Sorbonne Université
Master ANDROIDE

Apprentissage par renforcement profond : analyse détaillée de SVPG

UE de projet M1

Julien CANITROT, Jules DUBREUIL, Tan Khiem HUYNH, Nikola KOSTADINOVIC

2022

Table des matières

Chapitre 1

Introduction

Stein Variational Policy Gradient (SVPG) (Y. Liu et al. 2017), est une méthode d'apprentissage par renforcement (*reinforcement learning* ou RL) (Sutton et Barto 2018), qui permet l'apprentissage et l'exploitation de plusieurs politiques. Plusieurs agents (que l'on appellera "particules" par la suite) travaillent en parallèle, ce qui accélère l'exploration. L'avantage de SVPG est qu'il empêche ces particules d'apprendre la même solution en les éloignant les unes des autres, ce qui favorise une plus grande diversité de solutions.

L'objectif de notre projet est ainsi, sous l'encadrement d'Olivier Sigaud et d'Oliver Serris, de moderniser cet algorithme en utilisant des outils plus modernes tels que PyTorch (Paszke et al. 2019), SaLinA (Denoyer et al. 2021) et OpenIA Gym (Brockman et al. 2016), de le comparer à d'autres méthodes de Policy Gradient classiques comme Advantage Actor Critic (A2C) (Mnih et al. 2016) ou REINFORCE (Williams 1992), de développer des outils de visualisation afin de reproduire les résultats obtenus dans l'article original et de mettre en avant les cas d'utilisation pertinents.

Lien vers notre implémentation : https://github.com/Anidwyd/pandroide-svpg.

État de l'art

Nous présentons ici les concepts nécessaires à la compréhension de notre projet.

2.1 Méthodes de Policy Gradient

Les méthodes de Policy Gradient, comme A2C ou REINFORCE, sont un type de techniques d'apprentissage par renforcement qui reposent sur l'optimisation de politiques paramétrées par rapport à la valeur de récompense attendue par descente de gradient. Cependant, ces méthodes souffrent généralement de variance élevée, de convergence lente ou d'exploration inefficace (Y. Liu et al. 2017). L'algorithme SVPG tente de répondre à ces problèmes en s'appliquant par dessus ces méthodes. Il appartient à la catégorie des méthodes d'apprentissage par renforcement distribué, une sous-partie du RL dans laquelle plusieurs particules travaillent en parallèle pour faciliter l'exploration des paramètres.

2.2 Stein Variational Gradient Descent (SVGD)

En apprentissage par renforcement, il est nécessaire de pouvoir approximer une distribution de probabilité sur les différentes actions possibles afin d'obtenir une idée sur les récompenses probables de nos politiques. Les méthodes usuelles pour ces approximations sont l'inférence variationnelle (*variational inference* ou VI) (Blei, Kucukelbir et McAuliffe 2017) et Markov Chain Monte Carlo Sampling (MCMC). SVPG lui, repose sur Stein Variational Gradient Descent (SVGD) (Q. Liu et Wang 2019), un algorithme d'inférence bayésienne des mêmes auteurs. Il combine les avantages de MCMC qui ne confine pas l'approximation dans une famille paramétrique, et de VI qui converge rapidement grâce à des mises à jour déterministes qui utilisent le gradient.

SVGD trouve un compromis entre une force *attractive*, appliquée par le gradient sur les différentes particules, et une force *répulsive* afin de garder une part d'exploration entre ces mêmes particules. Pour la répulsion, SVGD se base sur la méthode de Stein, méthode utilisée en théorie des probabilités afin d'obtenir des bornes sur des distances entre deux distributions selon une certaine divergence. À partir de l'*identité de Stein*, on peut caractériser l'écart $\mathbb{S}(p, q)$ entre différentes distributions p et q dites *smooth* (dérivables). Avec cet écart, nous pouvons faire réagir les particules les unes par rapport aux autres.

Dans les faits, ce calcul nécessite une optimisation variationnelle très complexe le rendant quasiment insoluble pour nos ordinateurs. C'est pourquoi, à partir d'un kernel, on définit une distance appelée Kernelized Stein Discrepancy (KSD) (Q. Liu, Lee et Jordan 2016) pour caractériser cet écart, que l'on combine avec des méthodes de Policy Gradient préexistantes (A2C, REINFORCE).

2.3 Stein Variational Policy Gradient (SVPG)

SVPG a déjà été testé de manière expérimentale avec A2C et REINFORCE. L'article original identifie les différents avantages de la méthode SVPG par rapport aux méthodes classiques d'apprentissage par renforcement. On comprend notamment que SVPG explore l'espace des paramètres de manière plus intelligente en exploitant les zones les plus fructueuses grâce à l'apprentissage commun des particules. Cela lui permet d'apprendre des politiques diverses et performantes. Ses performances sont surtout remarquables sur des environnements complexes comme CartPole Swing-Up et Double Pendulum.

Afin de gérer l'impact de la force répulsive, les auteurs introduisent un facteur de température α devant le terme de répulsion, qui permet de choisir entre une politique plutôt exploratrice ou exploitatrice. L'article insiste sur l'intérêt de tester différentes méthodes de sélection de α pour rendre l'algorithme plus adaptatif lors de son apprentissage.

Malheureusement l'implémentation existante de SVPG repose sur des outils de développement qui ne sont plus vraiment d'actualité (Theano) (Al-Rfou et al. 2016) voire obsolètes (Rllab) (Duan et al. 2016).

2.4 Sequential Learning of Agents (SaLinA)

Dans le cadre de notre projet, il est nécesaire d'utiliser des méthodes de décisions séquentielles. En effet, nous utilisons SVPG pour résoudre des problèmes de type "jeu" dont le but est de réagir aux réponses de l'environnement face à nos décisions (ou évolutions naturelles), de facon à maximiser un objectif.

Les bibliothèques de Deep Learning classiques (comme PyTorch) ne sont pas optimisées pour répondre à ces problèmes. Elles nécessitent d'implementer des outils suplémentaire qui définissent de nouvelles couches d'abstraction, pouvant rendre la prise en main plus compliquée.

SaLinA est une bibliothèque récente et prometteuse, basée sur PyTorch, qui a pour objectif de rendre l'implémentation de modèles d'apprentissage séquentiel plus naturelle. Pour ce faire, elle se base sur 2 principes :

1. tout est un agent, les méthodes s'appliquent donc séquentiellement à tous les agents,
2. les agents échangent des informations à travers un *workspace* accessible à tous, facilitant les interactions.

C'est pourquoi, pour moderniser et faciliter l'implémentation de l'algorithme, nous avons choisi de nous baser principalement sur SaLinA, PyTorch, Numpy (Harris et al. 2020) et Gym (Brockman et al. 2016).

