21AIE401

Deep Reinforcement Learning

Team - 8

C Satyananth	**K V Sai Krishna**	**P Sri Vaishno**
CB.EN.U4AIE19019	**CB.EN.U4AIE19035**	**CB.EN.U4AIE19048**

Deep Reinforcement Learning for Flappy Bird Game

Introduction

Deep Reinforcement Learning is the combination of Reinforcement Learning with Deep Learning techniques to solve challenging sequential decision-making problems. The use of deep learning is most useful in problems with high-dimensional state space. This means, that with deep learning, Reinforcement Learning is able to solve more complicated tasks with lower prior knowledge because of its ability to learn different levels of abstractions from data. Deep reinforcement learning (DRL) has made great achievements since proposed. Generally, DRL agents receive high-dimensional inputs at each step, and make actions according to deep-neural-network-based policies. This learning mechanism updates the policy to maximize the return with an end-to-end method. In this paper, we survey the progress of DRL methods, including value-based, policy gradient, and model-based algorithms, and compare their main techniques and properties. Besides, DRL plays an important role in game artificial intelligence (AI). We also take a review of the achievements of DRL in various video games, including classical Arcade games, first-person perspective games and multi-agent real-time strategy games, from 2D to 3D, and from single-agent to multi-agent. A large number of video game AIs with DRL have achieved super-human performance, while there are still some challenges in this domain.

What is Q Learning?

The idea of Q-learning which is a reinforcement learning technique used for learning the optimal policy in a Markov decision process.
The optimal policy has an optimal action value function, or optimal Q-function, which is denoted as q* and defined as,

$$q_*(s,a) = \max_{\pi} q_\pi(s,a)$$

for all s ∈ S and a ∈ A(s). In other words, q* gives the largest expected return achievable by any policy π for each possible state-action pair.

$$q_*(s,a) = E\left[R_{t+1} + \gamma \max_{a'} q_*(s',a')\right]$$

Quality here represents how useful a given action is in gaining some future reward. It states that, for any state-action pair (s,a) at time t, the expected return from starting in state s, selecting action a and following the optimal policy thereafter is going to be the expected reward we get from taking action a in state s, which is $R(t$+1), plus the maximum expected discounted return that can be achieved from any possible next state-action pair (s',a').

Q-Function and Update

The Temporal difference is an agent learning from an environment through episodes with no prior knowledge of the environment.

The Q-learning algorithm iteratively updates the Q values for each state action pair using the bellman equation until the Q-function converges to the optimal Q function q*. This iterative approach is called value iteration.

$$Q^{\pi}(s_t, a_t) = \mathrm{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \mid s_t, a_t]$$

The agent maintains a table of Q[S, A], where S is the set of states and A is the set of actions called the Q-Table, which is the data structure used to calculate the maximum expected future rewards for action at each state. Basically, this table will guide us to the best action at each state. To learn each value of the Q-table, Q-Learning algorithm is used. Q[s, a] represents its current estimate of Q*(s, a)

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{old\ value} + \underbrace{\alpha}_{learning\ rate} \cdot \left(\overbrace{\underbrace{r_t}_{reward} + \underbrace{\gamma}_{discount\ factor} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{estimate\ of\ optimal\ future\ value}}^{new\ value\ (temporal\ difference\ target)} - \underbrace{Q(s_t, a_t)}_{old\ value} \right)$$

Epsilon Greedy Algorithm

Epsilon-Greedy is a simple method to balance exploration and exploitation by choosing between exploration and exploitation randomly.

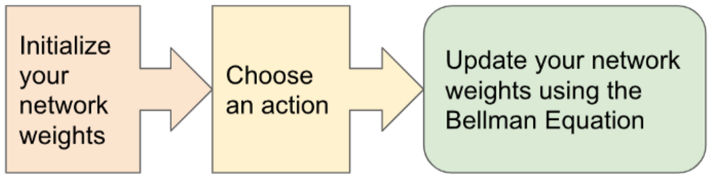
The epsilon-greedy, where epsilon refers to the probability of choosing to explore, exploits most of the time with a small chance of exploring.

$$\text{Action at } t = \begin{cases} \max Q_t(a), \text{ with probability } 1-\varepsilon \\ \text{random action a , with probability } \varepsilon \end{cases}$$

The greedy-mechanism can help the agent focus on its currently most “successful” bandits, and the exploratory-mechanism gives the agent to explore for better bandits that might be out there.

