Compte Rendu

TP2 — Intelligence Bio-Inspirée

Apprentissage Profond & Renforcement

Etudiants : Anthony BACCUET, Valentin BERGER

Langage : Python 🐍

IDE : PyCharm

Framework : PyTorch 🔥

Versioning : Git et GitHub

Lien GitHub : https://github.com/Cynnexis/tp-ibi-reinforcement

Contexte de Réplicabilité : Virtual-Environment, Docker🐳

Sommaire

Partie 1

Exécution du Script

Le projet fut réalisé avec git, docker et Virtual-Environment. Premièrement, vous pouvez cloner le projet en faisant :

```
git clone https://github.com/Cynnexis/tp-ibi-reinforcement.git
```

Ensuite, vous pouvez lancer le code Python soit en utilisant un environnement virtuel :

```
# Activer l'environnement virtuel
pip install -r requirements.txt
```

Soit en utilisant Docker :

```
docker-compose up -d
```

L'exécution de la partie 2 se réalise ensuite de la manière suivante :

```
python qlearning.py
```

Et l'exécution de la partie 3:

```
python breakout.py
```

⚠**Attention** : L'image docker crée par le `Dockerfile` utilise l'image `pytorch/pytorch:1.3-cuda10.1-cudnn7-runtime`, pesant 2Go. Aussi, il est déconseillé d'utiliser docker avec matplotlib dû au manque de GUI.

Partie 2

Début

Question 1

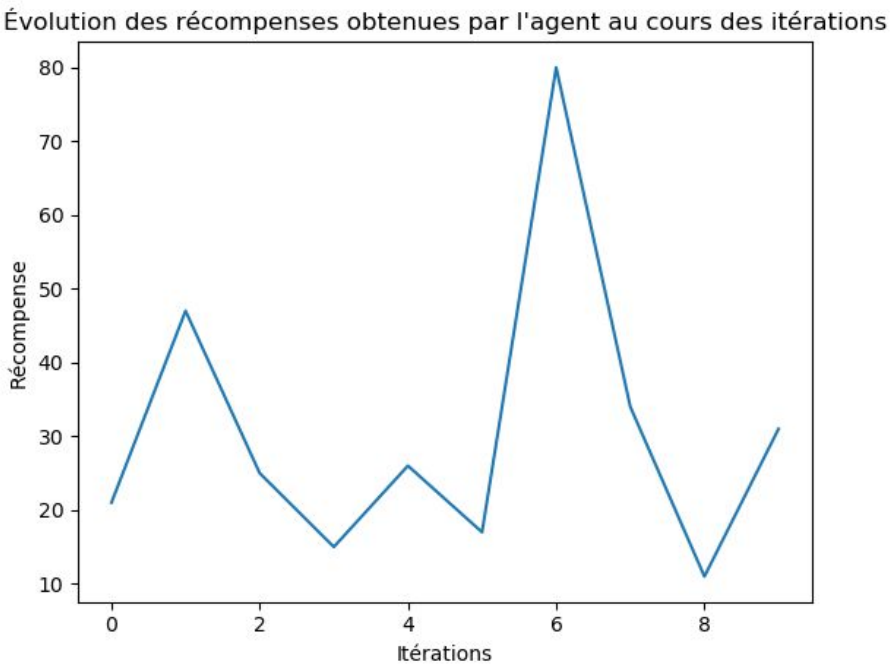
A l'aide de la documentation sur Gym, nous avons conçu l'agent suivant :

```
class RandomAgent(object):
    def __init__(self, action_space):
        self.action_space = action_space

    def act(self, observation, reward, done):
        return self.action_space.sample()
```

Question 2

Pour suivre les récompenses obtenues par l'agent au cours du temps, nous utilisons matplotlib de manière obtenir un graphe comme ci-dessous :



Pour chaque itération, on sauvegarde la somme des récompenses dans un tableau que l'on affiche à la fin de la simulation.

Expérience Replay

Question 3

On utilise une liste deque comme buffer afin de stocker nos interactions. L'avantage de ce type est que nous pouvons ajouter un élément au début ou à la fin de la liste, et si la taille maximale est atteinte, l'élément situé à l'autre bout de la queue est supprimé. Du fait que deque n'est pas un tableau, son appel est très rapide (mais plus coûteux en termes de mémoire, ce qui est négligeable pour ce projet).