Contribution

Nous décrivons maintenant les différentes étapes allant de la compréhension à l'implémentation du projet.

3.1 Besoins préalables

Pour pouvoir nous lancer dans le projet sereinement, avant de débuter l'implémentation de l'algorithme, nous avons accumulé un certain nombre de connaissances.

Dans un premier temps, afin de mieux comprendre les différents termes techniques, nous avons dû en apprendre plus sur le Machine Learning de manière générale. Les cours de RA ont été particulièrement utiles. Les premiers cours ayant été dispensés par notre encadrant M. Sigaud, il a pu nous donner des explications plus approfondies et nous orienter vers ses cours de M2. Il nous a aussi préparé plusieurs ressources (Google Collabs, slides...) sur SVPG et A2C. Nous avons ensuite étudié l'article de référence afin de comprendre les concepts et les résultats obtenus.

Dans un second temps, nous avons pris en main les différents outils. Nous avons appris à manipuler PyTorch et Gym à partir des différents exemples de leur documentation. Pour SaLinA nous avons lu le papier, étudié la documentation et suivi une série de vidéos explicatives [1] de l'auteur, Ludovic Denoyer. Ce dernier étant un ancien membre du Lip6, nous avons pu échanger avec lui pour nous éclairer sur certains points.

Enfin, les notions et les outils bien en tête, nous avons commencé le "décryptage" du code de référence à partir du dépôt original [2] et du collab d'Olivier Sigaud. À partir de là nous avons pu commencer l'implémentation de l'algorithme.

3.2 Implémentation

Notre implémentation générale se sur la classe `Algo`. Cette dernière initialise les différents attributs (agents, workspaces, environnements, optimizers...) et définit les méthodes communes de nos algorithmes (`execute_agents`, `run`...). Les algorithmes de Policy Gradient (`A2C`, `REINFORCE`) héritent de cette classe et implémentent le calcul de leurs différentes losses dans `compute_loss`. Cette fonction est appelée par la méthode `run` de la classe `Algo`. On peut ainsi lancer ces algorithmes de manière indépendante.

`SVPG` est une classe à part entière qui requiert un `Algo` (comme A2C) en paramètre. Ce dernier permet de calculer les losses requises par SVPG.

1. https ://youtube.com/playlist ?list=PLyiofmh1TG3tAZP_AFaPfuYeYCJSreIZy
2. https ://github.com/largelymfs/svpg_REINFORCE

3.2.1 A2C

N'ayant pas eu le temps d'implémenter REINFORCE complètement, pour nous focaliser sur la reproduction des résultats, décrivons notre implémentation d'A2C.

Cet algorithme est présenté par Ludovic Denoyer dans ses vidéos d'introduction à SaLinA. En plus de nous avoir familiarisé avec la notion d'agent, nous nous sommes inspirés de cette implémentation pour créer l'architecture globale d'un `Algo`. A2C étant un algorithme du type *actor-critic*, commençons donc par définir ses agents.

`ActionAgent`. Cet agent récupère les observations de l'environnement, détermine une distribution de probabilités sur les actions et en tire une (de manière stochastique ou déterministe). Si les actions sont discrètes, la distribution est déterminée par une couche SoftMax. Sinon, dans le cas continu, par une densité normale $X \sim \mathcal{N}(\mu, \sigma^2)$. L'agent cherche alors à optimiser μ et σ.

`CriticAgent`. Le critic évalue un état s à partir des observations, selon la fonction de valeur d'états $V(s)$.

Pour chaque particule, la boucle principale (`run`) exécute son action-agent et son critic-agent puis calcule une critic-loss, une A2C-loss et une entropy-loss. Comme dans l'article original, nous utilisons le *generalized advantage estimator* (GAE) (Schulman et al. 2018) pour le critic.

3.2.2 SVPG

Boucle principale. Pour détailler notre implémentation de SVPG, partons du pseudo-code de l'article :

Algorithme 1 : Stein Variational Policy Gradient

1 **Entrées** *Learning rate* ϵ, *kernel* $k(x, x')$, *temperature* α, *particules initiales* $\{\theta_i\}$.
2 **for** *iteration* $t = 0, 1, \ldots, T$ **do**
3 **for** *particule* $i = 0, 1, \ldots, n$ **do**
4 Calculer $\nabla\theta_i J(\theta_i)$ avec A2C
5 **end for**
6 **for** *particule* $i = 0, 1, \ldots, n$ **do**
7 $\Delta\theta_i \leftarrow \frac{1}{n}\sum_{j=1}^{n}[\nabla_{\theta_j}\left(\frac{1}{\alpha}J(\theta_j) + \log q_0(\theta_j)\right)k(\theta_j, \theta_i) + \nabla_{\theta_j}k(\theta_j, \theta_i)]$
8 $\theta_i \leftarrow \theta_i + \epsilon\Delta\theta_i$
9 **end for**
10 **end for**

Cet algorithme explicite le fonctionnement de la boucle principale de SVPG. Les lignes 3 à 5 correspondent au calcul du gradient de chaque particule par l'algorithme de Policy Gradient utilisé (ici A2C). De la même manière que décrit précédemment, nous implémentons cette étape en exécutant l'action-agent et le critic-agent de chaque particule, avant de calculer les différentes loss (A2C, critic, entropy) et de faire une descente de gradient sur le critic. Nous accumulons l'entropy-loss et l'A2C-loss pendant plusieurs pas de temps avant de pouvoir calculer le gradient de l'acteur. Ce dernier est donc mis à jour moins souvent que le critic.

Noyau SVGD. La partie SVGD représente une partie importante de SVPG. Dans notre cas, elle repose sur la création d'un kernel K qui projette les politiques dans un espace plus simple. Cela nous permet de mesurer la similarité entre deux politiques afin d'éloigner les particules les unes des autres. Le code pour la création du kernel nous a été fourni par Nicolas Castanet. Puisque c'est un point central de l'algorithme, détaillons son fonctionnement.

```
1  class RBF(nn.Module):
2
3      def forward(self, X, Y):
4          XX = X.matmul(X.t())
5          XY = X.matmul(Y.t())
6          YY = Y.matmul(Y.t())
7
8          dnorm2 = -2 * XY + XX.diag().unsqueeze(1) + YY.diag().unsqueeze(0)
9
10         np_dnorm2 = dnorm2.detach().cpu().numpy()
11         h = np.median(np_dnorm2) / (2 * np.log(X.size(0) + 1))
12
13         gamma = 1.0 / (1e-8 + 2 * h)
14         K_XY = (-gamma * dnorm2).exp()
15
16         return K_XY
```

Listing 3.1 – Calcul du noyau RBF.

