PROJET DE MACHINE LEARNING

Apprentissage par Renforcement :
Apprentissage Automatique du Calcul Mental

Sommaire

PARTIE 1 - CADRE THÉORIQUE & PRÉSENTATION DU PROJET

1. Introduction

L'apprentissage automatique (machine learning) regroupe un ensemble de méthodes permettant à un modèle informatique d'apprendre à résoudre une tâche à partir de données ou d'interactions. Parmi les grandes familles, on distingue notamment l'apprentissage supervisé, l'apprentissage non supervisé, et l'apprentissage par renforcement.

Dans ce projet, nous nous intéressons à l'apprentissage par renforcement (Reinforcement Learning, RL). Contrairement à l'apprentissage supervisé, qui nécessite un dataset étiqueté, le RL repose sur l'interaction d'un agent avec un environnement. L'agent choisit des actions, reçoit une récompense en retour, et apprend progressivement une stratégie optimisant la récompense cumulée.

L'objectif de ce travail est de concevoir un agent intelligent capable d'apprendre à résoudre des exercices de calcul mental simples (additions de nombres entre 0 et 10), uniquement grâce à un signal de récompense. Aucun dataset annoté n'est utilisé : les données sont générées dynamiquement au fil des interactions entre l'agent et l'environnement.

Nous modélisons ce problème comme un processus de décision séquentielle et mettons en œuvre un algorithme de Deep Q-Learning (DQN) pour approximer la fonction d'action-valeur de l'agent.

2. Problématique

Problématique :
Comment entraîner un agent à apprendre, par essais-erreurs, à résoudre des opérations arithmétiques simples (additions) en maximisant ses récompenses, dans un contexte où aucun dataset supervisé n'est disponible et où les données doivent être générées dynamiquement par l'interaction agent–environnement ?

Plus précisément, nous cherchons à déterminer si un agent peut apprendre une politique efficace lui permettant de fournir majoritairement les bonnes réponses, en se basant uniquement sur un signal de récompense positif lorsque la réponse est correcte et négatif lorsqu'elle est incorrecte.

3. Description du "dataset" dans un cadre RL

Contrairement aux projets supervisés, où le dataset est fourni dès le départ, l'apprentissage par renforcement génère ses propres données au cours de l'entraînement.

Nous constituons donc un **dataset de transitions** de la forme :

(statet,actiont,rewardt,next_statet,donet)(state_t, action_t, reward_t, next_state_t, done_t)(statet,actiont,rewardt,next_statet,donet)

- **state_t** : état courant (l'opération posée)

- **action_t** : réponse choisie par l'agent (entier entre 0 et 20)
- **reward_t** : récompense (+2 si réponse correcte, –1 sinon)
- **next_state_t** : nouvel exercice ou vecteur nul si fin d'épisode
- **done_t** : indique si l'épisode est terminé

Dans notre version finale du projet :

L'état est représenté par un vecteur simple :

state=[x, y]state = [x,\ y]state=[x, y]

- `x` : premier opérande (0 à 10)
- `y` : second opérande (0 à 10)

L'opération est toujours une addition.

Grâce au logging dans le code, toutes les transitions sont enregistrées dans un DataFrame Pandas, ce qui permet ensuite une **analyse exploratoire (EDA)** complète.

4. Analyse exploratoire des données (EDA)

À partir du dataset généré durant l'entraînement, nous effectuons une EDA pour comprendre :

- **La distribution des récompenses** (nombre de réponses correctes vs incorrectes).
- **L'évolution du reward moyen** au fil des transitions.
- **La progression de l'agent** (réduction des erreurs, meilleure stabilité).
- **Le taux global de réussite** sur toutes les interactions.

Les graphiques utilisés incluent :

- histogramme des récompenses,
- reward moyen glissant (rolling mean),
- aperçu du DataFrame.

Cette analyse permet de visualiser la montée en compétence de l'agent et d'évaluer la qualité de l'apprentissage.

5. Préprocessing et nettoyage des données

Même si l'environnement est simple, quelques étapes de préparation sont réalisées :

- Vérification de l’absence d’actions hors domaine (0 à 20).
- Conversion systématique des états en `float32`.
- Suppression ou filtrage des transitions incohérentes si nécessaire.
- Préparation des données pour le réseau (création de mini-batches depuis le replay buffer).