On instancie notre buffer de la manière suivante :

```
d = deque(maxlen=100)
```

Et on ajoute des interactions comme suit :

```
d.append({"state": last_state, "action": action, "next_state":
env.state, "reward": reward, "end_ep": done})
```

Question 4

Pour faire un sampling aléatoire, nous utilisons la fonction suivante :

```
def sampling(buffer, batch_size):
    return random.sample(list(buffer), batch_size)
```

Deep Q-Learning

Question 5

Le réseau de neurone que l'on crée est une simple couche de 2 neurones prenant 4 entrées (l'observation) et renvoyant 2 sorties (la Q-valeur des actions) :

```
class ApproxQValue(nn.Module):
    """
    Neural network that predicts the q-values for all actions
    for a given state.
    """
    def __init__(self, input_size, output_size, activation:
Callable[[Any], Any] = nn.functional.relu):
        super(ApproxQValue, self).__init__()
        hidden_size = int(np.ceil((input_size + output_size) / 2))
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, output_size)
        self.activation = activation

    def forward(self, x):
        x = self.activation(self.fc1(x))
        x = self.activation(self.fc2(x))
        return x

# Création du réseau de neurone
neural_network = ApproxQValue(4, 2)
# Prédiction
Qaction = neural_network(torch.tensor(obs)).tolist()
```

Question 6

Nous utilisons le modèle de la Question 5 pour pouvoir donner la possibilité d'explorer :

```
Qaction = (neural_network(torch.tensor(obs)))
_, action = torch.max(Qaction, 0)
```

```
action = action.item()
rand = random.random()
if rand < self.t:
   rand = random.random()
   if rand < 0.5:
       return 0
   else:
       return 1
else:
   return action
```

Comme il n'y a que 2 actions, nous renvoyons 0 ou 1 avec une probabilité 0.5 dans le cas d'un choix aléatoire et l'action ayant la valeur de Q maximale dans le cas contraire.

Question 7

On rappelle que l'équation de Bellman est :

$$Q_\pi(s_t, a_t) = R(s_t, a_t) + \gamma \max_a Q_\pi(P(s_t, a_t), a)$$

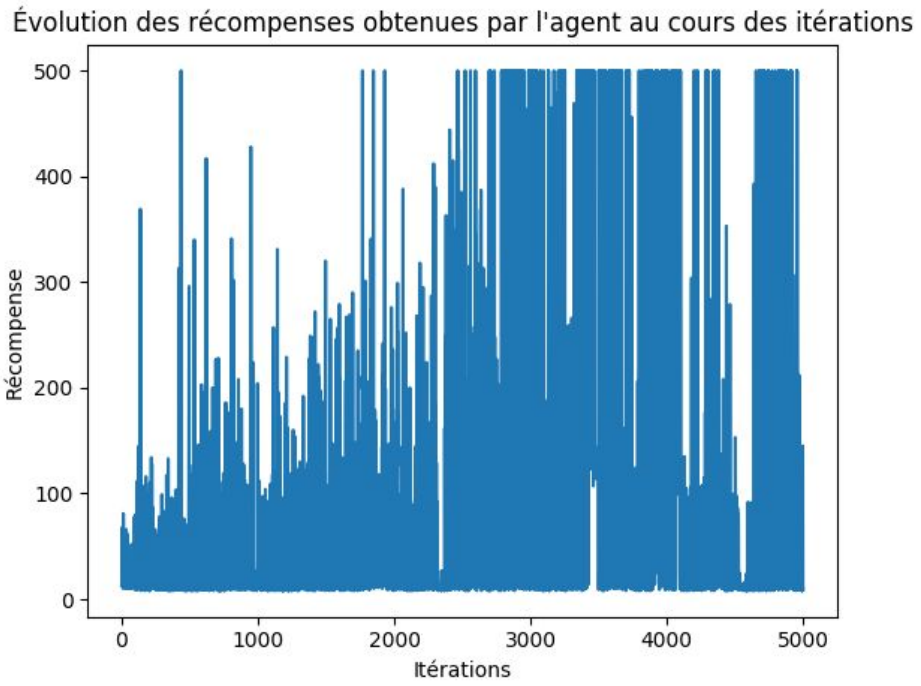
On définit la fonction d'apprentissage comme suit :

```
def learn(self, buffer):
          global target_neural_network
          if len(buffer) > 100:
               batch = sampling(buffer, 42)
               self.nb_learn += 1
               for ex in batch:
                    optim.zero_grad()
                    qvalue =
neural_network(torch.tensor(ex["state"], dtype=torch.float))
                    if ex["end_ep"]:
                         j = (qvalue[ex["action"]].item() - (
                                   ex["reward"])) ** 2
                    else:
                         j = (qvalue[ex["action"]].item() - (
                                   ex["reward"] + self.gamma *
neural_network(torch.tensor(ex["next_state"],
dtype=torch.float)).max().item())) ** 2
                    a = qvalue.clone()
                    a[ex["action"]] = j
                    loss = criterion(qvalue, a)
                    loss.backward()
                    optim.step()
```