K est un kernel RBF gaussien, $k(\theta_i, \theta_j) = \exp(-||\theta_i - \theta_j||^2/2h)$ tq. $h = med^2/2\log(n+1)$ où med est la médiane des distances par paires entre les particules. La classe `RBF` est définie comme un module PyTorch. Cela nous permet d'effectuer un appel sur la classe directement pour retourner un nouveau noyau.

Les deux paramètres `X`, `Y` de la méthode `forward()` sont identiques. Ils représentent une matrice qui contient les paramètres de toutes les politiques.

$$X = Y = \begin{pmatrix} \theta_1^1 & \theta_1^2 & \cdots & \theta_1^n \\ \theta_2^1 & \theta_2^2 & \cdots & \theta_2^n \\ \vdots & \vdots & \ddots & \vdots \\ \theta_n^1 & \theta_n^2 & \cdots & \theta_n^n \end{pmatrix},$$

avec $\theta_i = (\theta_i^1, \theta_i^2, ..., \theta_i^n)$ qui désigne les poids du réseau de neurones de l'action-agent de la particule i.

Nous avons besoin de 2 arguments pour la même matrice. En effet, lorsque nous utilisons `backward()` pour calculer le gradient du noyau, nous ne voulons pas que notre matrice de paramètres apparaisse deux fois dans le graphe de calcul. En utilisant 2 paramètres, nous pouvons calculer le noyau en appelant `RBF(X, X.detach())`. Pour rappel, `tensor.detach()` crée un tenseur qui partage le stockage avec un tenseur qui ne nécessite pas de gradient. En d'autres termes, il "détache" le calcul du gradient de la chaîne de calculs. Ainsi, nous nous assurons qu'aucun gradient ne sera rétro-propagé le long de cette variable. De $X = Y$ on déduit aussi que :

$$XX = XY = YY = XX^T.$$

Etudions la ligne 8 du code en détail. `XX.diag()` et `YY.diag()` désignent la diagonale de XX^T, qui est $(||\theta_1||^2, \dots, ||\theta_n||^2)$. `unsqueeze(1)` permet de la renvoyer comme vecteur colonne et `unsqueeze(0)` comme vecteur ligne.

Donc, `XX.diag().unsqueeze(1) + YY.diag().unsqueeze(0)` revient à calculer :

$$M = \begin{pmatrix} 2||\theta_1||^2 & ||\theta_1||^2 + ||\theta_2||^2 & \cdots & ||\theta_1||^2 + ||\theta_n||^2 \\ ||\theta_2||^2 + ||\theta_1||^2 & 2||\theta_2||^2 & \cdots & ||\theta_2||^2 + ||\theta_n||^2 \\ \vdots & \vdots & \ddots & \vdots \\ ||\theta_n||^2 + ||\theta_1||^2 & ||\theta_n||^2 + ||\theta_2||2 & \cdots & 2||\theta_n||^2 \end{pmatrix}.$$

La suite du calcul nous donne :

$$N = -2XY + M = \begin{pmatrix} 0 & ||\theta_1 - \theta_2||^2 & \cdots & ||\theta_1 - \theta_n||^2 \\ ||\theta_2 - \theta_1||^2 & 0 & \cdots & ||\theta_2 - \theta_n||^2 \\ \vdots & \vdots & \ddots & \vdots \\ ||\theta_n - \theta_1||^2 & ||\theta_n - \theta_2||^2 & \cdots & 0 \end{pmatrix}.$$

Après avoir calculé h, on obtient la matrice :

$$K = \exp(-\frac{N}{2h}) = \begin{pmatrix} \mathbf{k}(\theta_1, \theta_1) & \mathbf{k}(\theta_1, \theta_2) & \cdots & \mathbf{k}(\theta_1, \theta_n) \\ \mathbf{k}(\theta_2, \theta_1) & \mathbf{k}(\theta_2, \theta_2) & \cdots & \mathbf{k}(\theta_2, \theta_n) \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{k}(\theta_n, \theta_1) & \mathbf{k}(\theta_n, \theta_2) & \cdots & \mathbf{k}(\theta_n, \theta_n) \end{pmatrix}.$$

K est bien un noyau, avec pour diagonale $k(\theta_i, \theta_i) = 1, \forall i \in \{1, .., n\}$.

SVGD update. Découpons la mise à jour de la politique (ligne 7) en deux parties :

$$\frac{1}{n} \sum_{j=1}^{n} \Big[k(\theta_j, \theta_i) \nabla_{\theta_j} (\frac{1}{\alpha} J(\theta_j)) \Big], \tag{3.1}$$

$$\frac{1}{n} \sum_{j=1}^{n} \Big[\nabla_{\theta_j} k(\theta_j, \theta_i) \Big]. \tag{3.2}$$

La première partie (3.1) est calculée par la fonction :

```
1  def add_gradients(policy_loss, gamma, n_particles, kernel):
2      (-policy_loss * gamma / n_particles).backward(retain_graph=True)
3      for i in range(self.algo.n_particles):
4          theta_i = self.algo.action_agents[i].parameters()
5          for j in range(self.algo.n_particles):
6              if i == j: continue
7              theta_j = self.algo.action_agents[j].parameters()
8              for (wi, wj) in zip(theta_i, theta_j):
9                  wi.grad = wi.grad + wj.grad * kernel[j, i].detach()
```

Listing 3.2 – Première partie de la mise à jour SVGD.

Si on utilise `backward()` sur $\sum_{j=1}^{n}\left[k(\theta_j,\theta_i)(\frac{1}{\alpha}J(\theta_j))\right]$, on calculera bien :

$$\Delta\theta_i = \frac{1}{n}\sum_{j=1}^{n}\left[k(\theta_j,\theta_i)\nabla_{\theta_i}(\frac{1}{\alpha}J(\theta_j))\right], \forall i \in \{1,\dots,n\}.$$

Ainsi, puisque $\nabla_{\theta_i}J(\theta_j) = 0$, on a $\forall i \in \{1,\dots,n\}$:

$$\Delta\theta_i = \frac{1}{n}\left(k(\theta_i,\theta_i)\nabla_{\theta_i}(\frac{1}{\alpha}J(\theta_i))\right) = \frac{1}{\alpha n}\nabla_{\theta_i}J(\theta_i).$$

