


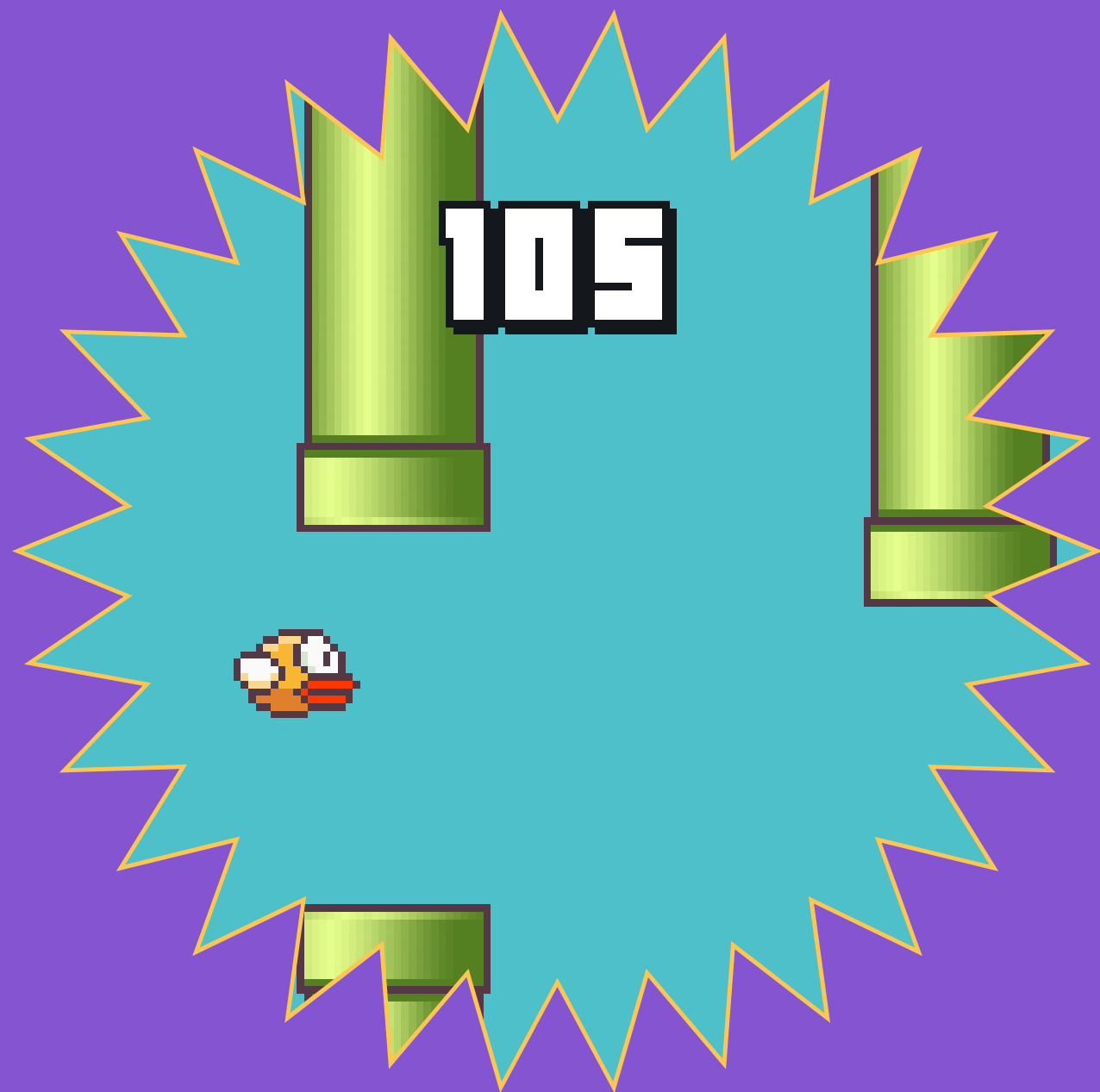
# Reinforced Flappy Bird



Avram Daniel  
Ducal Nicolae  
Tender Laura

Reinforcement Learning

# Introducere



În proiectul nostru am realizat un joc Flappy Bird. Scopul nostru a fost ca, folosind Reinforcement Learning să obținem un scor cât mai bun. Astfel, am folosit două abordări:

## Q-Learning

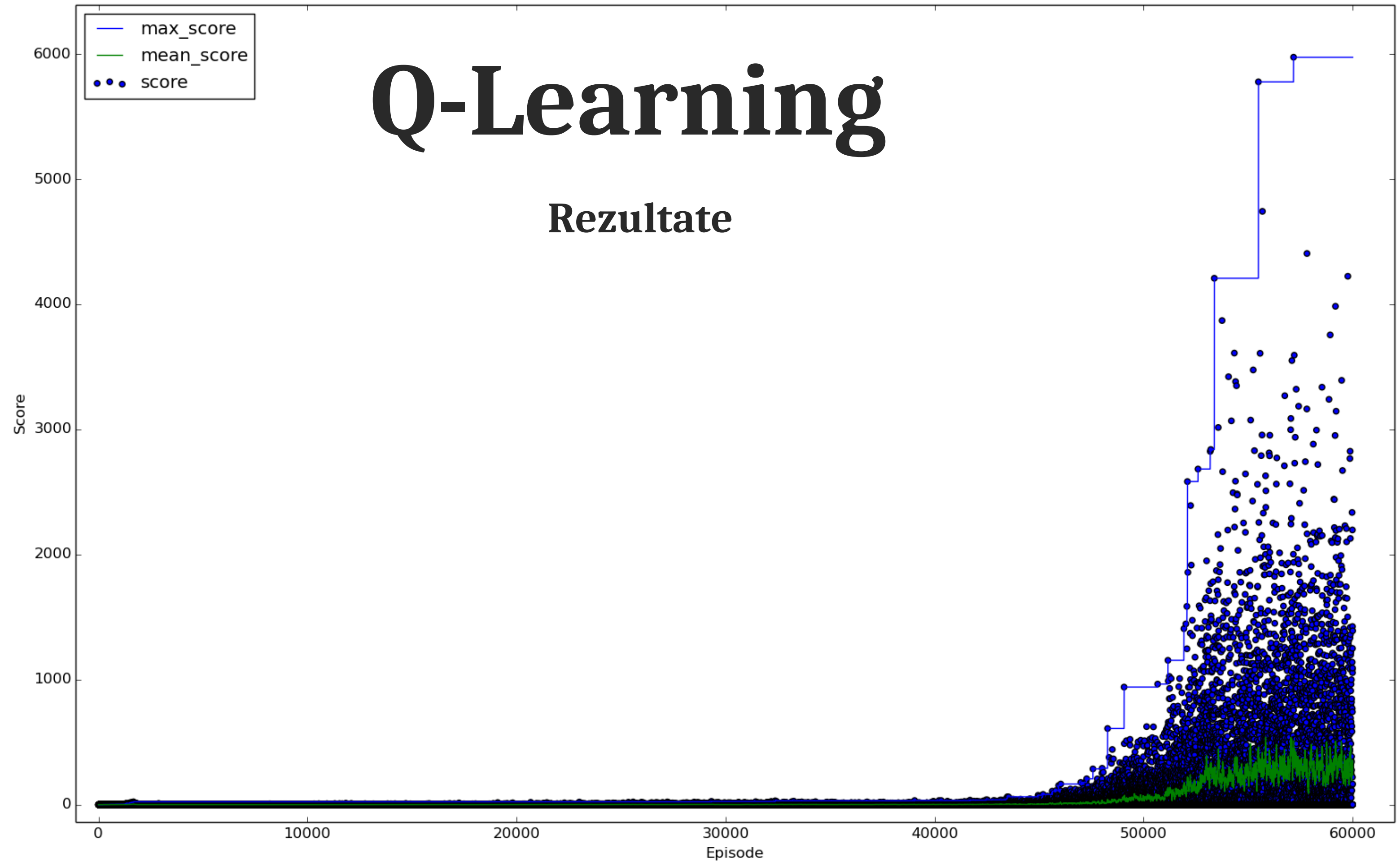
Q-Learning este un algoritm care caută reward-ul maxim folosind un Q-table și Bellman Equation.

## Deep Q-Learning

DQN folosește un deep neural network pentru a estima Q-values.

