TP IHMxIA: Interpretable Deep Reinforcement Learning

Dans ce tp nous allons étudier le comportement d'un modèle DQN, chargé de résoudre des tâches incluses dans le simulateur VizDOOM

Rendu du tp

Ce projet, non-noté, peut être fait en binôme. Le code devra cependant être déposé dans un repo github avec un ***readme.md*** à la racine affichant les ***visualisations obtenues par questions***. Le lien du répo doit être envoyé par mail à theo.jaunet@insa-lyon.fr. En cas de modifications particulières, le repo github devra contenir un readme.md à la racine précisant comment votre code doit être lancé. Le code doit impérativement être fonctionnel en python3. Le tp doit être rendu ***avant le prochain cours***.

Prérequis et installation.

Ce projet, utilise les mêmes dépendances que le projet DRL du module IBI. Il est donc envisageable d'utiliser comme base le même environnement virtuel. En parallèle, du code permettant de charger un modèle de DQN pré-entraîné, d'en faire des vidéos, ainsi que quelques fonctions nécessaires pour la suite du tp, est disponible sur ce répo github. Un fichier requierements.txt est fourni à la racine de ce répo, il permet d'installer les dépendances python comme il suit:

```
pip install -r requirement.txt
```

Il est toutefois recommandé de lancer cette commande dans un environnement python virtuel. Certaines librairies supplémentaires peuvent être requises par Vizdoom. Dans ce cas, il faut se référer aux instructions fournies sur ce répo, en fonction de votre OS. Une fois tout installé, il est possible de tester l'environnement avec la commande suivante :

```
python run.py
```

Ce script crée une vidéo d'un épisode dans le dossier vidéo à la racine. C'est dans ce dossier que tous les résultats des cartes de saliences seront affichés. Il y a également un dossier images pouvant être utilisé comme débug.

Saliency Maps

Les cartes de salience, font partie des premières visualisation de réseaux neuronal convolutionels (CNNs). Initialement, et comme demandé dans la première question de cette section, cette visualisation se résumait à l'affichage du gradient issu de la back-propagation. Ce gradient est ensuite affiche au-dessus de l'image en input, ainsi, les portions de l'input avec un gradient élevé peuvent être interprétés comme responsable de la décision du modèle. Bien que basique, il existe de nos jours de nombreuses variantes de cette visualisation (e.g. gradients intégrés, grad-CAM, smooth grad, etc..). Voici quelques ressources pour aider à la compréhension : Striving For Simplicity The All Convolutional Net, et une Vidéo explicative.

Une fois généré, il est possible d'afficher le gradient comme il suit :

```
    saliency = Image.fromarray(grads, 'L')
    saliency.save("images/saliency_" + str(nb) + ".jpg")

    grads = format_saliency(saliency)

    imgs.append(merge_img(state, grads, nb))

make_movie(imgs, "video/video_" + str(epoch + 1) + ".mp4")
```

Vous pouvez ajuster l'opacité de l'overlay dans la fonction format_saliency. Cette fonction utilise en entrée l'image d'input et le gradient (attention cependant, le gradient et l'image doivent être un array numpy int8 normalisé entre 0 et 255).

Questions:

- **1.** Visualiser le Gradient w.r.t. l'action choisie par le modèle.

Pour cela, vous devez modifier le script run.py afin de générer, pour chaque

forward d'un épisode une back-propagation de l'action choisie. Il est possible de préciser l'action en utilisant la fonction pytorch

```
output.backward(gradient=target)
```

Où target est une variable contentant un one-hot vector de l'action voulue (documentation). Il faut également s'assurer que

```
require_grad = true
```

pour l'input, et également qu'il soit détaché du graphe de calcul après le backward (sinon il sera pris en compte dans la suite des calculs).

```
tensor.detach_()
```

Le résultat de cette question doit être une vidéo de la vue de l'agent avec les cartes de salience affichée en overlay.

- **2.** Modifier le code pour n'afficher que les portions positives du gradient (Guided back-propagation).

Présentement, la carte de salience est très bruitée, il est donc difficile d'interpréter sur quelles portions de l'image l'agent se concentre. Cela vient du fait que toutes les valeurs du gradient positives et négatives sont affichées. Hors nous voulons uniquement afficher les portions de l'image favorisant l'action choisie (i.e. les valeurs positives.).

- **(Bonus)** Modifier le code pour n'afficher que les portions négatives du gradient (Guided back-propagation) La même chose que la question précédente peut être fait pour mettre en valeur les portions de l'image qui indique l'agent de ne pas faire l'action choisie.

- **(Bonus)** Comparer les cartes pour différentes actions (autre que celle choisie par le modèle)

Pour aller plus loin, je recommande la lecture de cet article interactif.

Projection Umap

Comme présenté dans le premier papier introduisant les DQNs sur atari, la réduction de dimensions permet une projection en 2D des états tels que vus par l'agent. Traditionnellement, cette projection l'algorithme t-SNE, qui par itération rapproche les états similaires, et éloigne les états différents. Bien que très efficace, cette approche possède des limites sur le plan computationnel avec des temps de calcul pouvant durer plus d'un jour en fonction des données. De plus, t-SNE ne supporte pas l'ajout de point à postériori. C'est pourquoi ici l'utilisation de umap est recommandé.

1. Projeter la sortie de la dernière couche de convolutionnel

Pour cela, installer la librairie python suivante: https://github.com/lmcinnes/umap. Ensuite, il faut modifier le code du modèle pour que le forward retournes, en plus de l'action choisie, le résultat de la dernière couche convolutionnelle sous la forme d'un vecteur. La prochaine étape est la collection de données. Commencez avec la génération des données avec environs 1000 étapes simulateur. Vous pouvez par la suite ajuster le nombre d'étapes afin d'avoir une visualisation plus globale..

Effectuer la projection avec la commande suivante (comme précisé par la librairie python):

```
embedding = umap.UMAP(n_neighbors=5,
    min_dist=0.3,
    metric='correlation').fit_transform(mydata)
```

Où ***mydata*** est une liste indexant les vecteurs du modèle.

Enfin, pour l'affichage, vous pouvez utiliser la fonction déjà donnée ***embedding2csv***. Cette fonction utilise en entrée, le résultat de la projection et le convertit en fichier CSV à la racine du projet. Il y a également un autre argument optionnel permettant de préciser une autre colonne dans le fichier CSV qui est ensuite utilisé comme couleur dans la projection. Si précisé, cet argument doit-être une liste avec une longueur égale à celle de ***embedding***. On peut, par example colorier les éléments par action choisie par l'agent, où par récompense obtenue par état.

```
embedding2csv(embedding,actions)
```

Vous pouvez ensuite ouvrir la page html ***projection.html*** dans votre navigateur pour voir la projection. Cette visualisation est faite en d3.js, vous pouvez donc la modifier à votre convenance.

- **2.** Faites varier les paramètres n_neighbors et min_dist afin d'obtenir la visualisation la plus lisible possible.

  Pour cela, vous pouvez vous aider de ce blog. Oubliez pas de mentionner dans le readme les meilleurs paramètres trouvés.

- **3.** Colorier les états par action choisie, récompense obtenue, ou Q-valeur de l'action choisie.

  Pour cela, préciser le second argument de ***embedding2csv*** avec la liste d'actions collectée en même temps que les vecteurs. Ainsi l'index 0 de ***embedding*** corresponds à l'index 0 de la liste d'actions.