ELTE | FACULTY OF INFORMATICS

DEEP Q-LEARNING

Deep Reinforcement Learning
Zoltán Barta, PhD student

ELTE | IK
DEPARTMENT OF ARTIFICIAL INTELLIGENCE

Outline

- Deep Reinforcement Learning
- **D**eep **Q** **N**etwork – DQN
- Double DQN
- Dueling DQN
- DQN in Atari environments

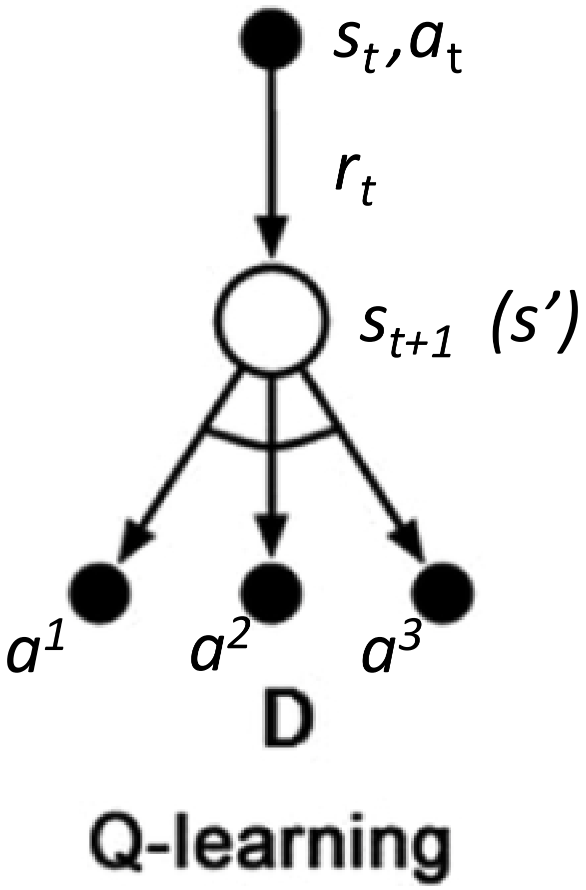
Recap – Q-learning

- Off-policy, model-free RL algorithm that learns the optimal **action-value function** *Q(s, a).*
- $Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a))$
  - **Learning Rate (α)**: Controls how much new information is used.
  - **Discount Factor (γ)**: Weighs future rewards.

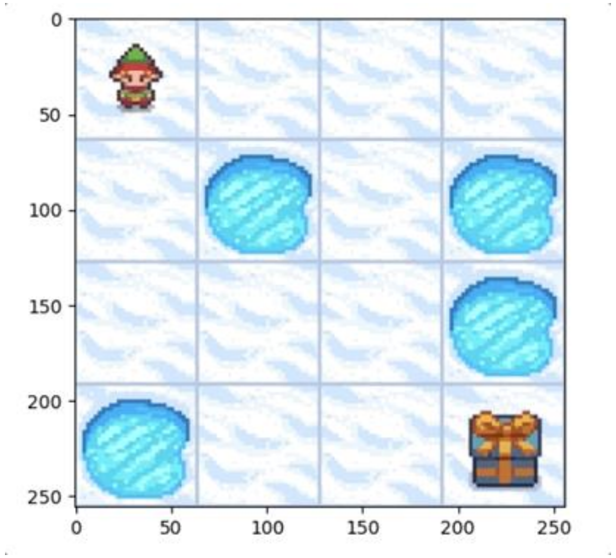
Stores *Q(s, a)* values in a table.

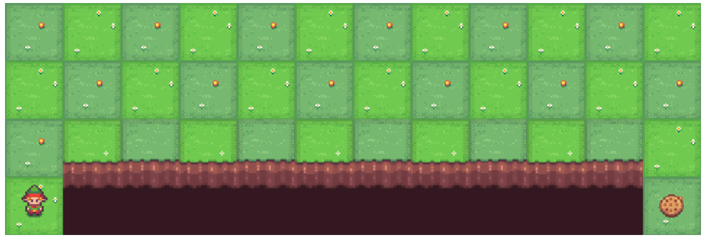
- **Converges** to the optimal **Q-function** *Q*(s, a)*
- Extract the optimal policy:

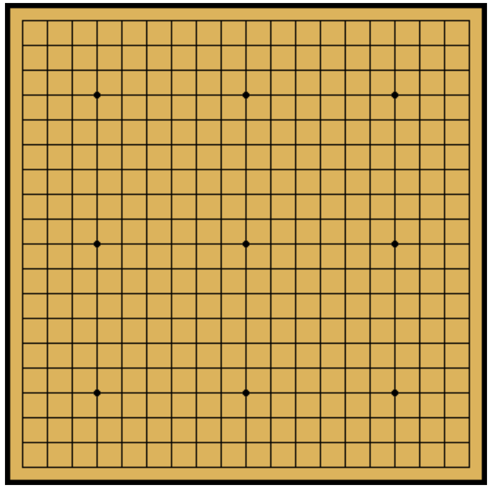
$$\pi^*(s) = \arg\max_a Q^*(s,a)$$

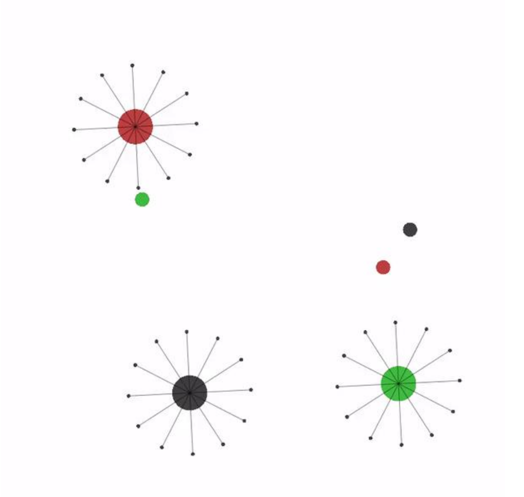


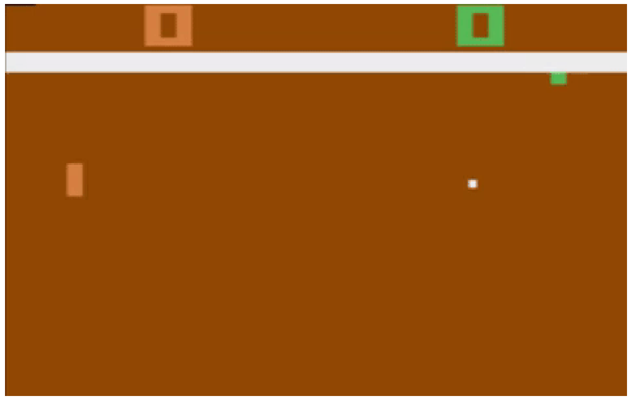
State and action spaces

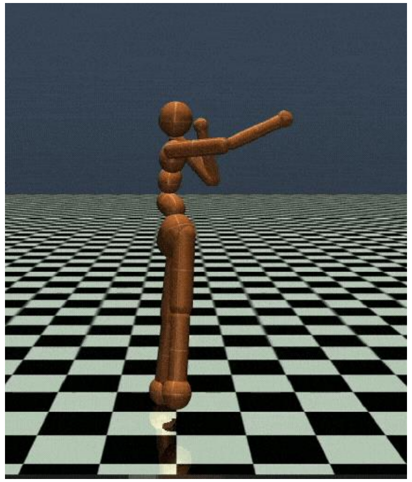










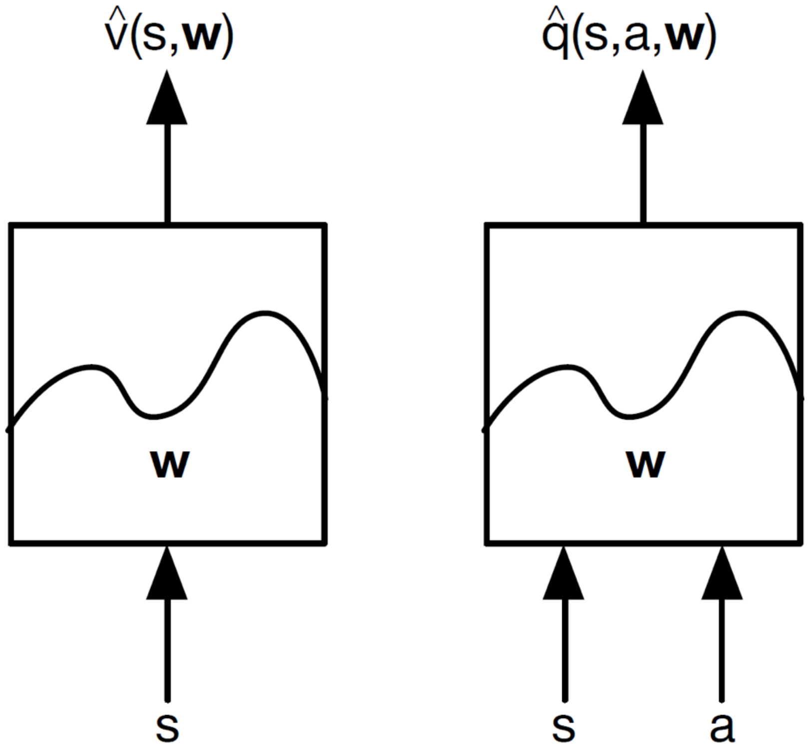


Function Approximation

- Generalize state/action representation
- Instead of a lookup table for *V(s)* or *Q(s,a)*, we use a **parameterized function** *V(s; w)* or *Q(s, a; w).*
- The goal is to find w that minimizes the prediction error (loss)
- Loss can be MSE:

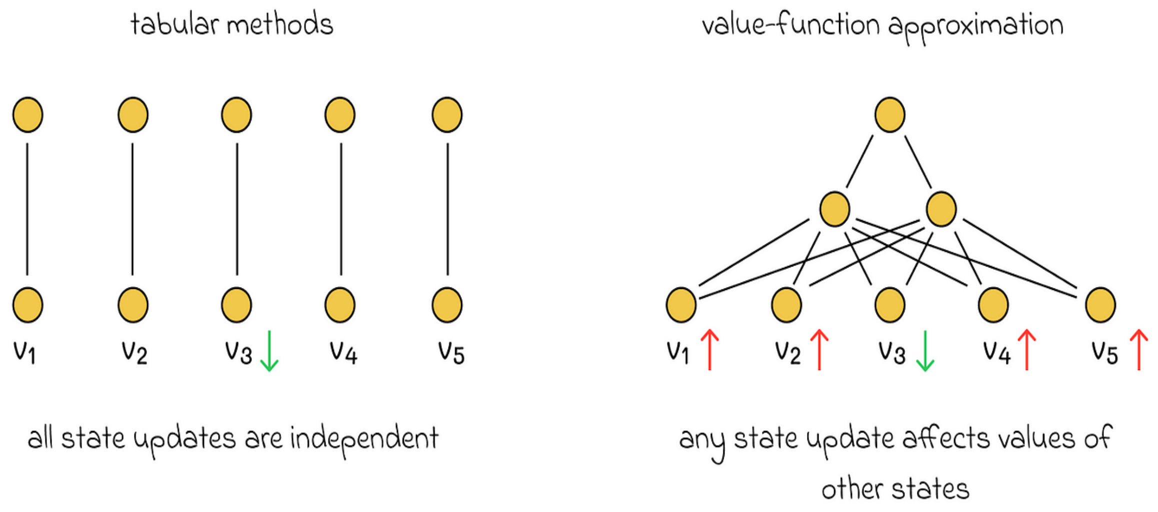
$$J(w) = \mathbb{E}_{\pi}\left[(Q^{\pi}(s,a) - \hat{Q}^{\pi}(s,a,w)^{2}\right]$$

- Find the local minimum for loss (optimization)



Function Approximation

- Neural Networks as parameterized function
- Input is the state
- Output is the state-action value – for each action
- A plathera of tools are available:
  - Multi Layered Perceptron
  - Convolutional Neural Network
  - Recurrent Neural Network



Function Approximation

- Two types learning occur in Deep RL:
  - The agent learns by interacting with the environment and optimizes $Q(s, a)$ using the Bellman equation.
  - The neural network is trained to approximate the Q-function by minimizing the loss between predicted and target Q-values.
- Training data is generated by the agent by interacting with the environment
- Training data distribution is changing – non stationary
- Samples are correlated

