardsdatascience.com/learning-rate-scheduler-d8a55747dd90`.

[6] X. Liu, J. Wu et S. Chen, « A context-based meta-reinforcement learning approach to efficient hyperparameter optimization, » *Neurocomputing*, t. 478, p. 89-103, 2022.

[7] 2009. adresse : `https://www.cs.toronto.edu/~kriz/cifar.html`.

[8] K. He, X. Zhang, S. Ren et J. Sun, *Deep Residual Learning for Image Recognition*, 2015. doi : `10.48550/ARXIV.1512.03385`. adresse : `https://arxiv.org/abs/1512.03385`.

[9] L. Deng, « The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web], » *IEEE Signal Processing Magazine*, t. 29, n° 6, p. 141-142, 2012. doi : `10.1109/MSP.2012.2211477`.

[10] Z. Ding, *Popular-RL-Algorithms*, `https://github.com/quantumiracle/Popular-RL-Algorithms`, 2019.

[11] Sutton et Barto, *Reinforcement Learning*, MIT Press, 2020.

[12] T. Haarnoja, A. Zhou, P. Abbeel et S. Levine, *Soft Actor-Critic : Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor*, 2018. arXiv : `1801.01290 [cs.LG]`.

Méthode de Policy Gradient

Au lieu de déduire la politique par l'intermédiaire de la fonction de valeur (*value function*) Q, les méthodes de Policy Gradient apprennent la Policy directement. La Policy est généralement paramétrée par un réseau neuronal avec des poids θ. L'objectif consiste alors à trouver les poids θ qui maximisent le retour estimé en suivant la Policy qui en résulte π_θ. À cet égard, les méthodes de Policy gradient s'attaquent de front au problème du RL, en se concentrant directement sur la Policy et en transformant la tâche en un problème d'optimisation. On peut alors définir la fonctin d'objectif J :

$$J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{+\infty} \gamma^t R_{t+1}\right] \tag{4}$$

À partir de poids initialisés aléatoirement, le but est de trouver θ^* tel que :

$$\theta^* = \arg\max_\theta J(\theta) \tag{5}$$

Les poids sont mis à jour suivant une méthode de montée de gradient de la fonction objective J par rapport aux poids :

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta) \tag{6}$$

où α est le learning rate. Pour calculer l'incrément de mise à jour, on a donc besoin du gradient de la fonction objectif. Le théorème du Policy gradient donne l'expression du gradient comme suit :

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a)\, Q_{\pi_\theta}(s,a)] \tag{7}$$

où $Q_{\pi_\theta}(s,a)$ est la fonction de valeur d'état-action (*la state-action value* fonction). Cette expression du gradient permet de le calculer en des termes connus et séparés : le gradient de la log policy et la fonction de valeur Q, égale au retour estimé. Par conséquent, utilisant une approche de Monte-Carlo pour estimer Q comme le retour empirique sur une trajectoire, on peut calculer une estimation du gradient. L'algorithme REINFORCE [11] décrit en 5 utilise cette technique dont le pseudo-code est présenté en Annexe.

En tant que méthode de Monte-Carlo, l'algorithme REINFORCE bénéficie d'une estimation non biaisée du gradient, mais souffre d'une variance élevée. Pour réduire cette variance tout en maintenant le biais inchangé, une approche simple introduit le concept de l'*avantage* d'une action. Une action est dite *avantageuse* si la valeur d'effectuer cette action est supérieure à la valeur de l'état actuel. La fonction d'avantage est définie comme suit :

$$A_{\pi_\theta}(s,a) = Q_{\pi_\theta}(s,a) - V_{\pi_\theta}(s,a) \tag{8}$$

Il apparaît naturellement que la valeur de Q_{π_θ} devrait être proche de V_{π_θ} pour un état et une action donnés - à moins que l'action ne soit particulièrement avantageuse, ce qui signifie que la valeur absolue de A_{π_θ} devrait être considérablement réduite par rapport à Q_{π_θ}. De plus, en injectant A_{π_θ} au lieu de Q_{π_θ} dans le côté droit de (7), on observe que :

$$\mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a)\, A_{\pi_\theta}(s,a)] = \nabla_\theta J(\theta) \tag{9}$$

où $d_{\pi_\theta}(s)$ est la distribution des états sous π_θ. Cette équation découle du fait que la somme de la politique pour toutes les actions possibles est égale à 1 et que le gradient d'une constante est égal à zéro.

Par conséquent, injectée dans le calcul du gradient de la fonction objectif, la fonction d'avantage ne modifie pas l'espérance mais permet de diminuer considérablement les valeurs de l'incrément dans (6). Cela permet de réduire considérablement la variance par rapport à l'utilisation d'une estimation de Q dans la mise à jour du gradient. Il existe d'autres techniques pour réduire davantage la variance des gradients de politique : la méthode de l'acteur-critique aborde ce problème en fournissant une nouvelle architecture pour l'agent RL.