Par conséquent, afin de calculer correctement le gradient, nous devons d'abord utiliser `backward()` sur $-\sum_{j=1}^{n}\frac{1}{\alpha n}J(\theta_j)$. Si on le fait pas, `optimizer.step()` va soustraire le gradient alors que l'on souhaite l'ajouter lors de la mise à jour. On obtient donc, pour chaque particule i :

$$\Delta\theta_i = \frac{1}{\alpha n}\nabla_{\theta_i}J(\theta_i).$$

Enfin, on boucle sur toutes les autres particules j :

$$\Delta\theta_i = \Delta\theta_i + \Delta\theta_j k(\theta_j,\theta_i).$$

Calculons à présent le second terme de la mise à jour SVGD (3.2). Ici, si on utilise `backward()` sur la somme du kernel, on calculera en réalité, pour tous les paramètres de particule θ_i :

$$\sum_{j=1}^{n}\nabla_{\theta_i}k(\theta_i,\theta_j).$$

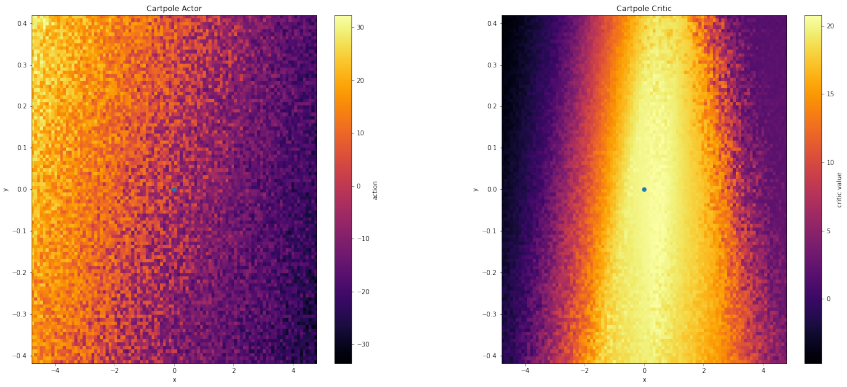
Notons que $\nabla_{\theta_j}k(\theta_j,\theta_i) = -\nabla_{\theta_i}k(\theta_i,\theta_j)$. Ainsi, avant d'exécuter `backward()`, pour calculer $\sum_{j=1}^{n}\left[\nabla_{\theta_j}k(\theta_j,\theta_i)\right]$ on utilise `-kernel.sum()` dans le calcul de la loss totale. Cependant, comme avec le premier terme (3.1), `optimizer.step()` va soustraire le gradient alors que dans la mise à jour SVGD on souhaite l'ajouter. C'est pourquoi on utilise `backward()` sur `kernel.sum() / n`.

```
...
params = get_policy_parameters() # Paramètres des particules
kernel = RBF()(params, params.detach())  # Calcul du kernel

add_gradients(policy_loss) # Calcul du premier terme

loss = -entropy_loss + kernel.sum() / n_particles
loss.backward() # Calcul du second

# Réinitialisation des losses
policy_loss = 0
entropy_loss = 0
critic_loss = 0 # plus fréquent...
...
```

Listing 3.3 – SVGD update en résumé

3.2.3 Outils de visualisation

Pour débugger et analyser les performances de nos algorithmes, nous avons développé plusieurs outils de visualisation. Pour visualiser les critics et les politiques apprises par les particules, nous avons adapté le code existant d'Olivier Sigaud (basé sur Stable Baselines3) (Raffin et al. 2021) pour SaLinA.



(a) Visualisation d'une politique sur CartPoleContinuous (A2C)

(b) Visualisation d'un critic sur CartPoleContinuous (A2C)

L'espace des états est représenté en 2 dimensions. Dans cet exemple, sur l'environnement CartPoleContinuous (customisé par Olivier Sigaud), l'axe x représente la position du cart et l'axe y l'angle du pôle. Pour la figure 3.1a la couleur représente l'action prédite par la politique de l'agent. Pour la 3.1b, la valeur du critic prédite par le critic-agent.

Avec ces visualisations, il est beaucoup plus facile d'interpréter les résultats de notre apprentissage. En effet, sur la figure 3.1a on voit que la politique finale correspond bien à ce qu'on aurait naturellement fait pour garder le pôle à la verticale. Le cart accèlère vers la droite lorsque que le pole à droite, et inversement. Ce qui explique la diagonale sur nos actions. Sur la figure 3.1b on voit que le critic valorise une position centrée du cart, ce qui est bien corrélé avec les observations de la figure 3.1a.

Pour visualiser les densités de visites des états (généralement en 4D), nous utilisons l'algorithme t-SNE pour réduire l'espace d'observation en 2D (comme dans l'article).

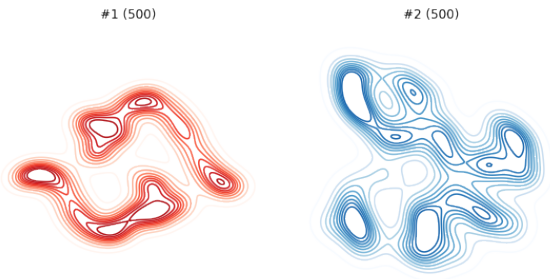


Figure 3.2 – Densités de visites des états sur CartPoleContinuous (A2C et SVPG).

On observe que les deux algorithmes ont appris des politiques valides (rewards de 500). Elles sont légèrement différentes mais gardent la même forme générale.

3.3 Expérimentations

Pour nous assurer de l'efficacité de SVPG, nous avons essayé de reproduire les résultats de l'article.

3.3.1 Paramètres et mise en contexte

Pour nous mettre dans les meilleures conditions de reproduction, nous avons fouillé l'article à la recherche des paramètres réglés par les auteurs. De cette manière, nous avons trouvé l'architecture des réseaux de neurones, le nombre de steps avant la mise à jour du gradient de l'acteur, et la plupart des hyper-paramètres. Cependant, cela n'étant pas suffisant, nous avons regardé dans leur code directement. Nous en avons profité pour vérifier qu'ils utilisaient bien les paramètres de l'article.

Malgré cela, plusieurs paramètres restaient inconnus, comme la fréquence de mise à jour du critic d'A2C, ou leur façon d'utiliser la technique du `annealed`. M. Sigaud leur a envoyé un mail à ce sujet, resté sans réponse.

Les environnements utilisés par les auteurs dépendent de Rllab, une librairie maintenant obsolète. Nous avons donc utilisé un *wrapper* afin de les encapsuler dans des environnements Gym, ce qui nous éloigne potentiellement des conditions expérimentales des auteurs. De plus, nous ne savons pas si ces environnement ont été modifiés depuis, ce qui altèrerait l'utilisation de leurs hyper-paramètres optimisés selon des conditions différentes.

Au vu de ces nombreux points de flous, il semblait compliqué de retrouver les résultats de l'article.