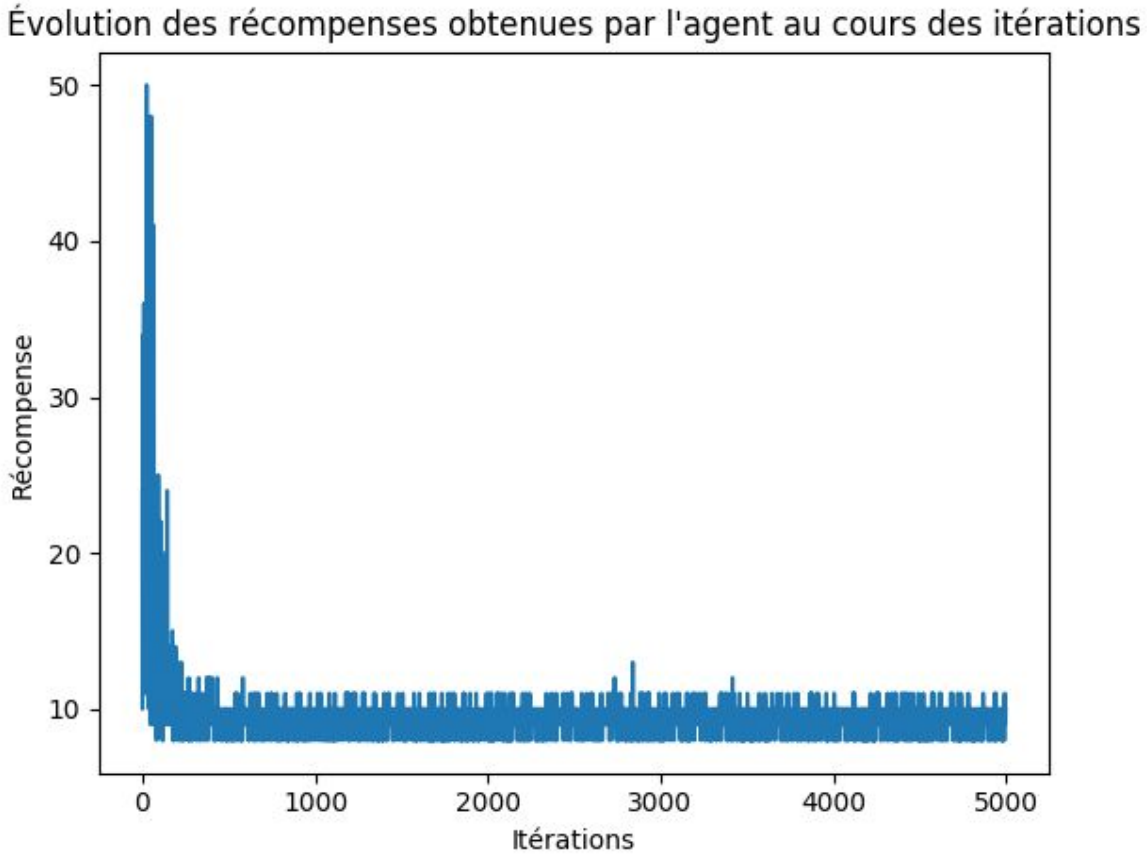
Question 8

```
target_neural_network = copy.deepcopy(neural_network)
# ...
class NeuralAgent(object):
      # ...
      def learn(self, buffer):
            global target_neural_network
            if len(buffer) > 100:
                  batch = sampling(buffer, 42)
                  self.nb_learn += 1
                  for ex in batch:
                        optim.zero_grad()
                        qvalue =
neural_network(torch.tensor(ex["state"], dtype=torch.float))
                        if ex["end_ep"]:
                              j = (qvalue[ex["action"]].item() - (
                                          ex["reward"])) ** 2
                        else:
                              j = (qvalue[ex["action"]].item() - (
                                          ex["reward"] + self.gamma *
target_neural_network(torch.tensor(ex["next_state"],
dtype=torch.float)).max().item())) ** 2
                        a = qvalue.clone()
                        a[ex["action"]] = j
                        loss = criterion(qvalue, a)
                        loss.backward()
                        optim.step()
                  if self.nb_learn > 10:
                        target_neural_network =
copy.deepcopy(neural_network)
                        self.nb_learn = 0
                  self.t = max(self.t * 0.99, 0.001)
```