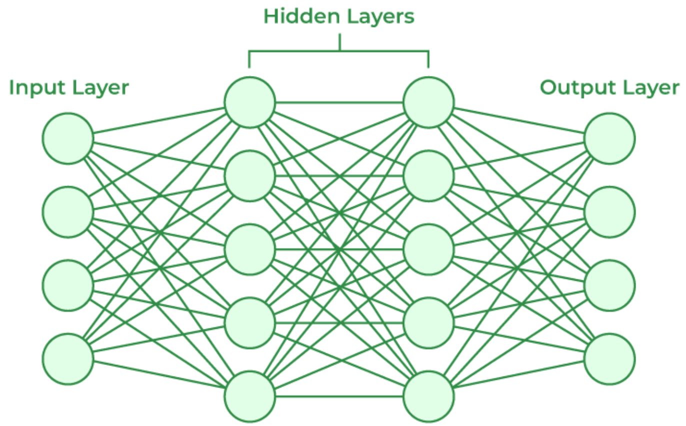
Deep Q Learning (DQN)

- Sample correlation problem
- Remove correlation by storing the experiences in a **Replay Buffer**
- Sample the dataset randomly
- Compute the target value
- Update the network weights

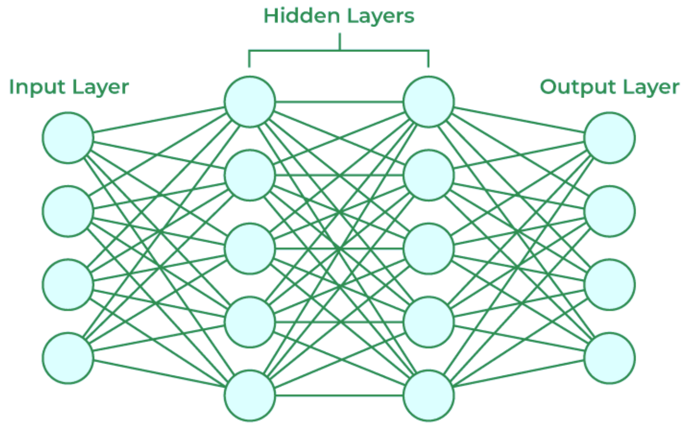
s_1, a_1, r_1, s_2
s_2, a_2, r_2, s_3
….
s_n, a_n, r_n, s_{n+1}

Deep Q Learning (DQN)

- Non-stationarity problem
- If we update Q-values using a single neural network (policy network), its targets keep shifting
- Maintain a separate, slowly updated network (target network)
- Use the target network to compute the Q-learning update
- Every N steps, update the target network with the current network weights *w*



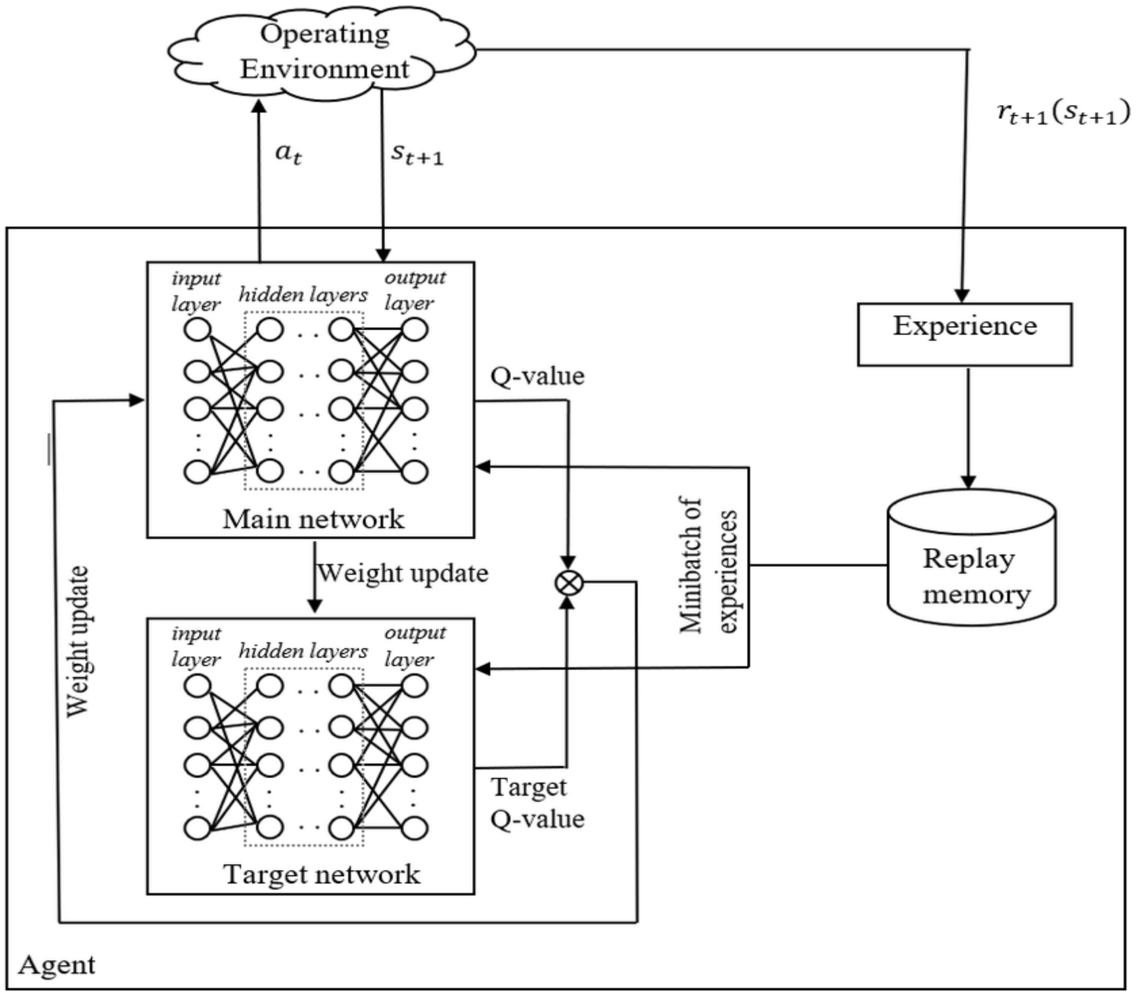
Policy network



Target network

Updating the target network

- Hard update
  - Copy the policy network weights at every N step
- Soft update
  - Slowly blend the target network weights (w^-) towards the policy by some factor τ:
    $$w^- \leftarrow \tau w + (1 - \tau) w^-$$
  - Typically used in continous action spaces

