Projet Apprentissage par renforcement

Alioune CISSE

Master 2 Ingénierie des données et Intelligence Artificielle

Tél : +221 78 306 80 16

Email : alioune.cisse97@univ-thies.sn

Université Iba Der THIAM de Thiès

Smart Geese Trained by Reinforcement Learning

Introduction :

Dans le cadre de l'application du cours de "Reinforcement Learning" (Apprentissage par renforcement en français), il nous est demandé de réaliser un projet nous permettant de mettre en pratique nos connaissances et nos compétences. Ayant une passion pour les jeux de stratégie, j'ai utilisé à la base un code pris sur le site kaggle qui permet à des oies de trouver de la nourriture dans un environnement tout en évitant de se heurter à d'autres oies ou à une partie de leur corps. Ce dudit document se voit comme un rapport expliquant en détails, le travail que nous avons eu à réaliser.

Présentation du travail :

Le code a été un peu mélangé sur kaggle et nous avons pris le soin de bien le traiter d'abord. Nous l'avons décomposé en deux fichiers :

- Un fichier learning.py qui permet de faire les apprentissages
- Un fichier run.ipynb qui permet de d'executer le modèle

I.) Learning.py :

Le code a été développé avec le langage pytorch. PyTorch est une bibliothèque logicielle Python open source d'apprentissage machine qui s'appuie sur Torch développée par Facebook. PyTorch permet d'effectuer les calculs tensoriels nécessaires notamment pour l'apprentissage profond. Contrairement à son principal concurrent Tensorflow développé par Google, PyTorch a su se démarquer par la simplicité de son architecture simple et ces graphiques dynamiques. De plus, Pytorch avec son graphique de calcul est défini à l'exécution. Depuis sa création en 2016, la communauté de Pytorch ne cesse de s'aggrandir bien vrai que Tensorflow occupe toujours la première place.

Le code est constitué de deux classes TorusConv2d et Geesenet. La première permet de faire un modèle simple en pytorch en utilisant une couche de convolution 2D et une couche de normalisation. La seconde utilise la première et calcul la policy et la valeur. En RL une **politique (policy)** est définie comme une stratégie qu'un agent utilise dans la poursuite d'objectifs . La politique dicte les actions que l'agent entreprend en fonction de l'état de l'agent et de l'environnement. Quant à la fonction **valeur (value)**, elle représente la valeur pour que l'agent soit dans un certain état. Plus précisément, la fonction de valeur d'état décrit le retour attendu G_t d'un état donné. En général, une fonction de valeur d'état est définie concernant une politique spécifique, puisque le retour attendu dépend de la politique.

Hormis les deux classes, nous avons la function make_input qui permet de donner aux modèles les entrées pour faire l'apprentissage. Cette fonction permet de récupérer les positions et mouvements de chaque oie. Donc à chaque itération la position des oies est connue.

Enfin, nous avons la fonction agent qui permet de passer tous les arguments et prendre les bonnes décisions. En RL, l'agent est celui qui prend des décisions en fonction des récompenses et des punitions que cela produit.

NB : Il y'a également une longue chaine de caractère stocké dans une variable appelée PARAM que vous pourriez trouver au niveau du code, c'est un modèle pré entrainé que l'auteur du code a réutilisé. Vous pouvez l'ignorer et donc ne pas executer l'intruction model.load_state_dict(). Cependant pour un meilleur résultat, je l'ai réutilisé dans mon code. L'image suivante montre une partie du code

```python
class TorusConv2d(nn.Module):
    def __init__(self, input_dim, output_dim, kernel_size, bn):
        super().__init__()
        self.edge_size = (kernel_size[0] // 2, kernel_size[1] // 2)
        self.conv = nn.Conv2d(input_dim, output_dim, kernel_size=kernel_size)
        self.bn = nn.BatchNorm2d(output_dim) if bn else None

    def forward(self, x):
        h = torch.cat([x[:,:,:,-self.edge_size[1]:], x, x[:,:,:,:self.edge_size[1]]], dim=3)
        h = torch.cat([h[:,:,-self.edge_size[0]:], h, h[:,:,:self.edge_size[0]]], dim=2)
        h = self.conv(h)
        h = self.bn(h) if self.bn is not None else h
        return h


class GeeseNet(nn.Module):
    def __init__(self):
        super().__init__()
        layers, filters = 12, 32
        self.conv0 = TorusConv2d(17, filters, (3, 3), True)
        self.blocks = nn.ModuleList([TorusConv2d(filters, filters, (3, 3), True) for _ in range(layers)])
        self.head_p = nn.Linear(filters, 4, bias=False)
        self.head_v = nn.Linear(filters * 2, 1, bias=False)

    def forward(self, x):
        h = F.relu_(self.conv0(x))
        for block in self.blocks:
            h = F.relu_(h + block(h))
        h_head = (h * x[:,:1]).view(h.size(0), h.size(1), -1).sum(-1)
        h_avg = h.view(h.size(0), h.size(1), -1).mean(-1)
        p = self.head_p(h_head)
        v = torch.tanh(self.head_v(torch.cat([h_head, h_avg], 1)))

        return {'policy': p, 'value': v}


# Input for Neural Network

def make_input(obses):
    b = np.zeros((17, 7 * 11), dtype=np.float32)
    obs = obses[-1]

    for p, pos_list in enumerate(obs['geese']):
        # head position
        for pos in pos_list[:1]:
            b[0 + (p - obs['index']) % 4, pos] = 1
        # tip position
        for pos in pos_list[-1:]:
            b[4 + (p - obs['index']) % 4, pos] = 1
        # whole position
```

