Deep Reinforcement Learning

Presentation Subject:

Deep Reinforcement Learning with Double Q-learning

(Double Deep Q-learning)

Course professor: Dr.Ebadzadeh

Presented by: Shervin Halat

Amirkabir University of Technology

Overview

- The popular Q-learning algorithm overestimate action values under certain conditions
  - Insufficiently flexible function approximator
  - Noise
- DQN algorithm suffers from overestimation
- Modifications mainly aim ‘Target Function’
- Double Deep Q-learning reduces overestimations

Q-learning (1988)

- Main idea:
  - To estimate optimal action values
- Approach:
  - A form of temporal difference learning (TD)

$$Y_t^{\mathrm{Q}} \equiv R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \boldsymbol{\theta}_t)$$

Double Q-learning (2010)

- Main idea:
  - Decouples action selection and evaluation as against Q-learning and DQN
- Approach:
  - two value functions are learned by assigning each experience randomly to update one of the two value functions
    - Two set of weights

Q-learning:

$$Y_t^{\mathrm{Q}} = R_{t+1} + \gamma Q(S_{t+1}, \operatorname*{argmax}_a Q(S_{t+1}, a; \boldsymbol{\theta}_t); \boldsymbol{\theta}_t)$$

Double Q-learning:

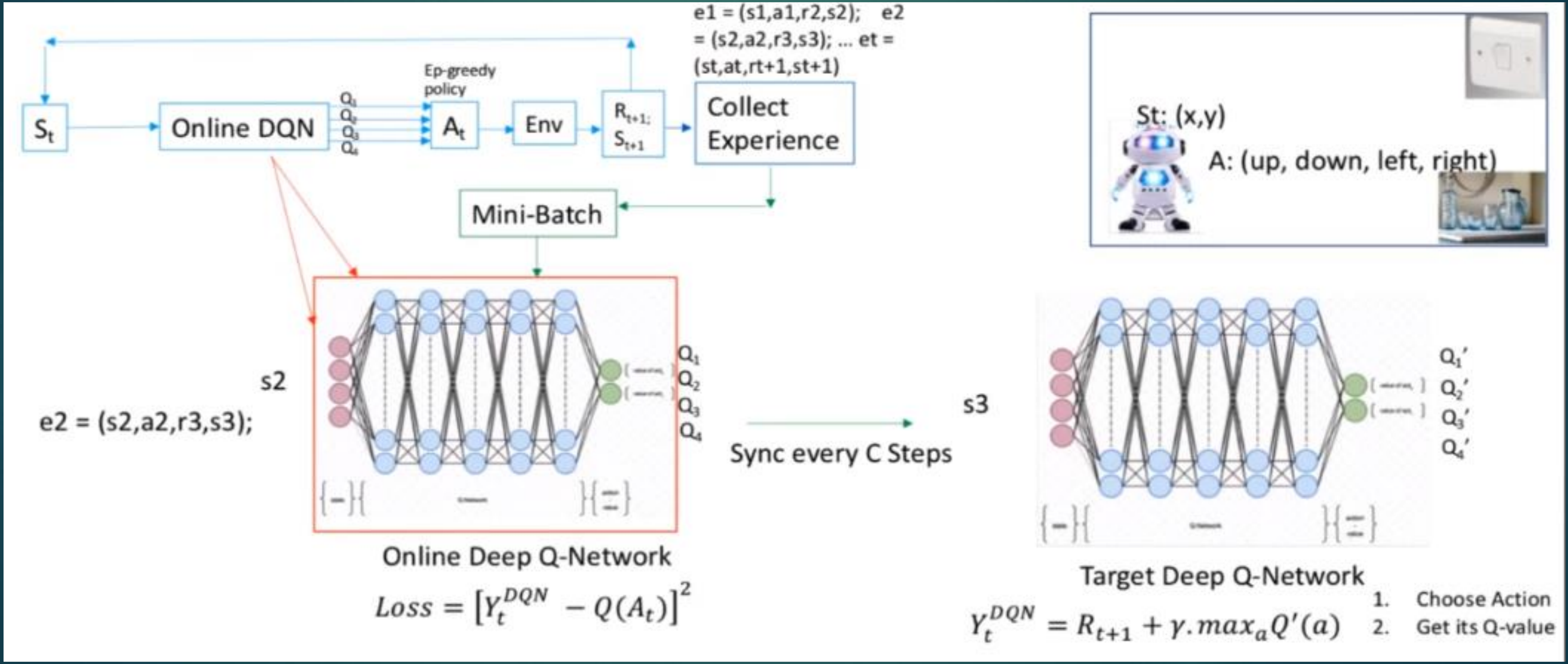
$$Y_t^{\mathrm{DoubleQ}} \equiv R_{t+1} + \gamma Q(S_{t+1}, \operatorname*{argmax}_a Q(S_{t+1}, a; \boldsymbol{\theta}_t); \boldsymbol{\theta}'_t)$$