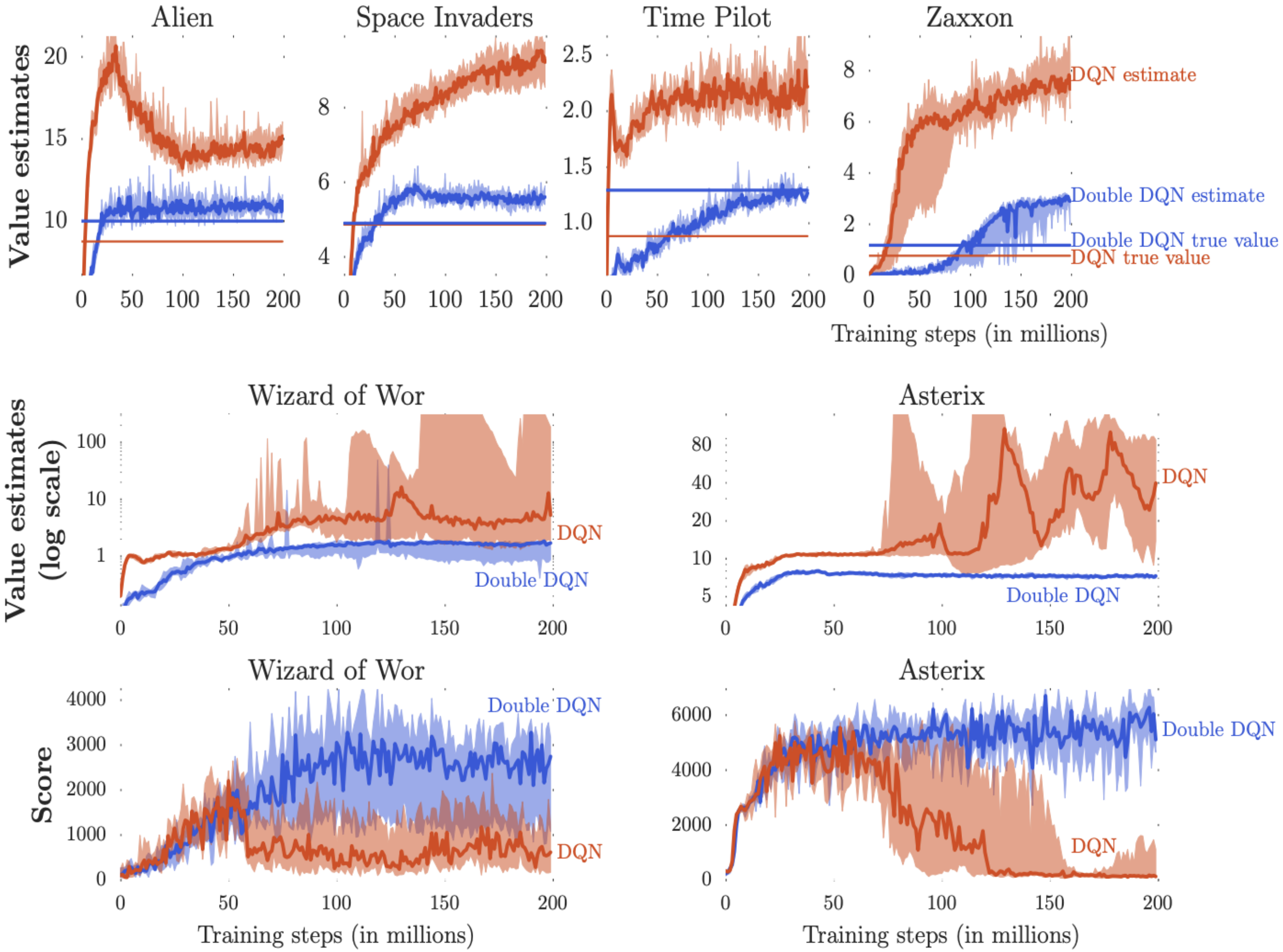


DQN pseudocode

```
Init D = []
Init Q = NN()
Init Q_target = NN()
for episode in range(max_episodes):
    while not done:
        if random() < epsilon:
            a = random_action()
        else:
            a = argmax(Q(s))  # Select action with highest Q-value
        s_next, r, done, _ = env.step(a)
        D.append((s, a, r, s_next, done))
        if len(D) > batch_size:
            batch = sample_minibatch(D, batch_size)
            for (s, a, r, s_next, done) in batch:
                if done:
                    y = r  # No future reward for terminal state
                else:
                    y = r + gamma * max(Q_target(s_next))  # Bellman equation
                loss = (y - Q(s, a))^2  # Mean Squared Error (MSE)
                optimizer.zero_grad()
                loss.backward()
                optimizer.step()
        s = s_next
    if episode % target_update_freq == 0:
        Q_target = copy(Q)
```