Une fois cette partie totalement codée, nous avons essayé d'améliorer l'apprentissage de notre agent jouant au *cartpole* en modifiant les hyperparamètres. En utilisant un pas d'apprentissage de 0.01 et 5000 itérations d'apprentissage, nous avons obtenu des résultats corrects, même si l'agent n'arrive pas à obtenir de façon constante de bons résultats :



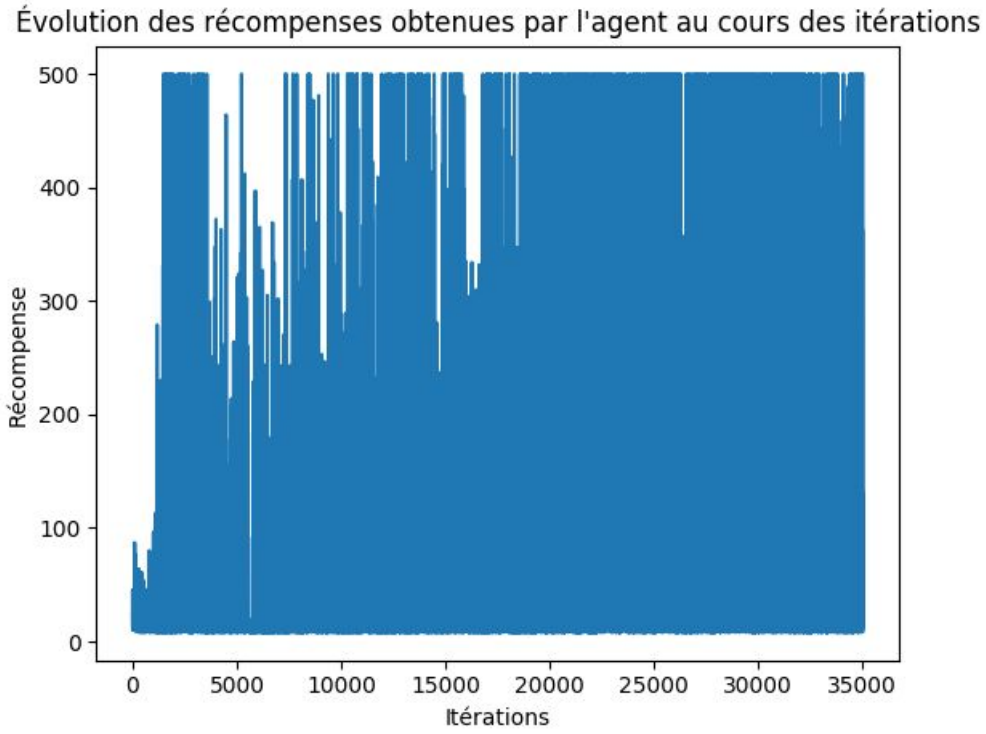
Par contre, en relançant avec les mêmes paramètres, nous avons obtenus de très mauvais résultat. L'apprentissage de l'agent ne semble donc pas être constant :