Deep Q Network (2015)

- Main ideas:
  - Target network
  - Experience replay (Replay Buffer)

$$Y_t^{\text{DQN}} \equiv R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \boldsymbol{\theta}_t^-)$$

Deep Q Network (2015)



Overestimation Reasons

- Q-learning overestimation (1993):
  - If action values contain random errors uniformly distributed in an interval of **[-ε, +ε]**
  - → Then target is overestimated up to $\gamma\epsilon\frac{m-1}{m+1}$ where 'm' is number of actions
  - → (Upper bound)
  - → Double Q-learning proposed

- **DDQN paper: Estimation errors of any kind can induce an upward bias, regardless of whether these errors are due to environmental noise, function approximation**

Unavoidable Overestimation

- Theorem 1. (Lower bound)
  - Consider a state s in which all the true optimal action values are equal at $Q_*(s,a) = V_*(s)$

$$V_*(s) = \max_{a \in \mathcal{A}} Q_*(s,a)$$

  - Let Qt be arbitrary value estimates that are on the whole unbiased:

$$\sum_a (Q_t(s,a) - V_*(s)) = 0,$$

  - but that are not all correct (C > 0)

$$\frac{1}{m} \sum_a (Q_t(s,a) - V_*(s))^2 = C$$

Unavoidable Overestimation

- Under mentioned conditions:

$$\max_a Q_t(s,a) \geq V_*(s) + \sqrt{\frac{C}{m-1}}$$

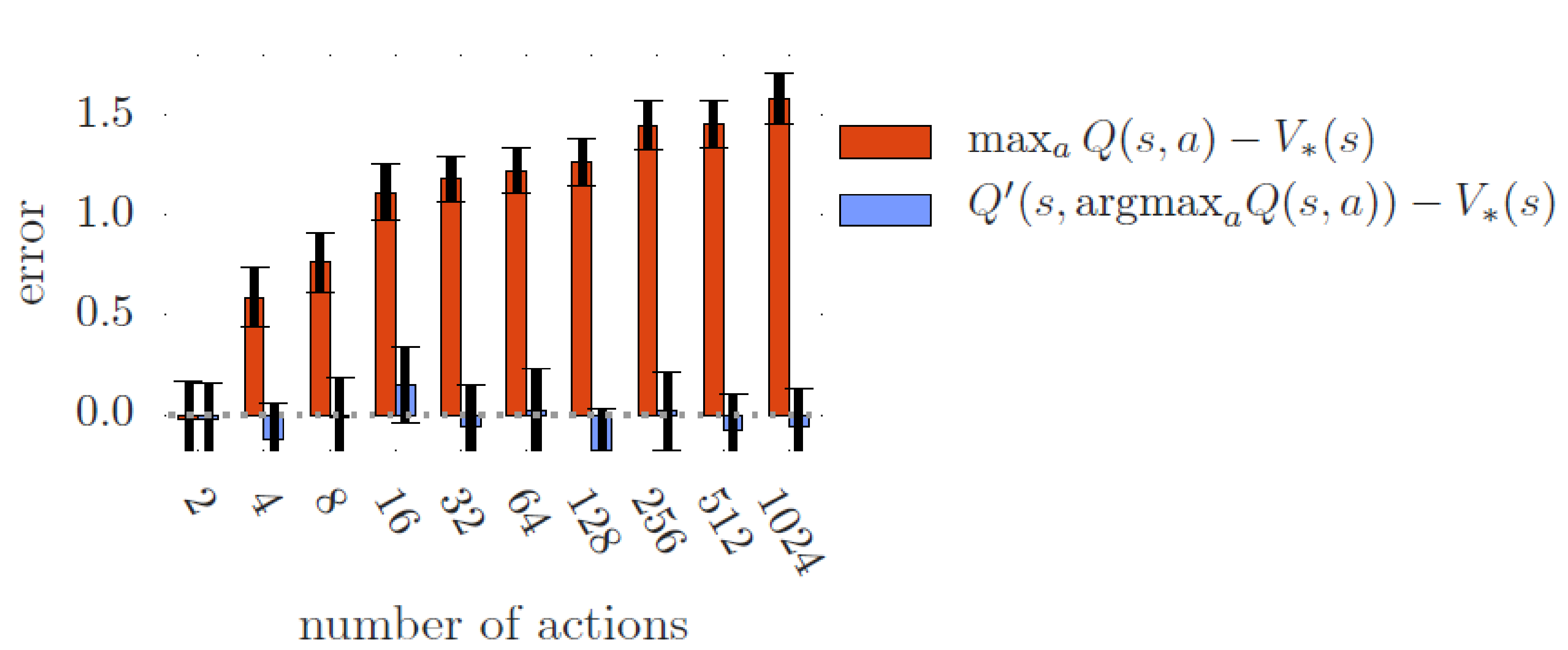
- If "Q(s,a) – Q*(s,a)" has uniform random distribution in [-1,1] then:

$$Q_t(s,a) \geq Q_*(s,a) + \frac{m-1}{m+1}$$

- **Under the same conditions, the lower bound on the absolute error of the Double Q-learning estimate is zero (proof in appendix)**

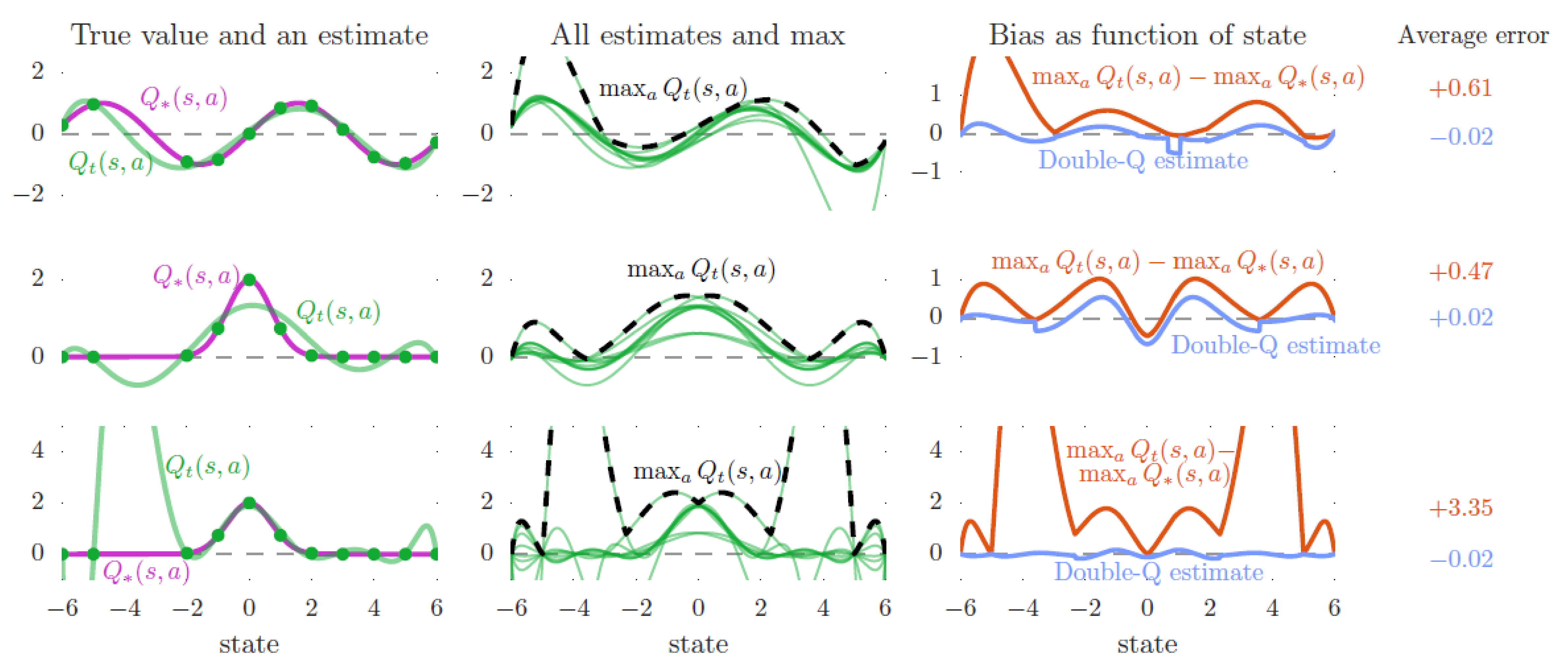
Overestimations in Experiments

- Bias in a single Q-learning update
  - When action values are **Q(s,a) = V*(s) + ε_a**