# Q-Learning

Rezultate



# Implementare Q-Learning

## Q-Table

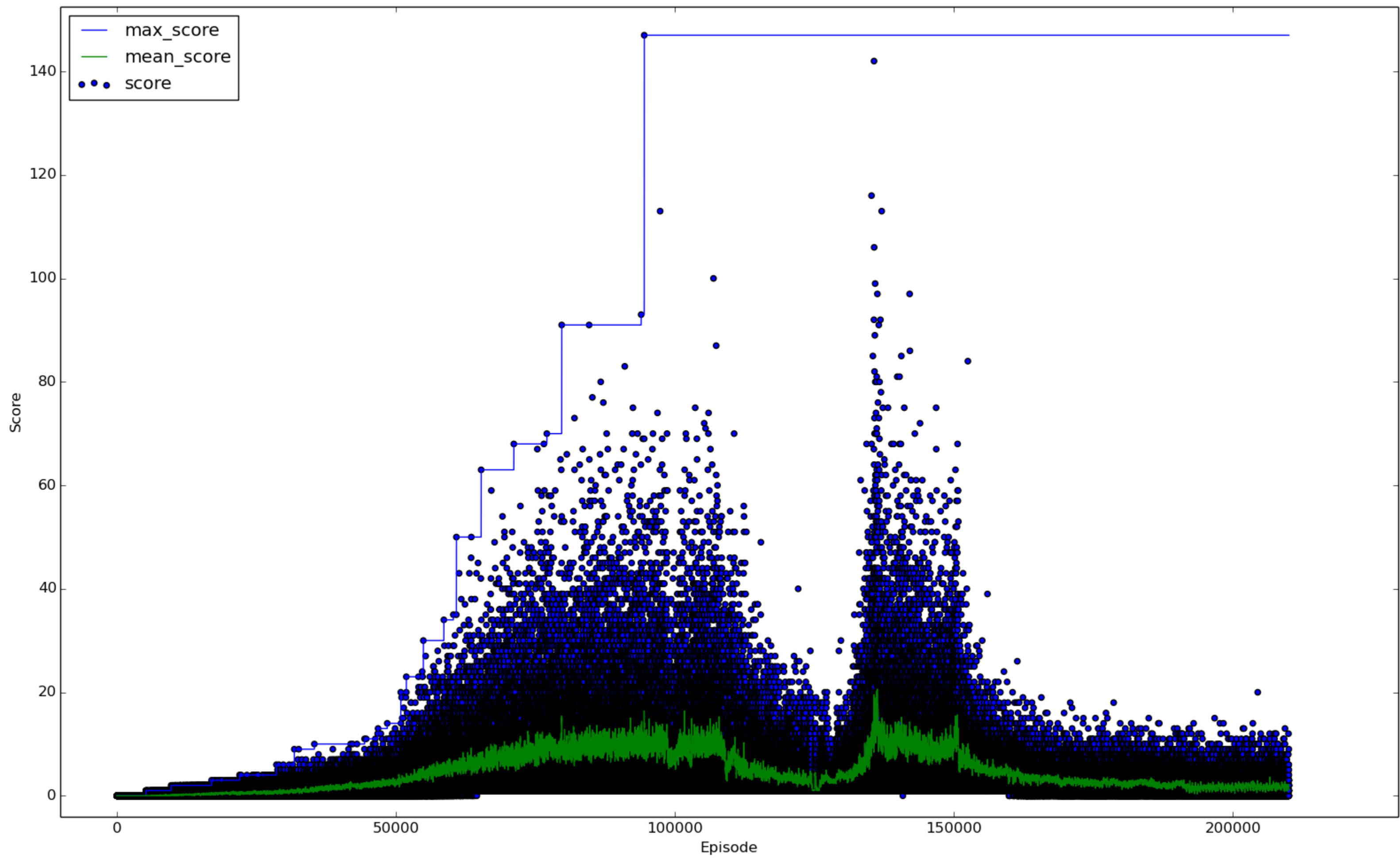
```
self.q_table = np.zeros((X_DISTANCE_MAX, Y_DISTANCE_MAX, MAX_VEL, env.action_space.n))
```

### Dimensiune

Q-Table-ul nostru conține distanța pe axa (Ox față de centrul dintre pipe-uri, distanța pe (Oy față de centru, viteza și numărul de acțiuni (2, a a se ridica sau a nu face nimic). Distanța pe (Ox este în modul, în timp ce cea pe (Oy are semn în funcție de jumătatea ecranului în care se află pasărea. Am decis să adăugăm viteza întrucât îmbunătățea foarte mult rezultatele (graficul se află pe slide-ul următor). Pentru a lucra cu valori pozitive am adăugat offsets la Y-distance și viteză.