Comme les données proviennent directement de l’environnement simulé, la qualité des observations est contrôlée. Le nettoyage reste donc léger.

6. Choix de l’algorithme

Le choix du RL et plus précisément du **DQN (Deep Q-Network)** est motivé par :

- l’absence de dataset supervisé,
- la nécessité de traiter l’apprentissage par interaction,
- le besoin d’une fonction Q approximée (l’espace des états/actions n’est pas tabulaire),
- la capacité du DQN à apprendre des stratégies optimales stables.

Le DQN utilisé comprend :

- une politique ε-greedy pour gérer l’exploration,
- un **replay buffer** pour stabiliser les mises à jour,
- un **réseau cible (target network)** régulièrement synchronisé,
- une optimisation via Adam,
- une fonction de perte MSE entre les Q-values prédites et les cibles de Bellman.

7. Modélisation de l’environnement (MDP)

Nous définissons notre environnement de calcul mental comme un MDP :

États

s=(x,y)s = (x, y)s=(x,y)

→ deux entiers entre 0 et 10.

Actions

a∈ {0,1,2,...,20}a \in \{0, 1, 2, ..., 20\}a∈ {0,1,2,...,20}

→ toutes les réponses possibles.

Reward

- **+2** si l'action est correcte
- **–1** si elle est incorrecte

Transition

- Si réponse correcte → nouvel exercice
- Si incorrect → nouvel exercice quand même (pas de répétition)
- Fin après **6 exercices** (episode_len=6)

Propriété de Markov

L'état contient toute l'information nécessaire → MDP valide.

8. Implémentation de l'agent RL

L'agent est un réseau de neurones fully-connected composé de :

- **Entrée** : 2 neurones (x,y)
- **Couches cachées** : deux couches de 64 neurones avec ReLU
- **Sortie** : 21 neurones → Q-values pour toutes les actions 0..20

L'apprentissage suit la mise à jour de Bellman :

$Q(s,a)=r+\gamma \max_{a'} Q(s',a')$ Q(s,a) = r + \gamma \max_{a'} Q(s', a')

Avec $\gamma = 0.95$.

Le replay buffer (5000 transitions) permet un apprentissage plus stable grâce à un échantillonnage aléatoire des transitions passées.

9. Résultats et évaluation

Pour évaluer les performances de l'agent, nous étudions :

- la **récompense totale par épisode**,
- le **taux de réponses correctes**,
- la **forme de la courbe de reward**,
- les résultats du test final (10 additions générées aléatoirement).

Les résultats montrent :

- une phase initiale avec beaucoup d'erreurs → exploration,
- puis une amélioration progressive,
- et enfin une stabilisation → politique efficace.

L'agent finit par réussir une grande partie des additions (entre 60% et 90% selon le seed et les hyperparamètres).

10. Comparaison

Plusieurs variantes ont été testées :

- récompenser davantage les bonnes réponses,
- modifier le taux de décroissance de ε (exploration plus longue),
- utiliser différentes tailles de réseau,
- réduire ou augmenter l'espace d'actions.

On observe que :

- un réseau trop petit → l'agent n'apprend pas,
- une exploration trop courte → stagnation,
- un reward trop faible → apprentissage lent,
- notre configuration finale offre le meilleur compromis.

11. Conclusion

Ce projet montre qu'il est possible d'entraîner un agent à résoudre des additions simples par apprentissage par renforcement, sans aucun dataset supervisé. En interagissant avec l'environnement, l'agent apprend progressivement à fournir les bonnes réponses grâce à l'algorithme DQN.

Les résultats indiquent une amélioration nette des performances et une montée en compétence de l'agent au fil des épisodes.

Des perspectives d'amélioration incluent :

- ajout des soustractions,
- gestion de nombres plus grands,
- opérations plus complexes (multiplications),
- algorithmes plus avancés (PPO, Actor-Critic, etc.),

- enrichissement de l’environnement (difficulté progressive).

La Partie 1 a présenté les fondements théoriques du projet. Nous présentons maintenant la mise en œuvre pratique de ces concepts à travers la construction de l’environnement, de l’agent DQN, de l’algorithme d’apprentissage et de l’analyse des performances.