L'algorithme REINFORCE

Algorithm 1 REINFORCE

Initialisation θ arbitrairement
for chaque trajectoire $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ **do**
  for $t = 1$ to $T - 1$ **do**
    $g_t \leftarrow \sum_{k=t}^{T-1} \gamma^{k-t}\, r_{k+1}$ ▷ calcule retour depuis t
    $\theta \leftarrow \theta + \alpha\, \nabla_\theta \log \pi_\theta(s_t, a_t)\, g_t$ ▷ mis à jour θ par montée de gradient
  end for
end for
return θ

L'algorithme PPO

Algorithm 2 PPO

Initialisation θ, w et s arbitrairement
for $k = 0, 1, 2, ...$ **do**
  Collecte un jeu de trajectoires en exécutant π_θ dans l'environement
  Calcule l'estimation de l'Avantage $\hat{A}_t$ à partir de la fonction de valeur actuelle $V_{critic}(s_t, w)$
  Met a jour les paramètres de la policy θ maximisant l'objectif PPO : Eq. (3)
  Met a jour les paramètres du Critic w minimisant l'erreur quadratique moyenne (MSE)
end for

L'algorithme Soft Actor-Critic

Certains des algorithmes de Reinforcement Learning (RL)les plus performants des dernières années comme Trust Policy Optimization (TRPO), Proximal Policy Optimisation (PPO) décrit en Section 3.5 ou Asynchronous Actor-Critic Agent (A3C), souffre de l'aspect *on-policy*[11] de l'apprentissage. Ces méthodes sont en effet inefficace pour échantillonner les données, contrairement aux méthodes *off-policy* comme Deep Deterministic Policy Gradient (DDPG) ou Twin Delayed Deep Deterministic Policy gradient (TD3PG), qui sont capables d'apprendre à partir d'échantillons passés grâce aux *Replay Buffers*[12] Cependant, le problème principal de ces méthodes $off - policy$ est leur grande dépendance aux hyper-paramètres et le potentiellement long *tuning* pour converger.

L'algorithme Soft Actor-Critic (SAC) s'inscrit dans les groupes des méthodes d'apprentissage *off-policy* tout en introduisant des méthodes pour lutter contre le problème de convergence.

L'algorithme SAC est défini pour des tâches dont le domaine d'action est continu. Comme cela est développé par Haarnoja *et al.* dans [12], la nouveauté apporté par l'algorithme SAC est la modification de la fonction objectif traditionnellement utilisée en RL : au lieu de chercher à maximiser le *return*[13], l'idée est ici de maximiser l'entropie de la *policy*.

11. Les méthodes $on - policy$ cherchent à évaluer et améliorer la $Target\ \ policy = Behavious\ \ policy$ qui est utilisée pour choisir l'action, contrairement aux méthodes $off - policy$ qui améliorent et évaluent une $Target\ \ policy$ différente de celle utilisée pour choisir l'action ,ie. la $Behaviour\ \ policy$.

12. Un *Replay Buffers* est une structure qui stock les expériences de l'agent à chaque step d'apprentissage. Un expérience étant définie comme le tuple (s_t, a_t, r_t, s_{t+1}), avec s_t l'état de l'agent au time step t, r_t et a_t les rewards et actions observés au time step t qui font transiter l'agent dans l'état s_{t+1}.

13. Le *return* correspond à la somme pondérée des *rewards* par épisode

L'idée de maximiser l'entropie de la *policy* permet d'encourager explicitement l'exploration en assignant aux actions ayant des *Q-values*[14] proches la même probabilité d'être effectuée, tout en évitant à l'agent d'effectuer la même action à répétition à cause d'une possible inconsistance d'approximation de la state-action value function Q.

Le pseudo-code de cet algorithme est présenté ci-après :

Algorithm 3 Soft Actor-Critic

Initialisation des paramètres ψ, $\bar{\psi}$,θ et ϕ
for chaque iteration **do**
    for chaque step de l'environnement **do**
        $a_t \sim \pi_\phi(a_t|s_t)$
        $s_{t+1} \sim p(s_{t+1}|s_t, a_t)$
        $D \leftarrow D \cup \{(s_t, a_t, r(s_t, a_t), s_{t+1})\}$
    end for
    for chaque step de gradient **do**
        $\psi \leftarrow \psi - \lambda_V \hat{\nabla}_\psi J_V(\psi)$
        $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$
        $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$
        $\bar{\psi} \leftarrow \tau\psi + (1 - \tau)\bar{\psi}$
    end for
end for

14. La state-action value fonction $Q(s, a)$ exprime la qualité de l'action effectuée a lorsque l'agent était dans l'état s