### Optimizare

Am împărțit distanțele pe (Ox și (Oy la 10 pentru că altfel numărul nostru de stări era egal cu  $200 \times 800 \times 20 = 3.2 \times 10^6$  iar modelul ar fi învățat foarte greu. Astfel, numărul de stări a fost redus de 100 de ori.



# Q-Learning Implementare

## Alegerea acțiunii

Am ales să nu folosim epsilon greedy strategy. Am obținut rezultate mai bune atunci când foloseam mereu exploitation.

```
action = np.argmax(self.q_table[obs_x, obs_y,  
obs_vel, :])
```

## Reward

Următorul reward a obținut cele mai bune rezultate. Dacă jocul se termină reward-ul este de -1000 sau de 0 altfel.

```
new_obs, reward, done, info = self.env.step(action,  
with_velocity=True)  
reward = -1000 if done else 0
```

# Calcularea valorilor din Q-Table

Formula de calcul:

$$q^{new}(s, a) = (1 - \alpha) \underbrace{q(s, a)}_{\text{old value}} + \alpha \overbrace{\left( R_{t+1} + \gamma \max_{a'} q(s', a') \right)}^{\text{learned value}}$$

Implementarea noastră:

```
self.q_table[obs_x, obs_y, obs_vel, action] = self.q_table[obs_x, obs_y,  
obs_vel, action] * (1 - self.learning_rate) + self.learning_rate *  
(reward + self.discount_rate * np.max(self.q_table[new_obs_x,  
new_obs_y, new_obs_vel, :]))
```

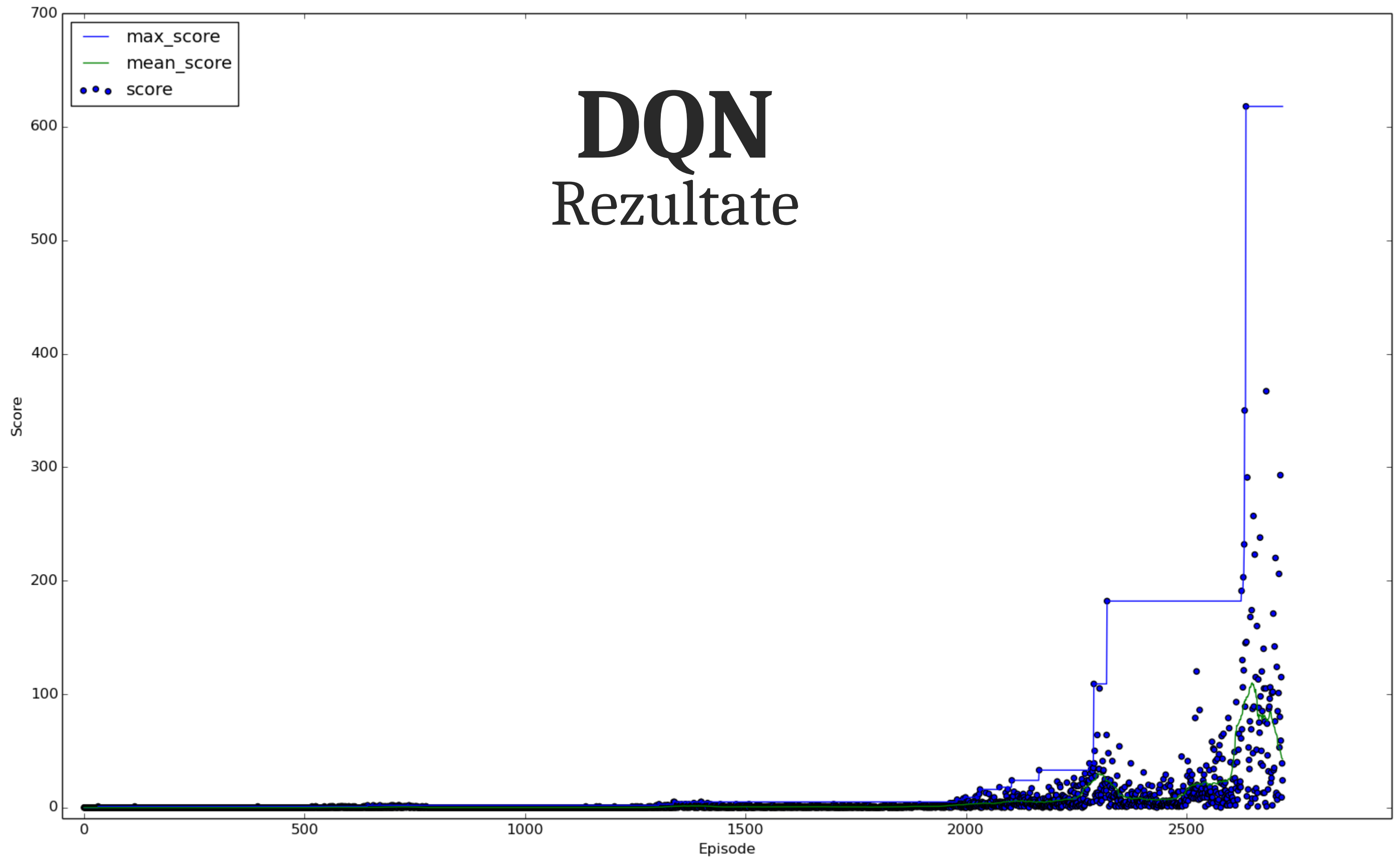
Valoarea constantelor:

**LEARNING\_RATE = 0.1**

**DISCOUNT\_RATE = 0.95**

# DQN

## Rezultate





1

# Rețeaua

```
self.network = nn.Sequential(  
    nn.Conv2d(in_channels=4, out_channels=32, kernel_size=8, stride=4),  
    nn.ReLU(inplace=True),  
    nn.Conv2d(in_channels=32, out_channels=64, kernel_size=4, stride=2),  
    nn.ReLU(inplace=True),  
    nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3),  
    nn.ReLU(inplace=True),  
    nn.Flatten(),  
    nn.Linear(in_features=3136, out_features=512),  
    nn.ReLU(inplace=True),  
    nn.Linear(in_features=512, out_features=actions)  
)
```

Am folosit optimizatorul Adam cu learning rate 1e-6.

2

## Constante

```
self.gamma = 0.99
self.final_epsilon = 0.0001
self.initial_epsilon = 0.01
self.number_of_iterations = 2000000
self.replay_memory_size = 10000
self.batch_size = 32
```

3

## Epsilon Greedy Strategy

```
epsilon_decrements =
np.linspace(self.model.initial_epsilon,
self.model.final_epsilon,
self.model.number_of_iterations)
```

```
epsilon = epsilon_decrements[iteration]
```

```
random_action = random.random() <= epsilon
action = random.randint(0, 1) if random_action
else torch.argmax(q_values).item()
```

## 4

# Replay Memory

Pentru replay memory am folosit un deque de dimensiune 10000. Batch-ul are dimensiune 32 și este ales random din replay memory.

Pentru a estima Q-values am folosit rețeaua prezentată anterior. Similar primei abordări, pentru a calcula valorile am ținut cont de ecuația Bellman. Am calculat loss-ul și l-am backpropagat în rețea.

```
replay_memory =  
deque(maxlen=self.model.replay_memory_size)  
replay_memory.append((state, action, reward,  
new_state, done))  
repl = list(replay_memory)  
batch = random.sample(repl, min(len(repl),  
self.model.batch_size))  
  
output_batch = self.model(new_state_batch)  
y_batch = torch.cat(tuple(reward_batch[i] if  
batch[i][4]  
else reward_batch[i] + self.model.gamma *  
torch.max(output_batch[i])  
for i in range(len(batch))))  
  
#  $q(s, a) = \text{rew} + \gamma * \max(s_{\text{new}}, a=0 \text{ sau } a=1)$   
  
q_value = torch.sum(self.model(state_batch) *  
action_batch, dim=1)
```

# References

- 01 Kevin Chen, *Deep Reinforcement Learning for Flappy Bird*, 2021,  
<[https://cs229.stanford.edu/proj2015/362\\_report.pdf](https://cs229.stanford.edu/proj2015/362_report.pdf)>
- 02 KK'S BLOG, 2019  
<<https://www.fromkk.com/posts/using-ddqn-to-play-flappy-bird/>>
- 03 DEEPLIZARD, 2018,  
<<https://deeplizard.com/learn/video/qhRNvCVVJaA>>
- 04 DEEPLIZARD, 2018,  
<<https://deeplizard.com/learn/video/wrBUkpiRvCA>>