3.3.2 Normalisation des environnements

N'arrivant pas à reproduire les résultats, nous sommes retourné voir le code original. Nous avons ainsi pu relever que les auteurs normalisaient leurs environnements (actions, observations, rewards). Dans leurs expériences, une action a est normalisée entre les bornes de l'espace d'action $[l_b, u_b]$, selon la formule :

$$a' = \min\left\{u_b, l_b + \max\left\{l_b, \frac{1}{2}(a+1)(u_b - l_b)\right\}\right\}.$$

Les observations et les rewards (en apprentissage) sont normalisées en utilisant un *exponential moving average* (EMA) avec un coef $\alpha = 10^{-3}$. Cela permet de changer la manière d'apprendre, ce qui pourrait aider à trouver de meilleures politiques.

Cependant, ces normalisations ne rendaient pas nos résultats meilleurs, mais plutôt instables. Nous avons donc abandonné cette idée.

3.3.3 Optimisation avec Optuna

Au pied du mur, nous avons voulu vérifier que les rewards de l'article obtenues sur CartPoleSwingUp de Rllab (environ 400) sont bien atteignables avec A2C-Independent.

Pour cela, nous avons cherché à optimiser les hyper-paramètres de A2C avec Optuna (Akiba et al. 2019) et *RL Baselines3 Zoo* 2022. L'optimisation est réalisée par grid search ou bien des algorithmes aléatoires, bayésiens et évolutionaires.

Cependant, RL-Zoo ne permet pas l'optimisation sur des environnements Rllab. Ainsi, nous avons trouvé une version Gym de CartPoleSwingUp [3], que nous avons modifiée pour nous rapprocher au plus de la version Rllab.

Les meilleurs résultats de A2C sur cet environnement avoisinne les 60. Avec les hyper-paramètres optimisés, nous obtenions des résultats à peu près similaires qu'Optuna, à la fois sur le SwingUp de Gym et celui de Rllab. Cela souligne la difficulté de faire apprendre A2C sur cet environnement (Figure 3.3).

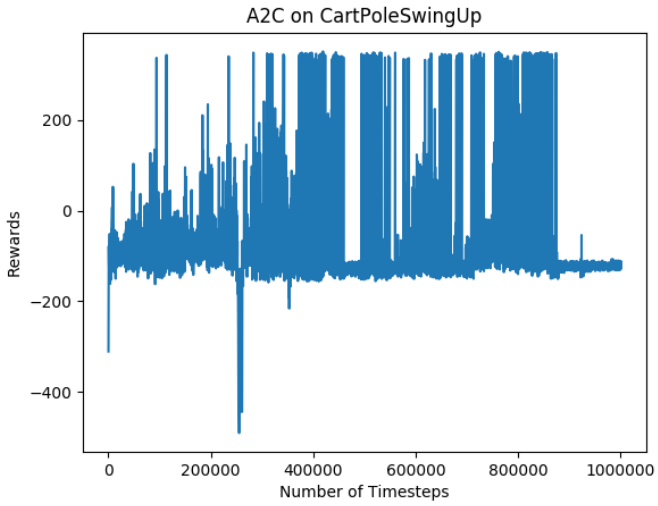


FIGURE 3.3 – Récompenses obtenues par la version SB3 de A2C sur CartPoleSwingUp.

Toujours loin des 400, nous avons répété l'expérience avec un autre algorithme, TD3 (FUJIMOTO, HOOF et MEGER 2018). Après optimisation cet algorithme convergeait bien aux alentours de 400 (Figure 3.4).

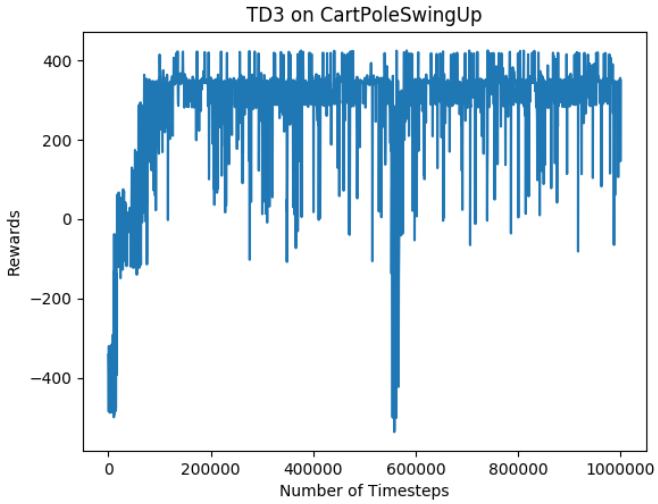


FIGURE 3.4 – Récompenses obtenues par la version SB3 de TD3 sur CartPoleSwingUp.

3. https ://github.com/0xangelo/gym-cartpole-swingup

3.4 Résultats

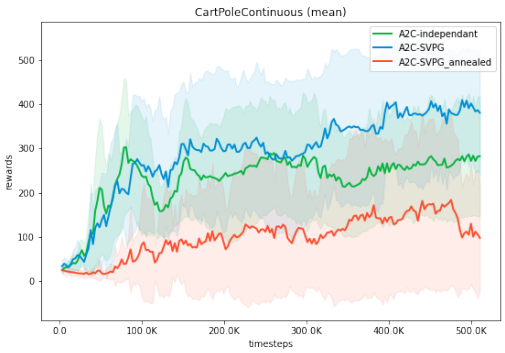
Pour analyser les performances de SVPG, l'article original implémente deux versions d'A2C. Une version *Independent*, où chaque particule exécute A2C indépendamment les unes des autres, ainsi qu'une version *Joint* où une seule particule reçoit toutes les données pour apprendre. D'après leurs résultats, cette dernière méthode performe toujours moins bien que A2C-Independent ou A2C-SVPG.

Le papier sur les méthodes d'annealing dans SVGD (D'Angelo et Fortuin 2021), souligne que SVGD possède des soucis d'exploration, surtout en grande dimension. Nous avons donc implémenté une version *annealed* de SVPG, qui fait varier la température α en cours d'expérience. Cela nous permet de favoriser la répulsion des particules en début d'apprentissage avant de la réduire plus tard pour permettre la convergence. En détail, la température α est controlée par cette formule :

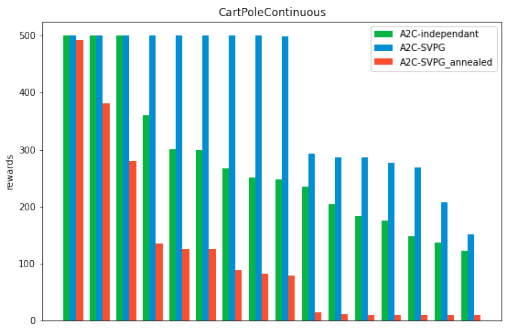
$$\frac{1}{\alpha} = tanh\big[5(\frac{t}{T})^{10}\big],$$

avec t le time step courant et T le nombre total de time step. C'est pourquoi nous avons décidé de comparer une version *annealed* de SVPG, plutôt qu'une version *Joint* de A2C.