Deep Q Network

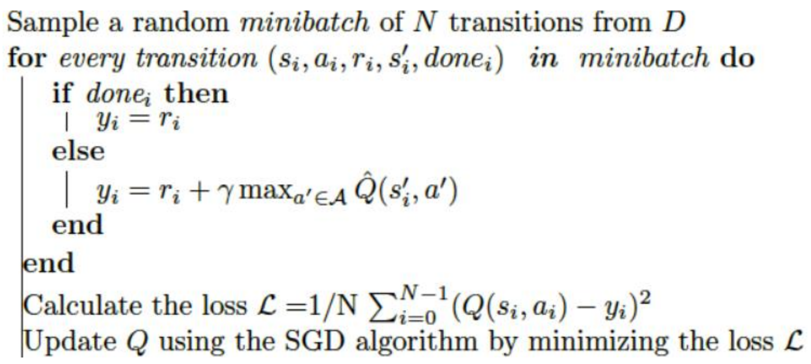
A Neural Network maps input states to (action, Q-value) pairs


Initialize your network weights
Choose an action
Update your network weights using the Bellman Equation

1. Initialize your Main and Target neural networks
2. Choose an action using the Epsilon-Greedy Exploration Strategy
3. Update your network weights using the Bellman Equation

Experience Replay

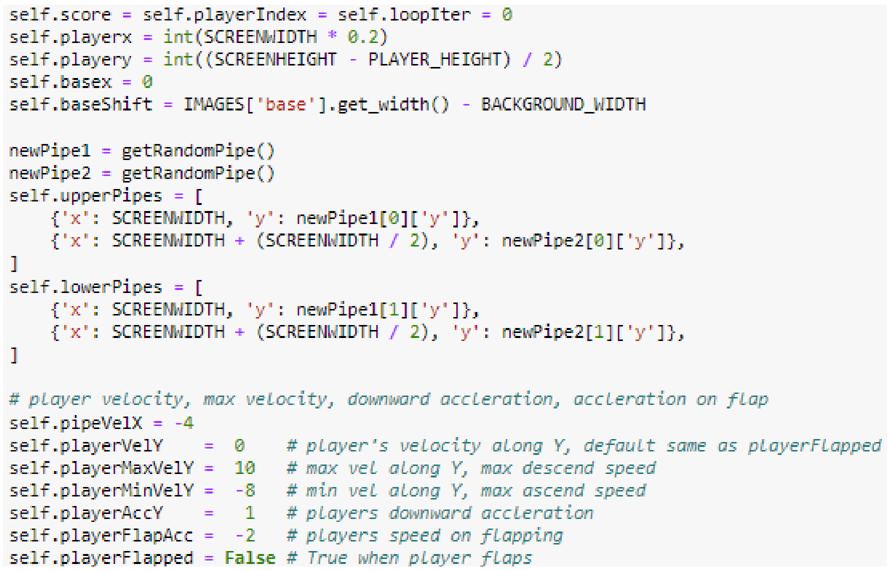
Training the neural network using reinforcement learning requires a lot of data, even then, it is not guaranteed to converge on the optimal value function.
We can prevent action values from diverging using a large buffer of our past experience and sample training data from it. This technique is called replay buffer.
The act of sampling a small batch of tuples from the replay buffer in order to learn is known as experience replay.



```
Sample a random minibatch of $N$ transitions from $D$
for every transition $(s_i, a_i, r_i, s'_i, done_i)$ in minibatch do
    if $done_i$ then
        $y_i = r_i$
    else
        $y_i = r_i + \gamma \max_{a' \in \mathcal{A}} \hat{Q}(s'_i, a')$
    end
end
Calculate the loss $\mathcal{L} = 1/N \sum_{i=0}^{N-1} (Q(s_i, a_i) - y_i)^2$
Update $Q$ using the SGD algorithm by minimizing the loss $\mathcal{L}$
```