Quelques apports : Passer de Pytorch à Pytorch Lightning

PyTorch Lightning est une bibliothèque Python open source qui fournit une interface de haut niveau pour PyTorch , un framework d' apprentissage en profondeur populaire. Il s'agit d'un framework léger et performant qui organise le code PyTorch pour dissocier la recherche de l'ingénierie, facilitant ainsi la lecture et la reproduction des expériences d'apprentissage en profondeur. Il est conçu pour créer des modèles d'apprentissage en profondeur évolutifs qui peuvent facilement s'exécuter sur du matériel distribué tout en gardant les modèles indépendants du matériel. Pour pouvoir l'utiliser ou executer le code, veuillez l'installer avec *pip install pytorch_lightning* si ce n'est pas déjà fait. Pytorch lightning offre de nombreux avantages qui seraient longs à détailler dans ce document. Pour en savoir plus sur ces avantages et fonctionnalités vous pouvez cliquer ici. Dans le document zippé mis en pièces jointes, vous trouverez à la fois le code écrit en pytorch simple que j'ai nommé learning_source.py et celui avec pytorch lightning nommé learning.py.

```python
class TorusConv2d(pl.LightningModule):
    def __init__(self, input_dim, output_dim, kernel_size, bn):
        super().__init__()
        self.edge_size = (kernel_size[0] // 2, kernel_size[1] // 2)
        self.conv = nn.Conv2d(input_dim, output_dim, kernel_size=kernel_size)
        self.bn = nn.BatchNorm2d(output_dim) if bn else None

    def forward(self, x):
        h = torch.cat([x[:,:,:,-self.edge_size[1]:], x, x[:,:,:,:self.edge_size[1]]], dim=3)
        h = torch.cat([h[:,:,-self.edge_size[0]:], h, h[:,:,:self.edge_size[0]]], dim=2)
        h = self.conv(h)
        h = self.bn(h) if self.bn is not None else h
        return h


class GeeseNet(pl.LightningModule):
    def __init__(self):
        super().__init__()
        layers, filters = 12, 32
        self.conv0 = TorusConv2d(17, filters, (3, 3), True)
        self.blocks = nn.ModuleList([TorusConv2d(filters, filters, (3, 3), True) for _ in range(layers)])
        self.head_p = nn.Linear(filters, 4, bias=False)
        self.head_v = nn.Linear(filters * 2, 1, bias=False)

    def forward(self, x):
        h = F.relu_(self.conv0(x))
        for block in self.blocks:
            h = F.relu_(h + block(h))
        h_head = (h * x[:,:1]).view(h.size(0), h.size(1), -1).sum(-1)
        h_avg = h.view(h.size(0), h.size(1), -1).mean(-1)
        p = self.head_p(h_head)
        v = torch.tanh(self.head_v(torch.cat([h_head, h_avg], 1)))

        return {'policy': p, 'value': v}

    def cross_entropy_loss(self, logits, labels):
        return F.nll_loss(logits, labels)

    def training_step(self, train_batch, batch_idx):
        x, y = train_batch
        logits = self.forward(x)
        loss = self.cross_entropy_loss(logits, y)
        self.log('train_loss', loss)
        return loss

    def validation_step(self, val_batch, batch_idx):
        x, y = val_batch
        logits = self.forward(x)
        loss = self.cross_entropy_loss(logits, y)
        self.log('val_loss', loss)

    def configure_optimizers(self):
```