Double DQN

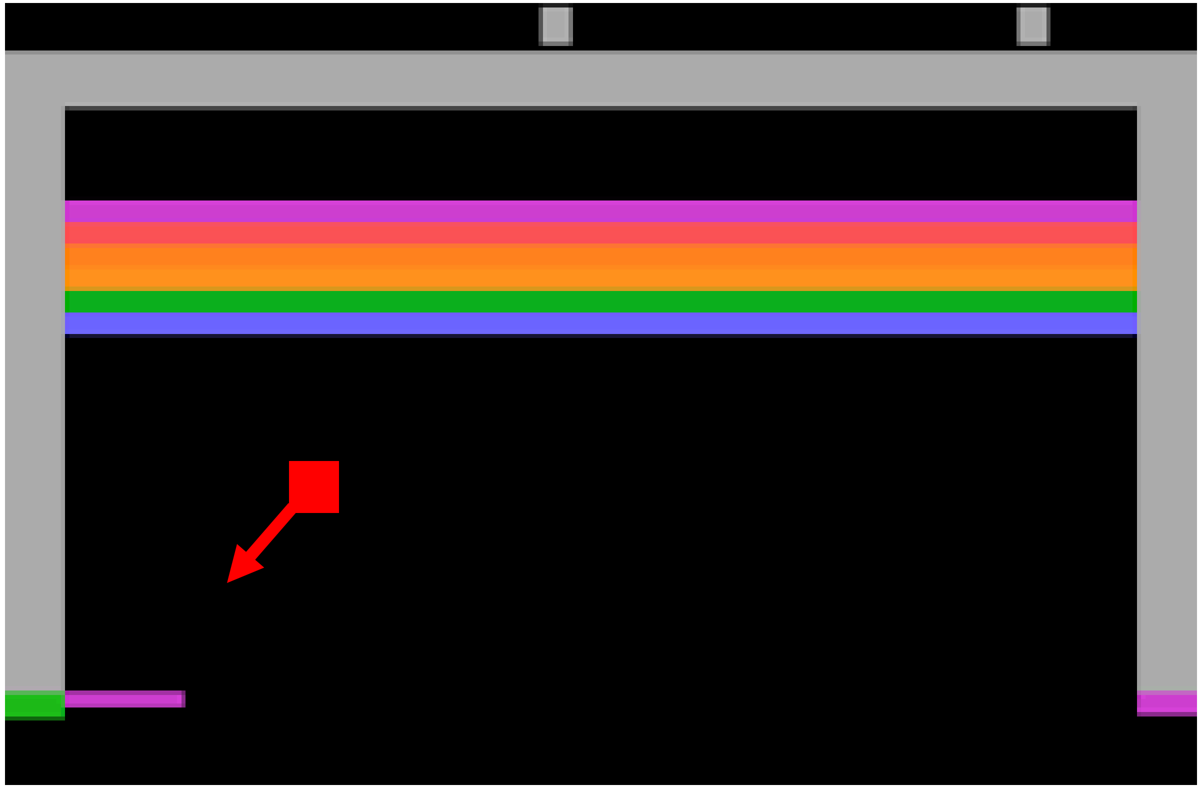
- Overestimation bias
- Present in Q-learning, but more prominent in DQN
  - NN's imperfect Q-value predictions
  - Max operator amplifies noise
  - Bootstrapping
- Solution - Seperate action selection and evaluation

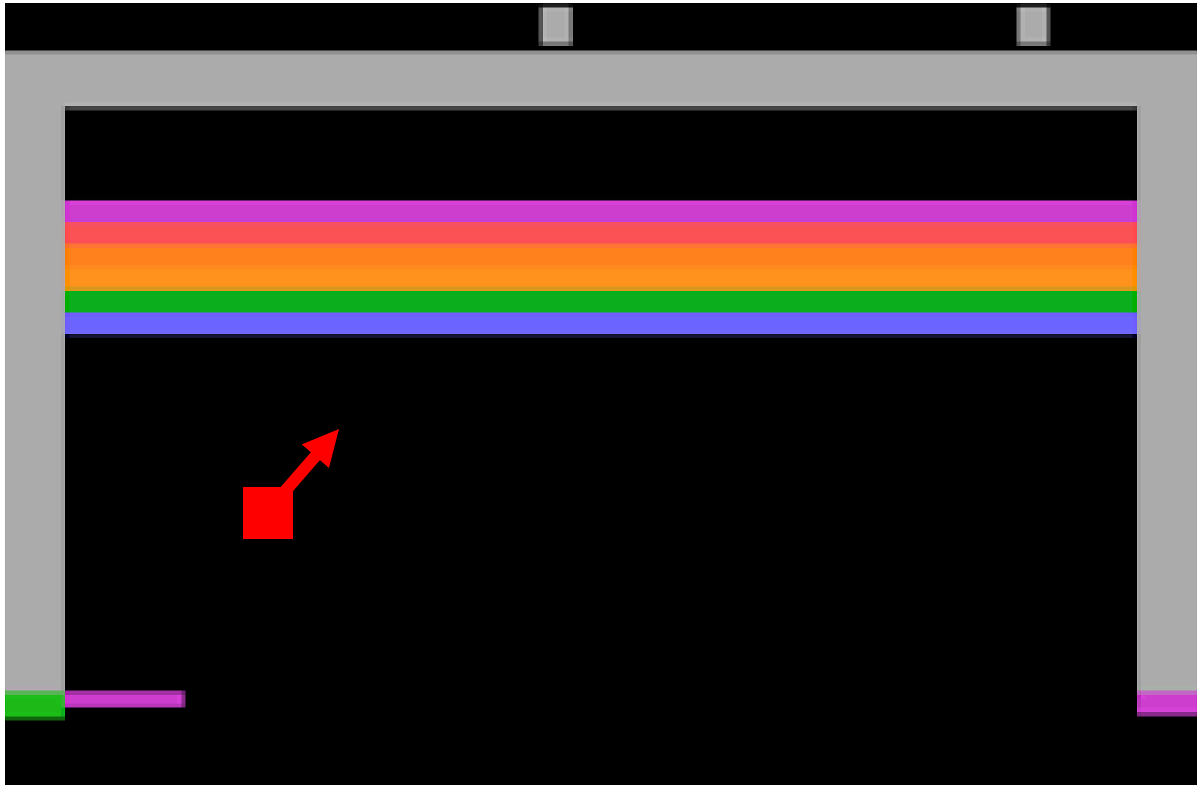


DoubleDQN pseudocode

```
Init D = []
Init Q = NN()
Init Q_target = NN()
for episode in range(max_episodes):
    while not done:
        if random() < epsilon:
            a = random_action()
        else:
            a = argmax(Q(s))  # Select action using online Q-network
        s_next, r, done, _ = env.step(a)
        D.append((s, a, r, s_next, done))
        if len(D) > batch_size:
            batch = sample_minibatch(D, batch_size)
            for (s, a, r, s_next, done) in batch:
                if done:
                    y = r  # No future reward for terminal state
                else:
                    a_next = argmax(Q(s_next))
                    y = r + gamma * Q_target(s_next, a_next)
                loss = (y - Q(s, a))^2  # Mean Squared Error (MSE)
                optimizer.zero_grad()
                loss.backward()
                optimizer.step()
        s = s_next
    if episode % target_update_freq == 0:
        Q_target = copy(Q)
```