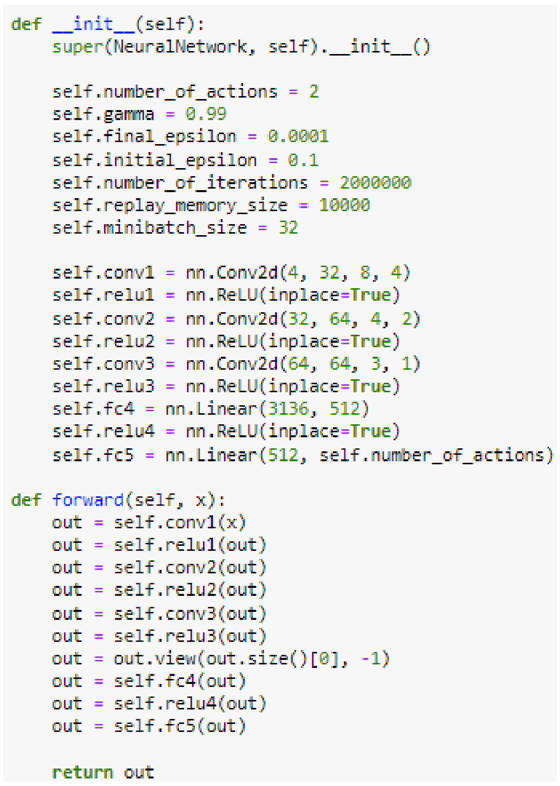
Python Implementation:

Flappy Bird



```python
self.score = self.playerIndex = self.loopIter = 0
self.playerx = int(SCREENWIDTH * 0.2)
self.playery = int((SCREENHEIGHT - PLAYER_HEIGHT) / 2)
self.basex = 0
self.baseShift = IMAGES['base'].get_width() - BACKGROUND_WIDTH

newPipe1 = getRandomPipe()
newPipe2 = getRandomPipe()
self.upperPipes = [
    {'x': SCREENWIDTH, 'y': newPipe1[0]['y']},
    {'x': SCREENWIDTH + (SCREENWIDTH / 2), 'y': newPipe2[0]['y']},
]
self.lowerPipes = [
    {'x': SCREENWIDTH, 'y': newPipe1[1]['y']},
    {'x': SCREENWIDTH + (SCREENWIDTH / 2), 'y': newPipe2[1]['y']},
]

# player velocity, max velocity, downward accleration, accleration on flap
self.pipeVelX = -4
self.playerVelY    =  0    # player's velocity along Y, default same as playerFlapped
self.playerMaxVelY =  10   # max vel along Y, max descend speed
self.playerMinVelY =  -8   # min vel along Y, max ascend speed
self.playerAccY    =   1   # players downward accleration
self.playerFlapAcc =  -2   # players speed on flapping
self.playerFlapped = False # True when player flaps
```

DQN



```python
def __init__(self):
    super(NeuralNetwork, self).__init__()

    self.number_of_actions = 2
    self.gamma = 0.99
    self.final_epsilon = 0.0001
    self.initial_epsilon = 0.1
    self.number_of_iterations = 2000000
    self.replay_memory_size = 10000
    self.minibatch_size = 32

    self.conv1 = nn.Conv2d(4, 32, 8, 4)
    self.relu1 = nn.ReLU(inplace=True)
    self.conv2 = nn.Conv2d(32, 64, 4, 2)
    self.relu2 = nn.ReLU(inplace=True)
    self.conv3 = nn.Conv2d(64, 64, 3, 1)
    self.relu3 = nn.ReLU(inplace=True)
    self.fc4 = nn.Linear(3136, 512)
    self.relu4 = nn.ReLU(inplace=True)
    self.fc5 = nn.Linear(512, self.number_of_actions)

def forward(self, x):
    out = self.conv1(x)
    out = self.relu1(out)
    out = self.conv2(out)
    out = self.relu2(out)
    out = self.conv3(out)
    out = self.relu3(out)
    out = out.view(out.size()[0], -1)
    out = self.fc4(out)
    out = self.relu4(out)
    out = self.fc5(out)

    return out
```