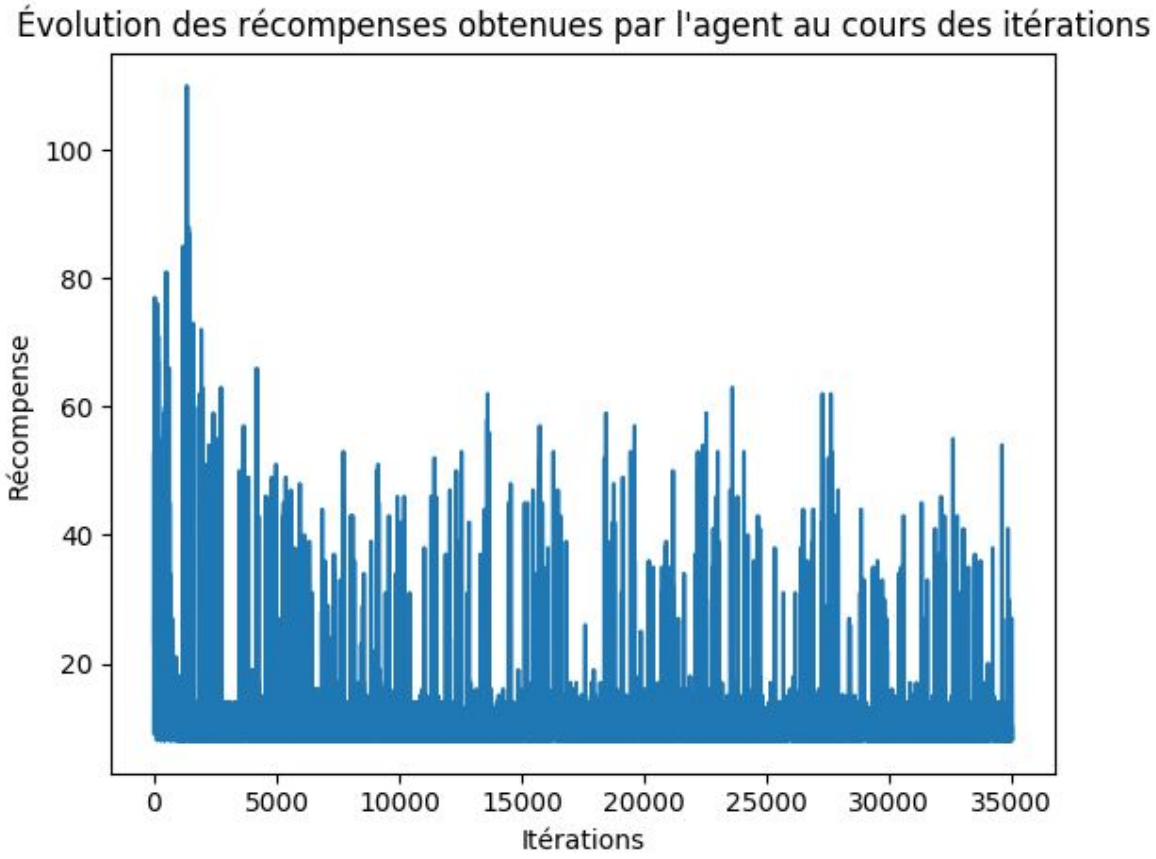


Dans cet exemple, il semble que l'agent ait appris à avoir le plus mauvais score possible au lieu du meilleur. Pour essayer d'améliorer cela, nous avons tenté différents paramètres, mais aucun d'entre eux ne nous a donné de vraiment bon résultat.

Pour tenter de limiter l'aléatoire dans l'apprentissage, nous avons tenté d'attendre que le buffer soit plus rempli (supérieur à 3000 éléments) et également d'augmenter le nombre d'itérations d'apprentissage, mais cela ne semble pas faire une réelle différence : Avec 35 000 itérations et un pas toujours de 0.01, les résultats ne semblent pas plus stables. Pour les autres paramètres, nous utilisons un alpha de 0.9, et pour la stratégie epsilon greedy, tant que l'agent n'a pas commencé à apprendre, il vaut 1 (la politique est donc totalement aléatoire), et il est ensuite multiplié par 0.99 à chaque pas d'apprentissage jusqu'à un minimum de 0.01. Le réseau est copié dans le réseau cible tous les 1000 pas d'apprentissage.

Nous arrivons à obtenir après 2 exécutions successives les graphes suivants :





On voit qu'on obtient toujours des résultats très variés en fonction des exécutions sans modifier les paramètres. Nous n'avons pas réussi à améliorer cela, et nous n'avons pas non plus réussi à comprendre d'où venait le problème. Nous avons également modifié chacun des autres paramètres, mais aucun n'a eu un réel impact.