II.) Run.ipynb

J'ai mis ce fichier en ipynb car nous aurons besoin de IPython pour visualiser notre environnement. Et en parlant d'environnement, celui que nous utilisons est *kaggle-environments*

Kaggle-environments a été créé pour évaluer les épisodes. Alors que d'autres bibliothèques ont établi des précédents d'interface (comme Open.ai Gym), l'accent de cette bibliothèque se concentre sur :

- Évaluation des épisodes (par rapport aux agents de formation).
- Environnements/cycles de vie des agents configurables.
- Création simplifiée d'agents et d'environnements.
- Syntaxe/interfaces compatibles/transpilables en plusieurs langues.

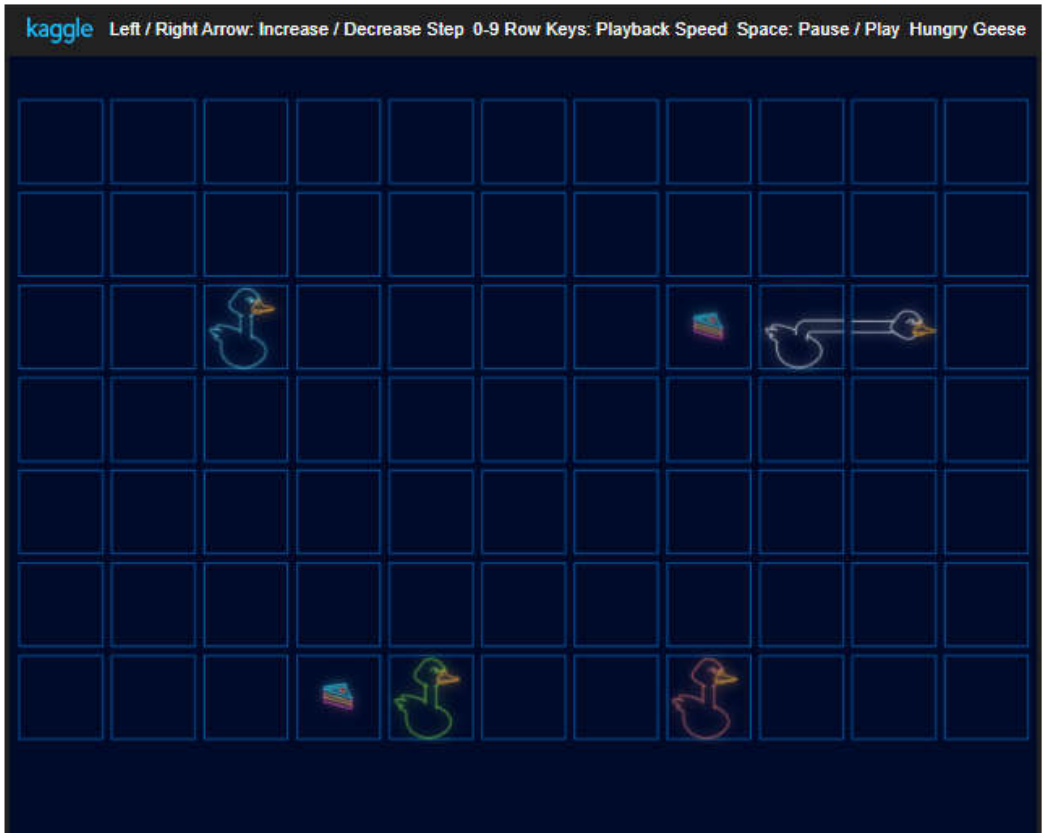
Sa méthode d'utilisation est semblable à celle de gym et offre (selon moi) un meilleur interface graphique d'où ma préférence à cet environnement.

Si nous revenons au code de notre fichier run.ipynb, sa structure est simple et très facile à comprendre. D'abord nous appelons l'environnement "hungry_geese" que nous utilisons puis nous ordonnons à cet environnement d'executer notre fichier learning.py et enfin nous lui suggérons d'afficher le travail en mode ipynb d'où l'importance de ne pas mettre ce fichier en format .py sinon, l'affichage peut ne pas être au rendez-vous. L'image suivante montre le code.