Dueling Deep Q Network

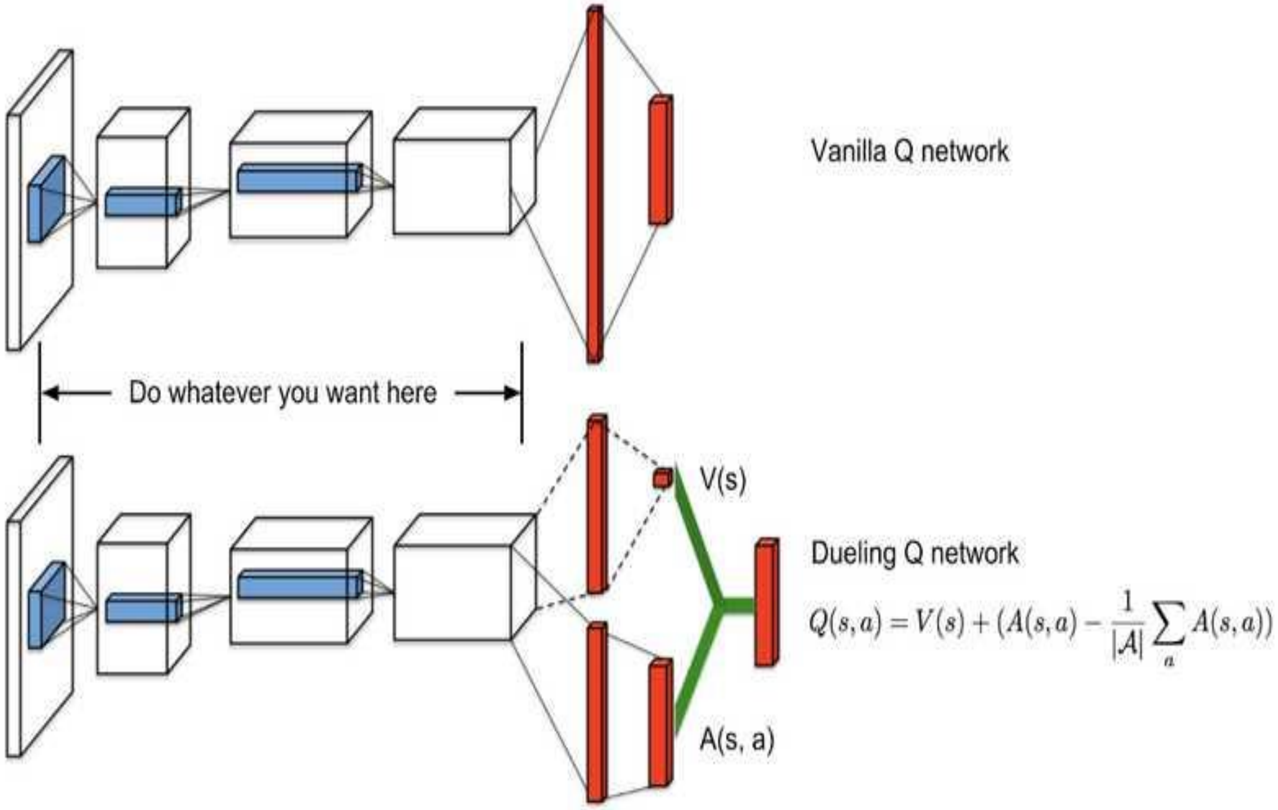




Dueling Deep Q Network

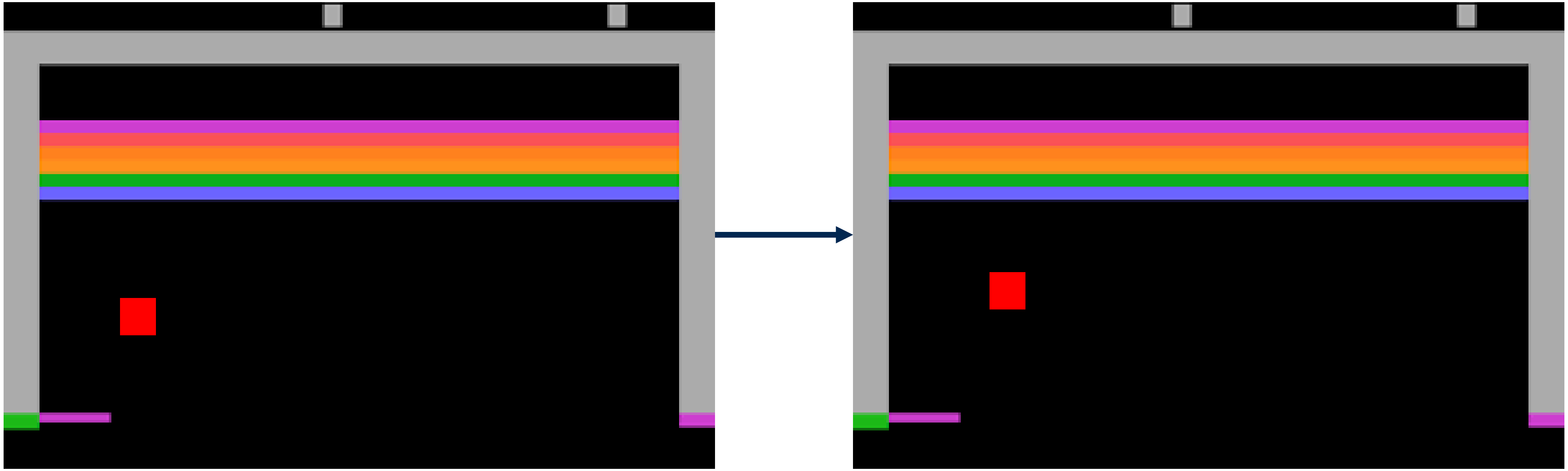
- Vanilla architecture treats every action in every state equally important
- Dueling architecture calculates:
  - How good a state is independent of actions -*V(s)*
  - How much better an action is compared to the average in that state – *A(s,a)*