3.4.1 CartPoleContinuous



(a) Récompenses moyennes des algorithmes sur CartPoleContinuous en apprentissage sur CartPoleContinuous.



(b) Récompenses moyennes des particules en évaluation avec un seed différent sur CartPoleContinuous.

Sur la figure 3.5a, on peut observer que SVPG et en moyenne plus performant que A2C, ce qui est confirmé par l'histogramme (3.5b). Ces courbes, bien que réalisées sur un environnement plus simple que Cartpole Swing-Up, démontrent que SVPG est capable d'améliorer la convergence d'A2C, ce qui correspond aux résultats du papier.

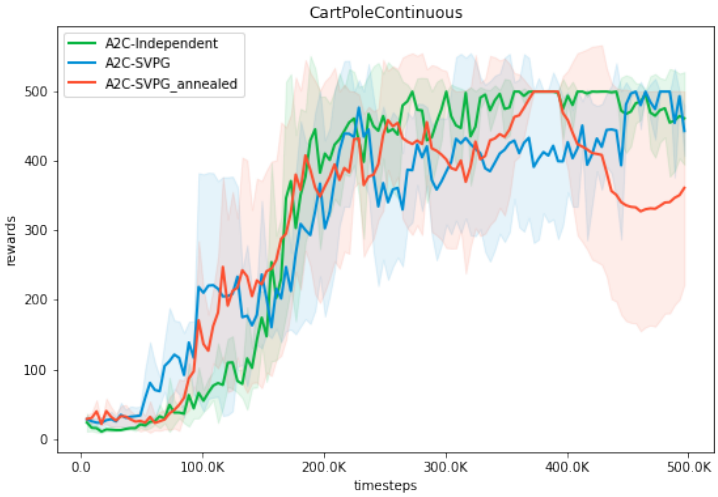
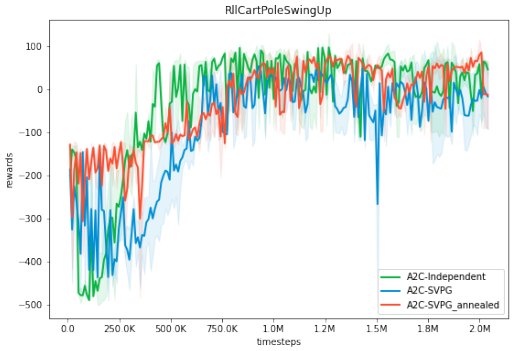


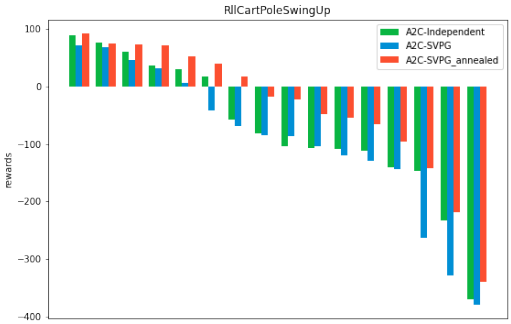
FIGURE 3.6 – Récompenses moyennes des meilleures particules en apprentissage sur 4 runs de CartPoleContinuous.

Si l'on se concentre uniquement sur les meilleures particules (Figure 3.6), comme dans l'article, nous retrouvons un comportement similaire à celui observé par les auteurs.

3.4.2 CartPoleSwingUp



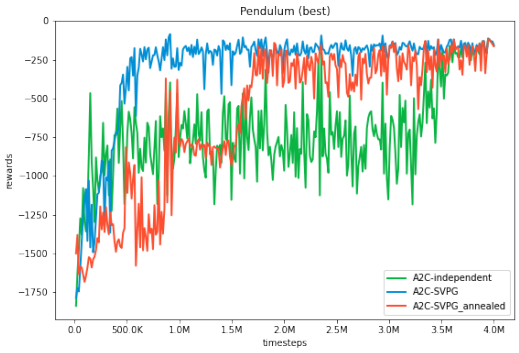
(a) Récompenses moyennes des meilleures particules en apprentissage sur 2 runs de CartPoleSwingUp.



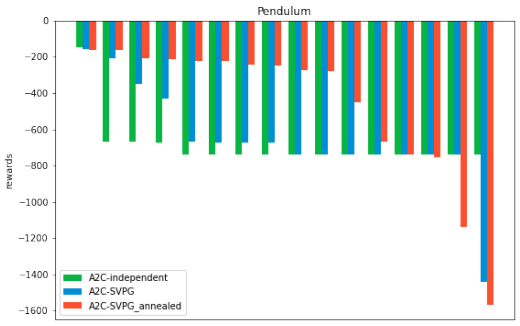
(b) Récompenses moyennes des particules en évaluation avec un seed différent sur CartpoleSwingUp.

Nous avons obtenu ces résultats (Figures 3.7a et 3.7b) en utilisant les hyper-paramètres suggérés par Optuna pour A2C sur notre Cartpole Swing-Up modifié. Nous retrouvons une allure similaire à celle d'A2C de SB3 (Figure 3.3), confirmant la difficulté d'apprendre dans cet environnement. Cependant, ces résultats ne nous permettent pas de confirmer les résultats de l'article. Cela pose la question de la possibilité de reproduire ces résultats avec les informations fournies.

3.4.3 Pendulum



(a) Récompenses moyennes des meilleures particules en apprentissage sur 2 runs de Pendulum.



(b) Récompenses moyennes des particules en évaluation avec un seed différent sur Pendulum.

Sur cet environnement on voit la nette supériorité des versions SVPG. En effet, ils arrivent à trouver une politique optimale. La version A2C quant à elle reste bloquée dans un optimum local beaucoup plus longtemps. Elle finit quand même par trouver la politique optimale à la fin de l'exécution (Figure 3.8a).

On constate de plus sur la figure 3.8b, l'intérêt du *annealed* qui réduit l'effet de répulsion vers la fin de l'apprentissage. Cela permet à beaucoup plus de particules de trouver la politique optimale.

3.4.4 Discussion

Malheuresement, nos résultats sur l'environnement Cartpole Swing-Up n'ont pas été concluant. Plus d'informations concernant les conditions expérimentales sont nécessaires.

Cependant, sur les deux autres environnements (Cartpole Continuous et Pendulum) les résultats sont plutôt rassurants. Ils permettent de montrer l'efficacité de SVPG à améliorer la convergence des algorithmes de Policy Gradient (spécifiquement A2C), ainsi que l'inconsistance des versions *annealed*.

Ces dernières donnent des résultats tantôt décevants (Figure 3.5a) tantôt encourageants (Figure 3.8b). Le critère de l'évolution du coefficient de température α doit probablement, lui aussi, être adapté à l'environnement.