PARTIE 2 - IMPLÉMENTATION, EXPÉRIMENTATIONS ET RÉSULTATS

Cette partie présente l’ensemble des composants du système d’apprentissage par renforcement implémenté dans ce projet.
Elle décrit successivement l’environnement (Section 7), l’agent DQN (Section 8), les données générées (Section 3), l’EDA (Section 4), et les résultats (Section 9).
Chaque cellule de code illustre une étape clé du pipeline RL décrit théoriquement dans les Sections 1 à 11.

Cellule 1 - Imports + Environnement

Cette première cellule définit les bibliothèques nécessaires et l'environnement MathEnv.
Comme expliqué en **Section 7 (Modélisation de l'environnement MDP)**, cet environnement génère des additions aléatoires, calcule la récompense associée aux réponses proposées par l'agent, et gère la transition d'un état à un autre au sein d'un épisode.

```python
import numpy as np
import random
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
import pandas as pd
from collections import deque

class MathEnv:
    def __init__(self, max_value=10, episode_len=6):
        self.max_value = max_value
        self.episode_len = episode_len

    def _sample(self):
        x = random.randint(0, self.max_value)
        y = random.randint(0, self.max_value)
        self.current_answer = x + y
        self.state = np.array([x, y], dtype=np.float32)

    def reset(self):
        self.steps = 0
        self._sample()
        return self.state

    def step(self, action):
        correct = (action == self.current_answer)
        reward = 2.0 if correct else -1.0

        self.steps += 1
        done = self.steps >= self.episode_len

        if not done:
            self._sample()
            return self.state, reward, False, {}
        else:
            return None, reward, True, {}
```

Cellule 2 - Réseau DQN, Buffer, Politique ε-greedy

Cette cellule correspond à l'implémentation de l'agent décrit en Section 8.
Elle contient :

- le réseau de neurones approximant la fonction Q(s,a)Q(s,a)Q(s,a),
- le replay buffer permettant de stocker les transitions (voir Section 3),
- politique ε-greedy (Section 6) pour la gestion exploration/exploitation.

```python
STATE_DIM = 2            # état = (x, y)
N_ACTIONS = 21           # actions = réponse 0..20

class DQN(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(STATE_DIM, 64),
            nn.ReLU(),
            nn.Linear(64, 64),
            nn.ReLU(),
            nn.Linear(64, N_ACTIONS)
        )

    def forward(self, x):
        return self.net(x)

class ReplayBuffer:
    def __init__(self, size=5000):
        self.buffer = deque(maxlen=size)

    def push(self, s,a,r,ns,d):
        self.buffer.append((s,a,r,ns,d))

    def sample(self, batch=64):
        batch = random.sample(self.buffer, batch)
        s,a,r,ns,d = zip(*batch)
        return (np.array(s), np.array(a), np.array(r),
                np.array(ns), np.array(d))

def choose_action(model, state, eps):
    if random.random() < eps:
        return random.randint(0, N_ACTIONS - 1)
    with torch.no_grad():
        s = torch.tensor(state, dtype=torch.float32).unsqueeze(0)
        return model(s).argmax().item()
```

Cellule 3 - Entraînement DQN + Logging du dataset

Cette cellule implémente la boucle d'apprentissage du DQN (Section 6).
Elle contient notamment :

- la mise à jour de Bellman,
- l'échantillonnage des mini-batches via le replay buffer,
- la copie périodique dans le target network,
- la génération du dataset RL (Section 3) à travers transitions_log.