$$Q(s,a) = V(s) + \left(A(s,a) - \frac{1}{|A|}\sum_{a'} A(s,a')\right)$$



DQN in Atari

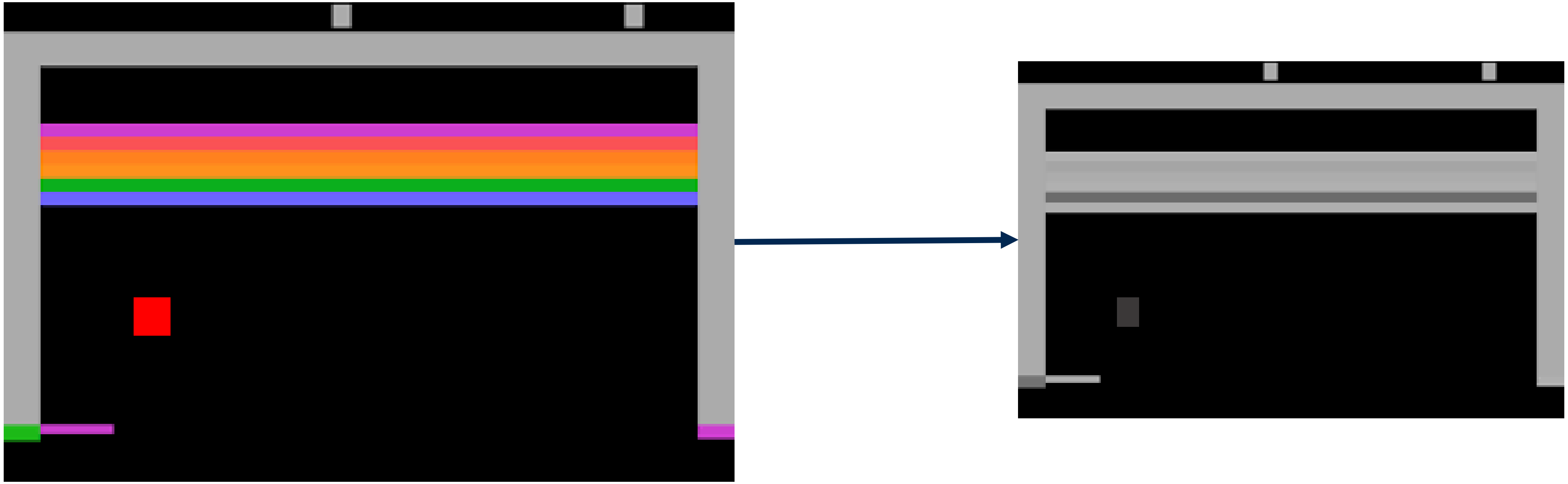
- Frameskipping
  - Take every N-th frame



t t+1

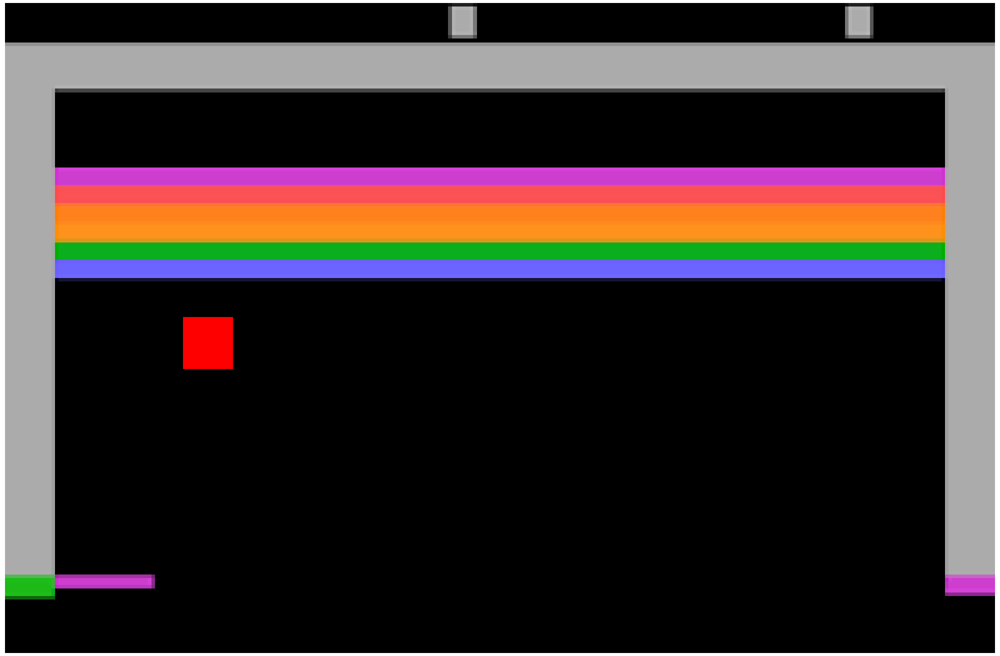
DQN in Atari

- Grayscaling
- Downsizing



DQN in Atari

- Framestacking
  - Have some information about the dynamic components

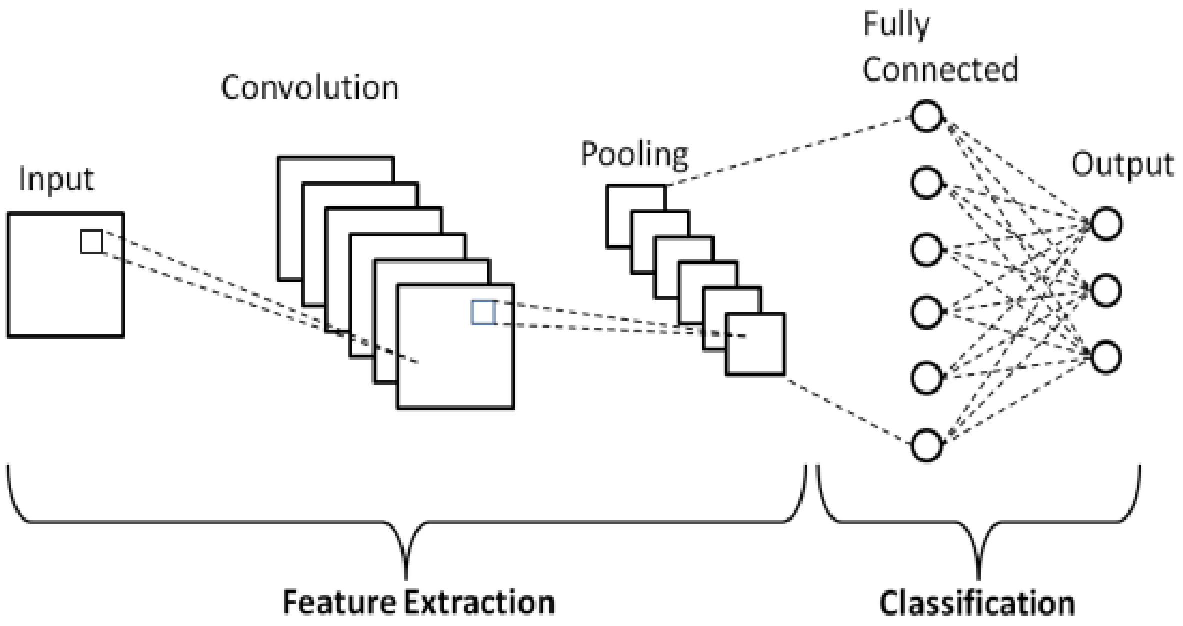


t

t, t+1, t+2, t+3

DQN in Atari

- End-to-end learning from pixels
- Input: raw pixel images
- Output: 1 button ”push” from 18 buttons/joystick
- CNN for feature extraction
- MLP for Q estimation
- Reward is the score of the player