Conclusion

Durant ce projet centré sur l'algorithme Stein Variational Policy Gradient (SVPG) (Y. Liu et al. 2017) nous avons fait une plongée dans l'univers du Deep Reinforcement Learning (Deep RL) (Sutton et Barto 2018). Avant de commencer l'implémentation de cet algorithme nous avons dû assimiler et manipuler de nombreux concepts avancés comme l'inférence variationnelle (VI) (Blei, Kucukelbir et McAuliffe 2017), la Kernelized Stein Discrepancy (KSD) (Q. Liu, Lee et Jordan 2016) ou encore le noyau SVGD (Q. Liu et Wang 2018).

Après cela nous avons implémenté l'algorithme d'apprentissage. Les résultats étaient prometteurs, SVPG apprenait dans les environnements les plus simples confirmant son implémentation correcte. Une supériorité de SVPG sur A2C (Mnih et al. 2016) semblait elle aussi se dégager dans les résultats, comme suggéré par l'article original. Durant cette phase de test nous avons mis en place plusieurs outils de visualisation intéressants pour la compréhension du fonctionnement des algorithmes.

Cependant nous nous sommes heurté à la difficulté de reproduire les résultats de l'article puisque la totalité des informations nécessaires n'était pas accessible. D'autant plus que nous sommes dans un domaine de recherche de l'informatique où tout évolue très vite. Des bibliothèques deviennent obsolètes (Rllab) (Duan et al. 2016) et de nouvelles, comme SaLinA (Denoyer et al. 2021), apparaissent. Au final, nous n'aurons pas réussi à reproduire exactement les courbes de l'article mais beaucoup de résultats restent intéressants.

Pour l'avenir de ce projet, il serait intéressant d'implémenter d'autres algorithmes de Policy Gradient comme REINFORCE (testé dans l'article) ou bien certains encore jamais expérimentés.

Ce projet fut pour nous une introduction au monde de la recherche. Il nous a permis de manipuler des notions et des outils (PyTorch, OpenIA Gym) (Paszke et al. 2019) (Brockman et al. 2016) essentiels du RL. Enfin, les difficultés rencontrées ont permis de mettre en évidence l'importance de détailler au maximum les conditions expérimentales, pour faciliter la reproductibilité de ses résultats.

Liu, Yang, Prajit Ramachandran, Qiang Liu et Jian Peng (avr. 2017). « Stein Variational Policy Gradient ». en. In : URL : https://arxiv.org/abs/1704.02399v1 (visité le 22/02/2022) (pages 1, 2, 15).

Sutton, Richard S. et Andrew G. Barto (2018). *Reinforcement learning : an introduction*. Second edition. Adaptive computation and machine learning series. Cambridge, Massachusetts : The MIT Press. ISBN : 978-0-262-03924-6 (pages 1, 15).

Paszke, Adam, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai et Soumith Chintala (déc. 2019). « PyTorch : An Imperative Style, High-Performance Deep Learning Library ». In : *arXiv :1912.01703 [cs, stat]*. arXiv : 1912.01703. URL : http://arxiv.org/abs/1912.01703 (visité le 03/04/2022) (pages 1, 15).

Denoyer, Ludovic, Alfredo de la Fuente, Song Duong, Jean-Baptiste Gaya, Pierre-Alexandre Kamienny et Daniel H. Thompson (oct. 2021). « SaLinA : Sequential Learning of Agents ». In : arXiv : 2110.07910. URL : http://arxiv.org/abs/2110.07910 (visité le 26/02/2022) (pages 1, 15).

Brockman, Greg, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang et Wojciech Zaremba (juin 2016). « OpenAI Gym ». In : *arXiv :1606.01540 [cs]*. arXiv : 1606.01540. URL : http://arxiv.org/abs/1606.01540 (visité le 03/04/2022) (pages 1, 3, 15).

Mnih, Volodymyr, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver et Koray Kavukcuoglu (juin 2016). « Asynchronous Methods for Deep Reinforcement Learning ». In : *arXiv :1602.01783 [cs]*. arXiv : 1602.01783. URL : http://arxiv.org/abs/1602.01783 (visité le 26/02/2022) (pages 1, 15).

Williams, Ronald J. (mai 1992). « Simple statistical gradient-following algorithms for connectionist reinforcement learning ». en. In : *Machine Learning* 8.3, p. 229-256. ISSN : 1573-0565. DOI : 10.1007/BF00992696. URL : https://doi.org/10.1007/BF00992696 (visité le 09/05/2022) (page 1).

Blei, David M., Alp Kucukelbir et Jon D. McAuliffe (avr. 2017). « Variational Inference : A Review for Statisticians ». In : *Journal of the American Statistical Association* 112.518. arXiv : 1601.00670, p. 859-877. ISSN : 0162-1459, 1537-274X. DOI : 10.1080/01621459.2017.1285773. URL : http://arxiv.org/abs/1601.00670 (visité le 03/04/2022) (pages 2, 15).

Liu, Qiang et Dilin Wang (sept. 2019). « Stein Variational Gradient Descent : A General Purpose Bayesian Inference Algorithm ». In : *arXiv :1608.04471 [cs, stat]*. arXiv : 1608.04471. URL : http://arxiv.org/abs/1608.04471 (visité le 03/04/2022) (page 2).

Liu, Qiang, Jason D. Lee et Michael I. Jordan (juill. 2016). « A Kernelized Stein Discrepancy for Goodness-of-fit Tests and Model Evaluation ». In : *arXiv :1602.03253 [stat]*. arXiv : 1602.03253. URL : http://arxiv.org/abs/1602.03253 (visité le 09/05/2022) (pages 2, 15).

Al-Rfou, Rami et al. (mai 2016). « Theano : A Python framework for fast computation of mathematical expressions ». In : *arXiv e-prints* abs/1605.02688. URL : http://arxiv.org/abs/1605.02688 (page 3).

Duan, Yan, Xi Chen, Rein Houthooft, John Schulman et Pieter Abbeel (mai 2016). « Benchmarking Deep Reinforcement Learning for Continuous Control ». In : *arXiv :1604.06778 [cs]*. arXiv : 1604.06778. URL : http://arxiv.org/abs/1604.06778 (visité le 10/05/2022) (pages 3, 15).

Harris, Charles R., K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke et Travis E. Oliphant (sept. 2020). « Array programming with NumPy ». en. In : *Nature* 585.7825. Number : 7825 Publisher : Nature Publishing Group, p. 357-362. ISSN : 1476-4687. DOI : 10.1038/s41586-020-2649-2. URL : https://www.nature.com/articles/s41586-020-2649-2 (visité le 10/05/2022) (page 3).