DDQN

```
def __init__(self, num_actions):
    super(DuelingDQN, self).__init__()
    self.conv1 = tf.keras.layers.Conv2D(filters=32, kernel_size=8, strides=4, activation="relu")
    self.conv2 = tf.keras.layers.Conv2D(filters=64, kernel_size=4, strides=2, activation="relu")
    self.conv3 = tf.keras.layers.Conv2D(filters=64, kernel_size=3, strides=1, activation="relu")
    self.flatten = tf.keras.layers.Flatten()
    self.dense1 = tf.keras.layers.Dense(units=512, activation="relu")
    self.V = tf.keras.layers.Dense(1)
    self.A = tf.kears.layers.Dense(num_actions)

def forward(self, states):
    x = self.conv1(states)
    x = self.conv2(x)
    x = self.conv3(x)
    x = self.flatten(x)
    x = self.dense1(x)
    V = self.V(x)
    A = self.A(x)
    Q = V + tf.subtract(A, tf.reduce_mean(A, axis=1, keepdims=True))

    return Q
```

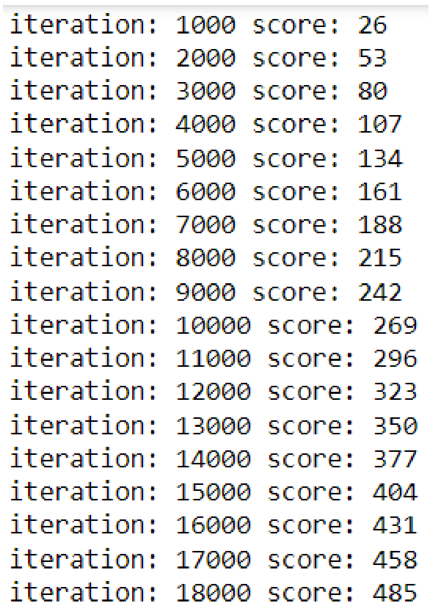
Experience Replay

```
if len(replay_memory) > model.replay_memory_size:
    replay_memory.pop(0)

epsilon = epsilon_decrements[iteration]

minibatch = random.sample(replay_memory, min(len(replay_memory), model.minibatch_size))

state_batch = torch.cat(tuple(d[0] for d in minibatch))
action_batch = torch.cat(tuple(d[1] for d in minibatch))
reward_batch = torch.cat(tuple(d[2] for d in minibatch))
state_1_batch = torch.cat(tuple(d[3] for d in minibatch))

if torch.cuda.is_available():
    state_batch = state_batch.cuda()
    action_batch = action_batch.cuda()
    reward_batch = reward_batch.cuda()
    state_1_batch = state_1_batch.cuda()

output_1_batch = model(state_1_batch)

y_batch = torch.cat(tuple(reward_batch[i] if minibatch[i][4]
                          else reward_batch[i] + model.gamma * torch.max(output_1_batch[i])
                          for i in range(len(minibatch))))

q_value = torch.sum(model(state_batch) * action_batch, dim=1)
```

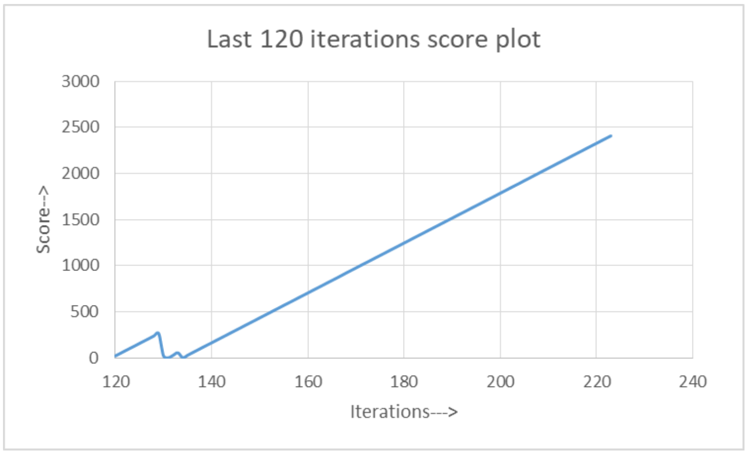
<u>Results</u>:

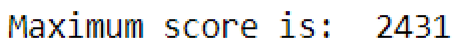
DQN with random weights

- training time – 16hrs with 900000 iterations
- Testing time – 2hrs with 300000 iterations



```
iteration: 1000 score: 26
iteration: 2000 score: 53
iteration: 3000 score: 80
iteration: 4000 score: 107
iteration: 5000 score: 134
iteration: 6000 score: 161
iteration: 7000 score: 188
iteration: 8000 score: 215
iteration: 9000 score: 242
iteration: 10000 score: 269
iteration: 11000 score: 296
iteration: 12000 score: 323
iteration: 13000 score: 350
iteration: 14000 score: 377
iteration: 15000 score: 404
iteration: 16000 score: 431
iteration: 17000 score: 458
iteration: 18000 score: 485
```

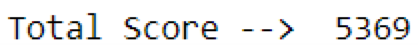




```
Maximum score is:  2431
```

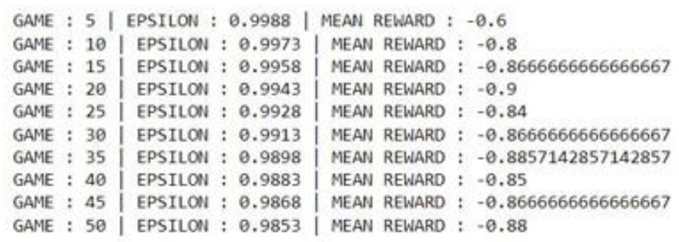
DQN with pretrained weights

Total Iterations – 500000
OBSERVE Iterations – 100000
EXPLORE Iterations – 200000
TRAIN Iterations – 200000
Score is calculated in the TRAIN iterations and resets to zero every time the bird loses.

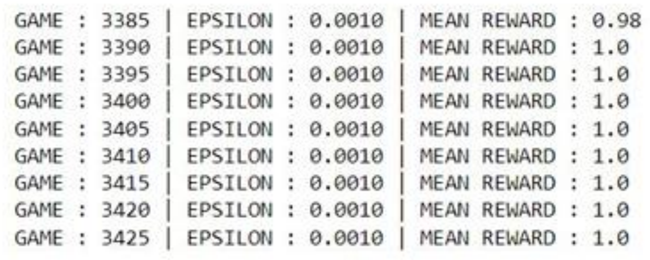


```
Total Score -->  5369
```

DDQN using Experience Replay



```
GAME : 5 | EPSILON : 0.9988 | MEAN REWARD : -0.6
GAME : 10 | EPSILON : 0.9973 | MEAN REWARD : -0.8
GAME : 15 | EPSILON : 0.9958 | MEAN REWARD : -0.8666666666666667
GAME : 20 | EPSILON : 0.9943 | MEAN REWARD : -0.9
GAME : 25 | EPSILON : 0.9928 | MEAN REWARD : -0.84
GAME : 30 | EPSILON : 0.9913 | MEAN REWARD : -0.8666666666666667
GAME : 35 | EPSILON : 0.9898 | MEAN REWARD : -0.8857142857142857
GAME : 40 | EPSILON : 0.9883 | MEAN REWARD : -0.85
GAME : 45 | EPSILON : 0.9868 | MEAN REWARD : -0.8666666666666667
GAME : 50 | EPSILON : 0.9853 | MEAN REWARD : -0.88
```



```
GAME : 3385 | EPSILON : 0.0010 | MEAN REWARD : 0.98
GAME : 3390 | EPSILON : 0.0010 | MEAN REWARD : 1.0
GAME : 3395 | EPSILON : 0.0010 | MEAN REWARD : 1.0
GAME : 3400 | EPSILON : 0.0010 | MEAN REWARD : 1.0
GAME : 3405 | EPSILON : 0.0010 | MEAN REWARD : 1.0
GAME : 3410 | EPSILON : 0.0010 | MEAN REWARD : 1.0
GAME : 3415 | EPSILON : 0.0010 | MEAN REWARD : 1.0
GAME : 3420 | EPSILON : 0.0010 | MEAN REWARD : 1.0
GAME : 3425 | EPSILON : 0.0010 | MEAN REWARD : 1.0
```