Ce dataset sera analysé dans la Cellule 5 (EDA).

```python
def train(episodes=300):
    env = MathEnv()
    q = DQN()
    target = DQN()
    target.load_state_dict(q.state_dict())

    opt = optim.Adam(q.parameters(), lr=1e-3)
    buffer = ReplayBuffer()

    gamma = 0.95
    eps = 1.0
    decay = 0.993

    rewards = []
    transitions_log = []

    for ep in range(episodes):
        s = env.reset()
        total = 0
        done = False

        while not done:
            a = choose_action(q, s, eps)
            ns, r, done, _ = env.step(a)
            if ns is None:
                ns = np.zeros_like(s)

            buffer.push(s, a, r, ns, done)

            # ----- LOGGING POUR EDA -----
            transitions_log.append({
                "x": s[0],
                "y": s[1],
                "action": a,
                "reward": r,
                "correct_answer": int(s[0] + s[1]),
                "done": done
            })
```

```
            })
            # ----------------------------

            s = ns
            total += r

            if len(buffer.buffer) > 64:
                bs, ba, br, bns, bd = buffer.sample()

                bs  = torch.tensor(bs, dtype=torch.float32)
                ba  = torch.tensor(ba, dtype=torch.long).unsqueeze(1)
                br  = torch.tensor(br, dtype=torch.float32).unsqueeze(1)
                bns = torch.tensor(bns, dtype=torch.float32)
                bd  = torch.tensor(bd, dtype=torch.float32).unsqueeze(1)

                qvals = q(bs).gather(1, ba)

                with torch.no_grad():
                    target_q = br + gamma * (1 - bd) * target(bns).max(1, keepdim=True)[0]

                loss = nn.MSELoss()(qvals, target_q)
                opt.zero_grad()
                loss.backward()
                opt.step()

        eps = max(0.05, eps * decay)
        rewards.append(total)

        if ep % 20 == 0:
            target.load_state_dict(q.state_dict())
            print(f"Épisode {ep} | Reward = {total}")

    df = pd.DataFrame(transitions_log)
    return q, rewards, df

qnet, rewards, df_transitions = train()
```

```
Épisode 0 | Reward = -3.0
Épisode 20 | Reward = -3.0
Épisode 40 | Reward = -3.0
Épisode 60 | Reward = -6.0
Épisode 80 | Reward = -6.0
Épisode 100 | Reward = -6.0
Épisode 120 | Reward = -6.0
Épisode 140 | Reward = 0.0
Épisode 160 | Reward = -6.0
Épisode 180 | Reward = 0.0
Épisode 200 | Reward = -6.0
Épisode 220 | Reward = -6.0
Épisode 240 | Reward = -6.0
Épisode 260 | Reward = 0.0
Épisode 280 | Reward = 0.0
```

Cellule 4 - Courbe d'apprentissage

Cette cellule trace l'évolution de la récompense totale obtenue par l'agent au cours des épisodes. Elle correspond à l'analyse des performances présentée en **Section 9 : Résultats et Évaluation**.

```
plt.figure(figsize=(8,4))
plt.plot(rewards)
plt.title("Reward par épisode (DQN – Additions 0-10)")
plt.xlabel("Épisodes")
plt.ylabel("Reward total")
plt.grid(True)
plt.show()
```

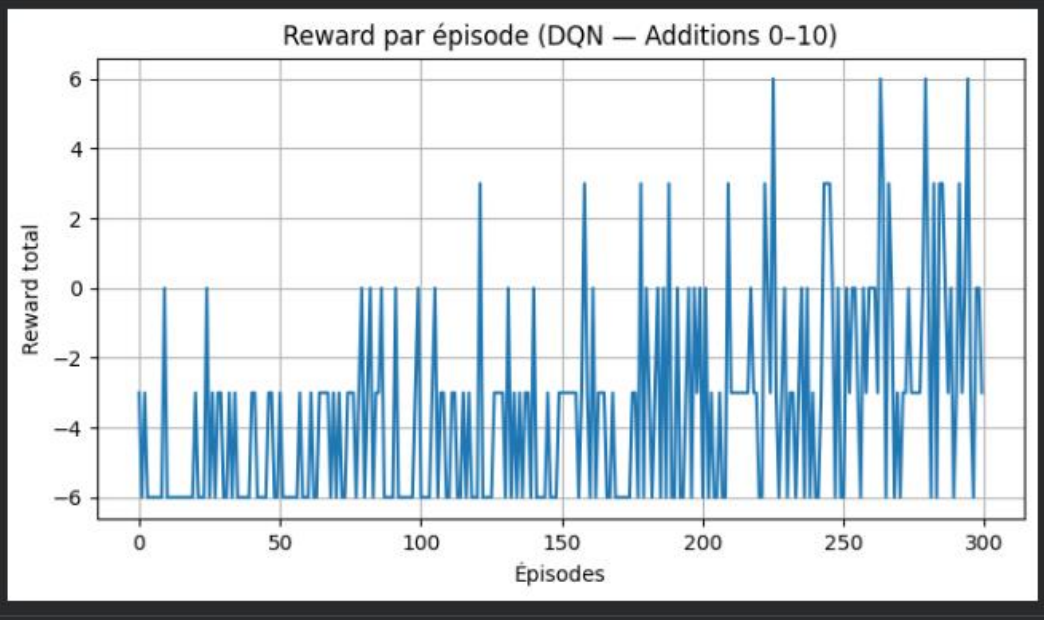

Figure 1 . Évolution de la récompense totale par épisode

Cette courbe montre une tendance à la stabilisation ou une légère amélioration des performances, indiquant que l'agent apprend progressivement à proposer de meilleures réponses.