Bilan de la comparaison des deux approches

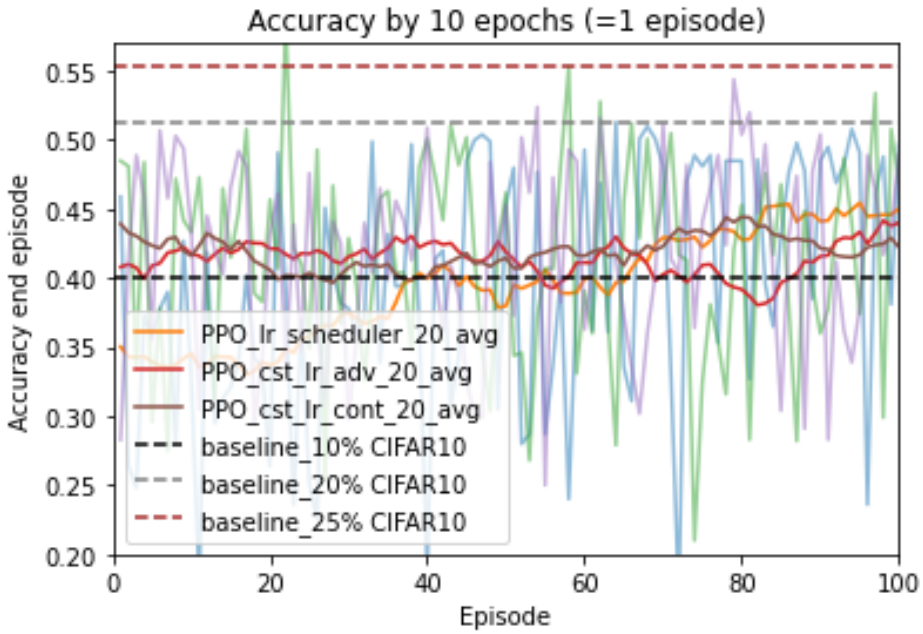

FIGURE 8 – Accuracy par épisode pour l'agent avec learning rate constant dans l'environnement à variation continu (rouge), dans l'environnement référant au contexte [15] (marron) et l'agent scheduler (orange)

Problème 2 : courbes d'apprentissage moyenne

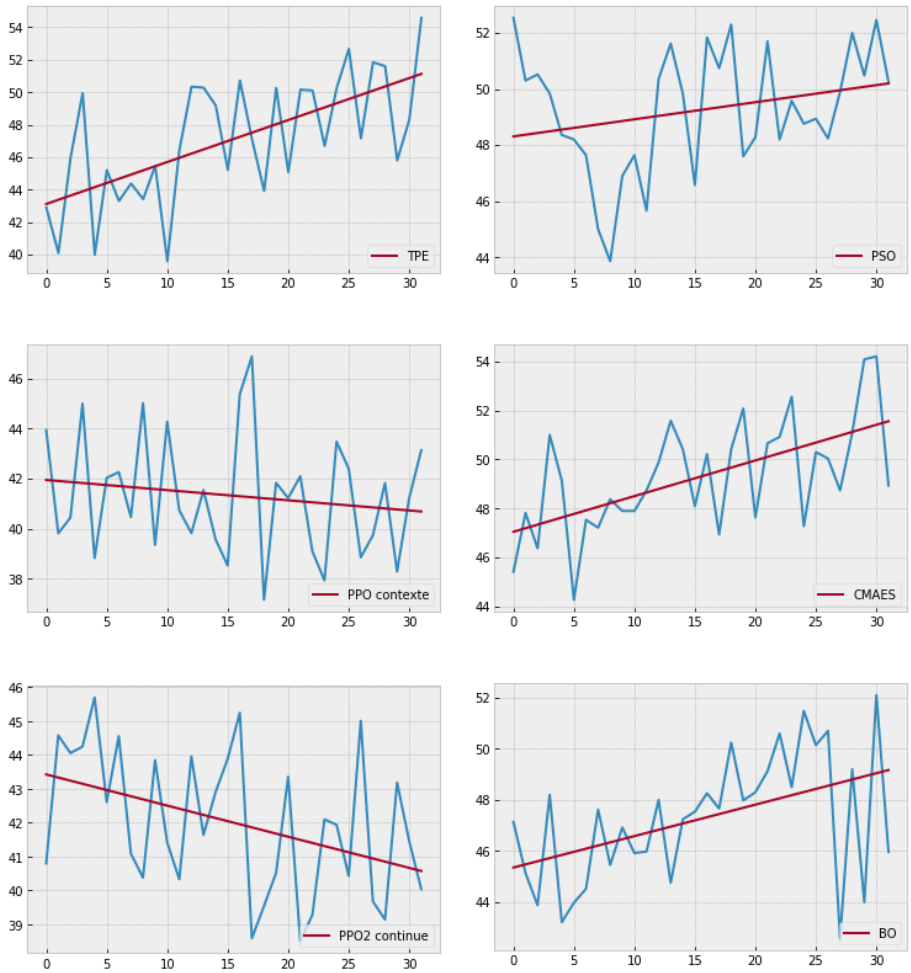

FIGURE 9 – Courbes d'apprentissage moyenne des différentes méthodes utilisées dans le problème 1 avec leur tendance (regression linéaire)