Schulman, John, Philipp Moritz, Sergey Levine, Michael Jordan et Pieter Abbeel (oct. 2018). *High-Dimensional Continuous Control Using Generalized Advantage Estimation*. Rapp. tech. arXiv :1506.02438. arXiv :1506.02438 [cs] type : article. arXiv. DOI : 10.48550/arXiv.1506.02438. URL : http://arxiv.org/abs/1506.02438 (visité le 19/05/2022) (page 5).

Raffin, Antonin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus et Noah Dormann (2021). « Stable-Baselines3 : Reliable Reinforcement Learning Implementations ». In : *Journal of Machine Learning Research* 22.268, p. 1-8. URL : http://jmlr.org/papers/v22/20-1364.html (page 9).

Akiba, Takuya, Shotaro Sano, Toshihiko Yanase, Takeru Ohta et Masanori Koyama (juill. 2019). *Optuna : A Next-generation Hyperparameter Optimization Framework*. Rapp. tech. arXiv :1907.10902. arXiv :1907.10902 [cs, stat] type : article. arXiv. DOI : 10.48550/arXiv.1907.10902. URL : http://arxiv.org/abs/1907.10902 (visité le 20/05/2022) (page 10).

RL Baselines3 Zoo (mai 2022). *RL Baselines3 Zoo : A Training Framework for Stable Baselines3 Reinforcement Learning Agents*. original-date : 2020-05-05T05 :53 :27Z. URL : https://github.com/DLR-RM/rl-baselines3-zoo (visité le 20/05/2022) (page 10).

Fujimoto, Scott, Herke van Hoof et David Meger (oct. 2018). *Addressing Function Approximation Error in Actor-Critic Methods*. Rapp. tech. arXiv :1802.09477. arXiv :1802.09477 [cs, stat] version : 3 type : article. arXiv. DOI : 10.48550/arXiv.1802.09477. URL : http://arxiv.org/abs/1802.09477 (visité le 20/05/2022) (page 11).

D'Angelo, Francesco et Vincent Fortuin (mars 2021). « Annealed Stein Variational Gradient Descent ». In : *arXiv :2101.09815 [cs]*. arXiv : 2101.09815. URL : http://arxiv.org/abs/2101.09815 (visité le 03/04/2022) (page 12).

Liu, Qiang et Dilin Wang (oct. 2018). « Stein Variational Gradient Descent as Moment Matching ». In : *arXiv :1810.11693 [cs, stat]*. arXiv : 1810.11693. URL : http://arxiv.org/abs/1810.11693 (visité le 09/05/2022) (page 15).

Chapitre A

Cahier des charges

Lors de la première réunion avec notre encadrant, Olivier Sigaud, nous nous sommes fixé les objectifs suivants :

— comprendre l'algorithme SVPG ;
— moderniser son implémentation avec SaLinA ;
— développer des outils de visualisation ;
— reproduire les résultats de l'article original ;
— mettre en avant les cas d'utilisation pertinents.

Manuel utilisateur

B.1 Installation

Pour installer notre projet il suffit de cloner le dépôt github :

```
git clone https://github.com/Anidwyd/pandroide-svpg.git
pip install -e ./pandroide-svpg/
```

Nous recommandons de plus d'installer les librairies suivantes : my_gym, rllab, pybox2D, et le fork d'Olivier Sigaud de salina.

B.2 Utilisation

Des exemples sont disponibles dans le répertoire `tests`. Dans la pratique, on utilise une configuration qui permet de définir les hyper-paramètres des algorithmes, la structure des réseaux de neurones des agents, l'environnement, l'optimizer...

```
config = OmegaConf.create({
    "logger": {
        "classname": "salina.logger.TFLogger",
        "log_dir": "./tmp/",
        "verbose": False,
    },
    "algorithm": {
        "n_particles": 16,
        "seed": 4,
        "n_envs": 8,
        "n_steps": 100,
        "eval_interval": 4,
        "n_evals": 1,
        "clipped": True,
        "max_epochs": 625,
        "discount_factor": 0.95,
        "gae_coef": 0.8,
        "policy_coef": 0.1,
        "entropy_coef": 0.001,
        "critic_coef": 1.0,
        "architecture": {"hidden_size": [64, 64]},
    },
    "gym_env": {
        "classname": "svpg.agents.env.make_gym_env",
        "env_name": "CartPoleContinuous-v1",
    },
    "optimizer": {"classname": "torch.optim.Adam", "lr": 0.01},
})  # Il est aussi possible d'utiliser hydra
```

Listing B.1 – Exemple de configuration

On peut ensuite créer des algorithmes défini dans le packague `svpg.algos`. Pour SVPG, le paramètre `is_annealed` permet d'activer l'option d'annealing ou non. Pour lancer un algorithme, on utilise sa fonction `run`.

```
from svpg.algos import A2C, SVPG

a2c = A2C(config) # A2C-Independent
a2c.run()
svpg = SVPG(A2C(config), is_annealed=False).run()  # A2C-SVPG
SVPG(A2C(config), is_annealed=True).run()  # A2C-SVPG_annealed
```

Listing B.2 – Exemple d'utilisation des algos

B.3 Outils

Le package `utils` offre plusieurs utilitaires permettant de sauvegarder et charger un algorithme, ou encore d'évaluer un agent.

```
from svpg.utils.utils import save_algo, load_algo
directory = "./runs/A2C"

# Sauvegarde / chargement de A2C
a2c.save_algo(directory)
action_agents, critic_agents, rewards, timesteps = load_algo(directory)

# Evaluation des politiques
eval_rewards = eval_agents_from_dir(directory, n_eval=100, seed=432)
```

Listing B.3 – Exemple d'utilisation des utilitaires

La visualisation des critics et des politiques à été directement été incluse dans le fork d'Olivier Sigaud de salina.

```
from svpg.agents.env immport make_gym_env
env = make_gym_env("CartPoleContinuous-v1")

# Visualisation des critics / politiques
plot_cartpole_critic(critic_agents[0], env, "critic0", directory)
plot_cartpole_policy(action_agents[0], env, "policy0", directory)

# Visualisation des histogrammes
rewards = eval_agents_from_dir(directory)
plot_histograms(rewards, "CartPoleContinuous-v1", save_fig=True)

# Calcul des espaces réduits avec t-SNE
embedded_spaces, rewards = get_embedded_spaces(directory, algo_name="A2C",
    n_eval=100)
# Visualisation des densités de visites des états
plot_state_visitation(directory, a2c_spaces, np.array(a2c_rewards),
    "A2C-Independent", cmap="Reds")
```

Listing B.4 – Exemple d'utilisation des fonctions de visualisation