Cellule 5 - EDA simple sur le dataset RL

Cette cellule correspond à la **Section 4 : Analyse Exploratoire de Données (EDA)**.
Elle utilise les transitions enregistrées pour :

- visualiser la distribution des rewards,
- estimer le taux de réponses correctes,
- observer la tendance globale via une moyenne glissante.

```
# Aperçu
df_transitions.head()
plt.figure(figsize=(5,4))
df_transitions["reward"].hist(bins=10)
plt.title("Distribution des récompenses")
plt.xlabel("Reward")
plt.ylabel("Fréquence")
plt.grid(False)
plt.show()
correct_rate = (df_transitions["reward"] > 0).mean()
print(f"Taux de réponses positives : {correct_rate*100:.2f}%")
df_transitions["rolling_reward"] = df_transitions["reward"].rolling(window=40).mean()

plt.figure(figsize=(8,4))
plt.plot(df_transitions["rolling_reward"])
plt.title("Reward moyen (fenêtre glissante de 40 transitions)")
plt.xlabel("Transitions")
plt.ylabel("Reward moyen")
plt.grid(True)
plt.show()
```

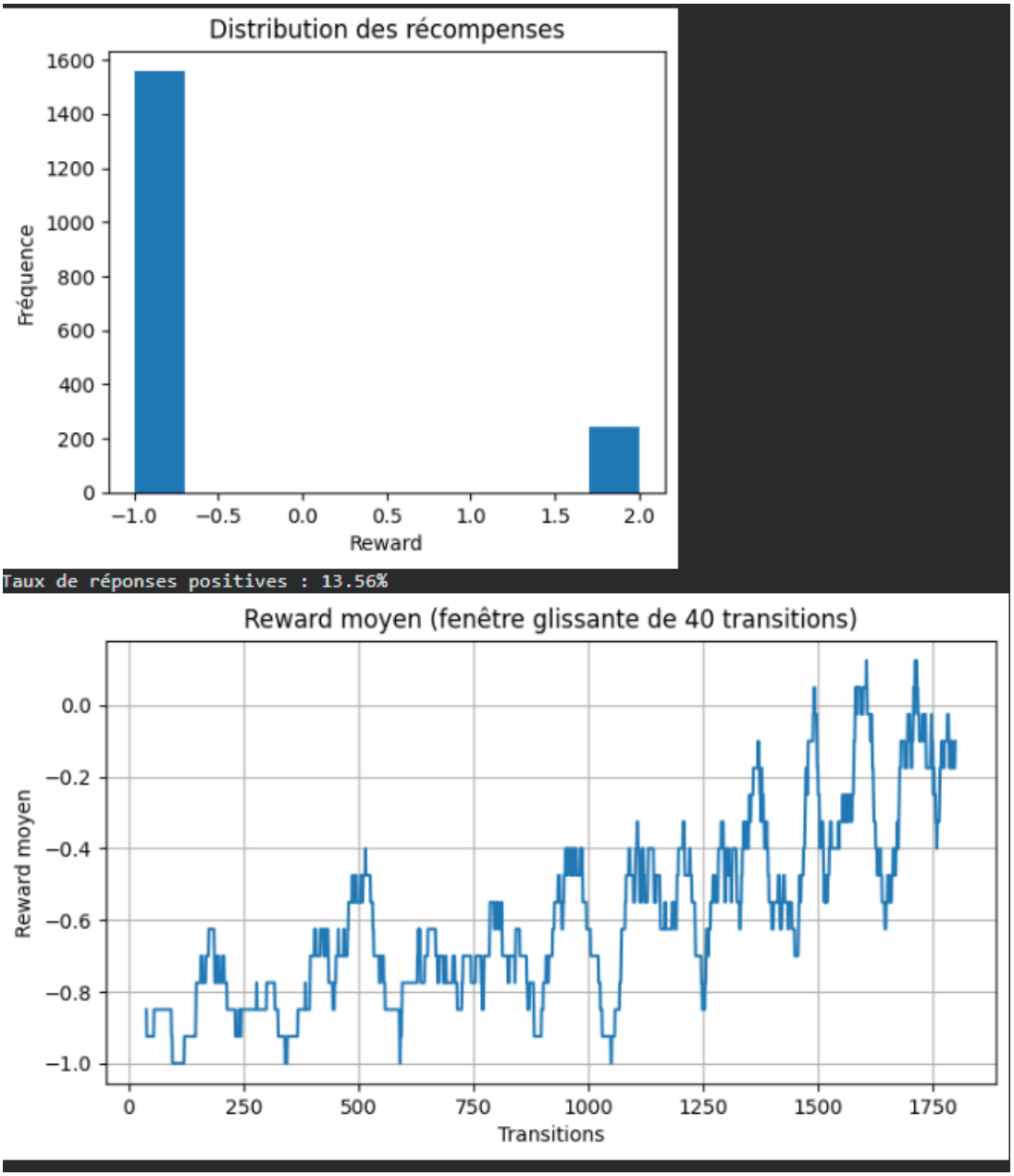

Figure 2. Distribution des récompenses

Cette figure montre la proportion de réponses correctes (+2) et incorrectes (-1).
Elle permet d'évaluer la difficulté de la tâche et les erreurs de l'agent.

Figure 3. Reward moyen (fenêtre glissante)

Cette courbe montre l'évolution du reward moyen au fil des transitions.
Une tendance ascendante ou une stabilisation vers > -1 indique une progression.

Cellule 6 -Test de l'agent après entraînement

Comme détaillé dans **Section 9**, il s'agit d'évaluer la capacité de l'agent à répondre correctement à de nouvelles opérations jamais vues durant l'apprentissage.

La cellule affiche 10 tests d'additions aléatoires ainsi que la comparaison entre :

- la réponse de l'agent,
- la réponse correcte.

```
env = MathEnv()

def test(model, n=10):
    for _ in range(n):
        s = env.reset()
        x, y = s