Comme les résultats que nous obtenons dépendent totalement des exécutions, il est difficile de tester l'impact des différents paramètres. Pour la plupart d'entre eux, Nous n'avons pas observé de grande différence. En effet nous continuons d'avoir parfois de bons résultats, et parfois de mauvais. Il est donc difficile de déterminer dans quelle mesure l'hyperparamètre a impacté l'apprentissage.

Nous avons ensuite adapté notre agent pour qu'il puisse apprendre sur l'environnement Atari *Breakout*.

Partie 3

Question 1

Le preprocessing des images données par Atari est réalisé avec les lignes ci-dessous :

```
env = wrappers.AtariPreprocessing(env, screen_size=84,
```

```
frame_skip=4, grayscale_obs=True)
env = wrappers.FrameStack(env, 4)
```

Nous avons choisi d'utiliser les wrappers déjà existant proposé par `gym`, afin de ne pas avoir à les implémenter nous-même. Le premier permet de convertir l'écran en un carré de 84 par 84 pixels, alors que le second permet de regrouper 4 frames afin de pouvoir détecter des informations sur les déplacements comme la vitesse de la balle.

Question 2

Pour le reste de notre code, il est en grande partie compatible sans aucune modification avec le nouvel environnement. Pour le buffer d'état, nous utilisons toujours une `deque` python en enregistrant les expériences sous forme de dictionnaire. Le format d'un état n'a donc pas d'importance.

Question 3

Nous avons suivi les recommandations de l'article pour le choix du réseau de neurones que nous avons utilisé. Nous avons donc implémenter 3 réseaux convolutionnels avec `torch.conv2d`. Le premier utilise un filtre de convolution de 8 par 8 et en produit 32. Le second utilise des filtres de 4 par 4 et en produit 64 et le dernier utilise des filtres de 3 par 3 et en produit 64 également. Entre chaque couche de convolution on utilise un filtre ReLU. Le réseau se termine par une couche qui permet de ressortir la Qvaleur de chacune des actions.

```
class ConvNet(nn.Module):
    """
    Neural network that predicts the q-values for all actions for a
given state.
    """

    def __init__(self):
        super(ConvNet, self).__init__()
        self.conv1 = nn.Conv2d(4, 32, 8, stride=4)
        self.conv2 = nn.Conv2d(32, 64, 4, stride=2)
        self.conv3 = nn.Conv2d(64, 64, 3, stride=1)
        self.fc1 = nn.Linear(64 * 7 * 7, 4)

    def forward(self, x):
        x = F.relu(self.conv1(x))
```

```
        x = F.relu(self.conv2(x))
        x = F.relu(self.conv3(x))
        x = x.reshape(-1)
        x = self.fc1(x)
        return x
```

Question 4

Nous avons essayé différents paramètres, en particulier ceux présenté dans l'article, mais nous n'avons pas réussi à obtenir un résultat satisfaisant. Pour les paramètres proposés par l'article, nous n'avons pas pu utiliser vraiment les mêmes, car nous n'avons pas la puissance de calcul nécessaire. En effet avec une mémoire d'expérience de 1 000 000, le stockage des états nécessite bien plus de RAM que celle que nous avons à disposition. Comme nous avons une mémoire moins importante, nous avons commencé l'apprentissage beaucoup plus tôt. De plus nous n'avons pas non plus le temps pour effectuer autant d'itérations d'apprentissage que ce qui est réalisé dans l'article. En effet l'exécution est assez lente, nous avons donc effectué des tests avec beaucoup moins d'itérations d'apprentissage pour pouvoir observer les résultats. Pour la plupart des tests, nous avons lancé entre 200 et 500 parties de breakout. Afin de réaliser au moins 1 test avec plus d'itération, nous avons lancé un test avec 5000 parties, mais le résultat est totalement aléatoire avec un meilleur score de 12. Nous n'avons donc pas du tout réussi à faire apprendre notre agent sur le breakout.

Question 5

Comme nous nous n'avons pas réussi à obtenir de meilleurs résultat que l'aléatoire, il n'y a pas d'intérêt à enregistrer une vidéo de notre agent en train de jouer. Cependant, nous avons tout de même exporté la vidéo, qui est disponible à la racine du projet sous le nom "`atari-breakout.mp4`".