Entrée [38]:
```
from kaggle_environments import make
env = make("hungry_geese", debug=True)

env.reset()
env.run(['learning.py', 'learning.py', 'learning.py', 'learning.py'])
env.render(mode="ipython", width=800, height=700)
```

```
Opposite action: (3, <Action.WEST: 4>, <Action.EAST: 2>)
Goose Collision: NORTH
```



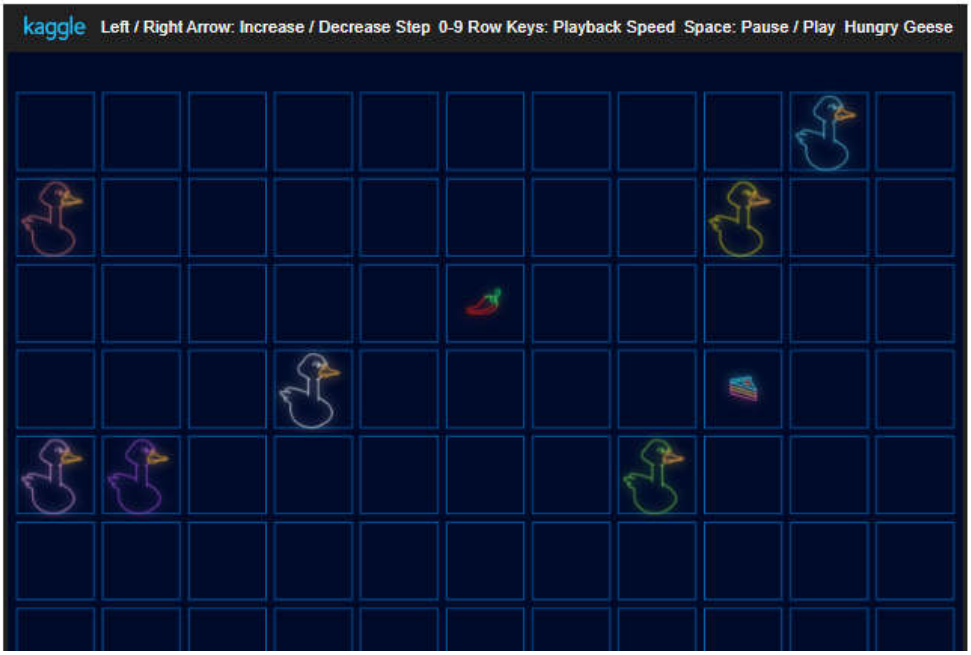
NB : Si le fichier learning.py se trouve dans un endroit différent de celui où se trouve le fichier run.ipynb merci d'indiquer le chemin complet avant d'executer.

Quelques améliorations : Vous avez certainement remarqué que pour faire jouer quatre oies, il m'a fallu appelé le fichier learning.py quatre fois ce qui n'est pas trop "pythonique" ou pas très ingénieux. J'ai donc modifié cette partie en créant une nouvelle variable appelé n_geese qui détermine le nombre d'oies que nous voulons tester. (Cette variable peut prendre des valeurs de 1 à 8). L'image suivante montre l'exemple avec 7 oies.

Entrée [41]:
```
from kaggle_environments import make
env = make("hungry_geese", debug=True)

n_geese = 7
geeses = ['learning.py' for i in range(n_geese)]
env.reset()
env.run(geeses)
env.render(mode="ipython", width=800, height=700)
```

```
Goose Collision: EAST
Goose Collision: WEST
Goose Collision: EAST
Goose Collision: WEST
Goose Collision: WEST
Goose Collision: SOUTH
```



Donc maintenant, pour avoir 4 oies il suffit juste remplacer 7 par 4 et ce sera terminé.

Conclusion :

Ce projet, mon tout premier en RL a été d'une importance capitale pour moi. Il m'a permis de découvrir de nouvelles choses comme l'environnement pygame (que j'utilise pour d'autres fins en ce moment), kaggle-environments, mais aussi d'approfondir des connaisances dans certains packages comme Pytorch et Pytorch Lightning que j'utilisais souvent pour faire du Deep Learning et que pour la première, je les ai utilisé dans le cadre du RL. Il m'a permis également de renforcer mes connaissances théoriques dans le RL notamment grâce aux nombreux tutoriels que j'ai regardé sur des sites comme medium, towards data science, github, ... et des videos sur youtube mais aussi et un excellent ouvrage que j'ai lu sur scholarvox.