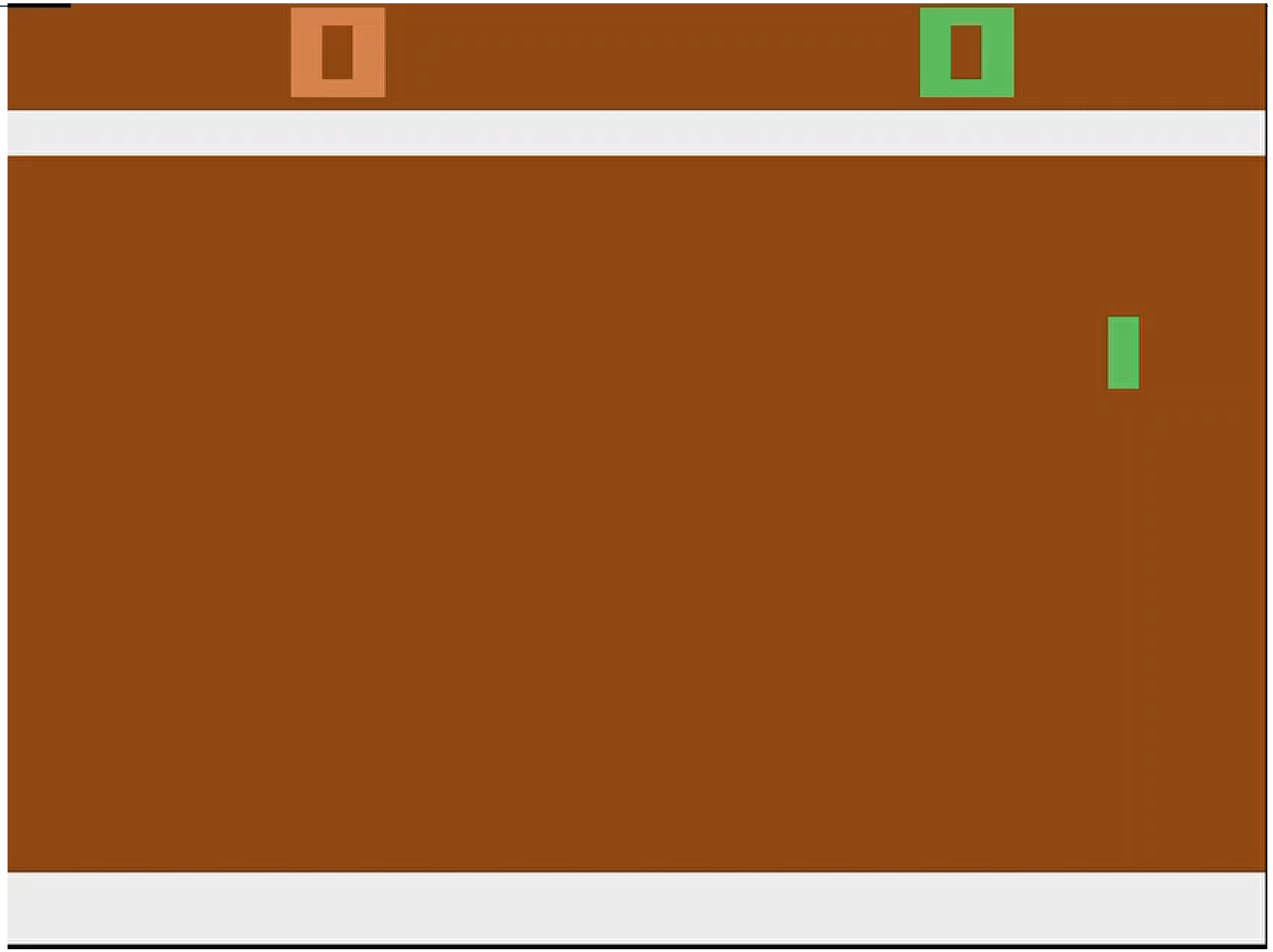
DQN in Atari – Good to know

- Most of the Atari games has ”RAM” observation type
- Much faster to train on, because the state space is the state of the RAM, instead of images
- No transformations needed

Env-id	obs_type=	frameskip=	repeat_action_probability=
Pong-v0	`"rgb"`	`(2, 5)`	`0.25`
Pong-ram-v0	`"ram"`	`(2, 5)`	`0.25`
Pong-ramDeterministic-v0	`"ram"`	`4`	`0.25`
Pong-ramNoFrameskip-v0	`"ram"`	`1`	`0.25`
PongDeterministic-v0	`"rgb"`	`4`	`0.25`
PongNoFrameskip-v0	`"rgb"`	`1`	`0.25`
Pong-v4	`"rgb"`	`(2, 5)`	`0.0`
Pong-ram-v4	`"ram"`	`(2, 5)`	`0.0`
Pong-ramDeterministic-v4	`"ram"`	`4`	`0.0`
Pong-ramNoFrameskip-v4	`"ram"`	`1`	`0.0`
PongDeterministic-v4	`"rgb"`	`4`	`0.0`
PongNoFrameskip-v4	`"rgb"`	`1`	`0.0`
ALE/Pong-v5	`"rgb"`	`4`	`0.25`
ALE/Pong-ram-v5	`"ram"`	`4`	`0.25`

Homework

- Implement the Dueling architecture
- Train a Dueling Double DQN agent on a RAM version of Atari Pong (or any atrai game)
- Use the provided notebook
- Deadline: **2025.04.08**



Thank you for your attention!