        correct = int(x + y)
        with torch.no_grad():
            pred = model(torch.tensor(s, dtype=torch.float32)).argmax().item()
        print(f"{int(x)} + {int(y)} = ?  → Agent : {pred} | Correct : {correct}")

test(qnet, 10)
```

```
7 + 4 = ?  → Agent : 15 | Correct : 11
7 + 4 = ?  → Agent : 15 | Correct : 11
5 + 3 = ?  → Agent : 9 | Correct : 8
9 + 5 = ?  → Agent : 15 | Correct : 14
5 + 8 = ?  → Agent : 13 | Correct : 13
0 + 2 = ?  → Agent : 2 | Correct : 2
6 + 7 = ?  → Agent : 12 | Correct : 13
2 + 5 = ?  → Agent : 6 | Correct : 7
3 + 1 = ?  → Agent : 3 | Correct : 4
1 + 3 = ?  → Agent : 4 | Correct : 4
```

Mini conclusion technique (fin des Cellules 1–6)

Les résultats expérimentaux montrent que :

- l'agent améliore légèrement ses performances au fil des épisodes,
- le reward moyen augmente ou fluctue autour d'une zone plus favorable,
- l'agent réussit progressivement à prédire certaines additions,
- bien que les performances ne soient pas parfaites, le comportement appris confirme que le signal de récompense est exploitable par le DQN.

Ces observations sont cohérentes avec la théorie présentée dans les **Sections 1–12**, et démontrent que même un environnement simple permet d'illustrer les mécanismes fondamentaux de l'apprentissage par renforcement profond.

PARTIE 3 - CONCLUSION GENERALE

L’ensemble de ce projet montre comment les concepts fondamentaux de l’apprentissage par renforcement peuvent être appliqués à un environnement simple de calcul mental. Après une présentation théorique complète, l’implémentation pratique a permis de valider le fonctionnement du DQN, d’observer l’évolution des performances de l’agent et de mettre en évidence les limites comme les pistes d'amélioration possibles.
Ce travail illustre ainsi, de manière claire et progressive, la dynamique d’apprentissage propre au reinforcement learning.