    - **(ε_a are independent standard normal random variable errors)**



Overestimations in Experiments

- Top row: **Q*(s,a) = sin(s)** & Middle and bottom row: **Q*(s,a) = 2.exp(-s²)**

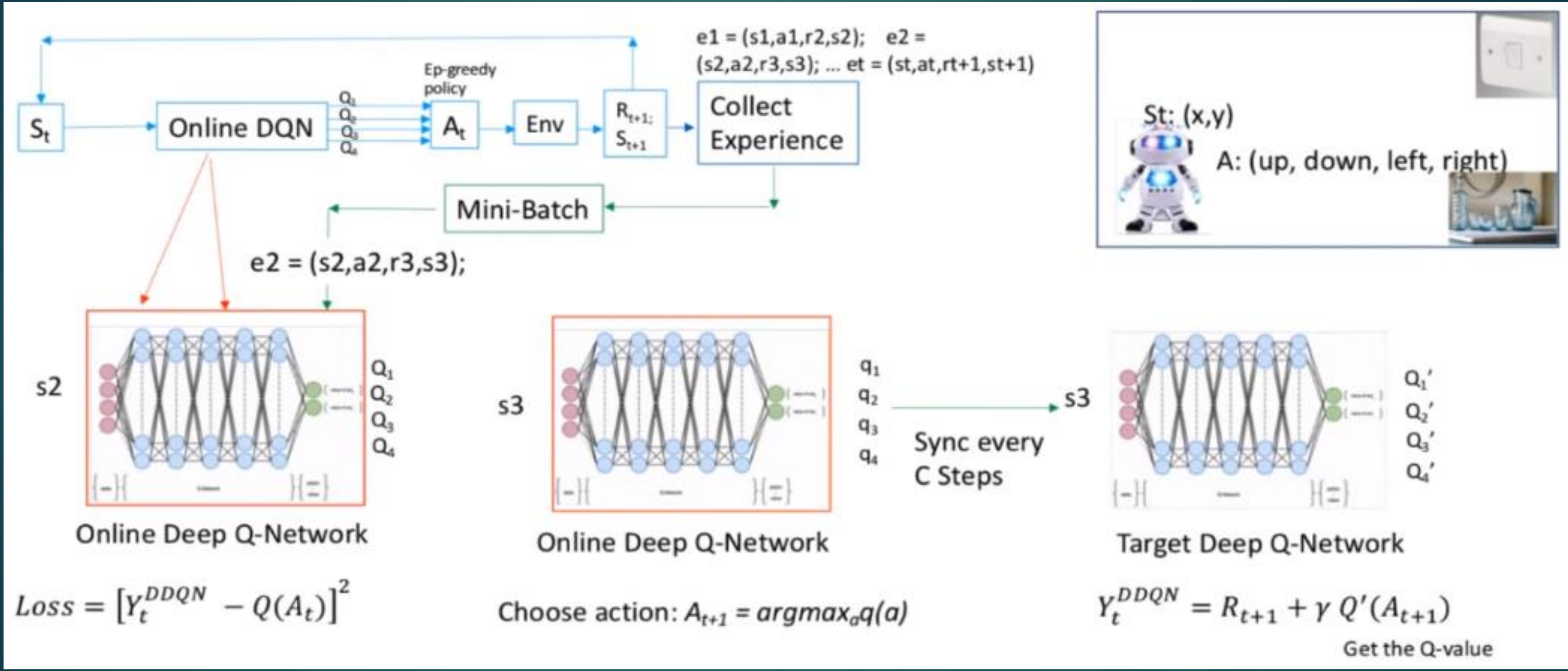


Double Deep Q Network (Dec, 2015)

- Main idea:
  - Reduce overestimation by decomposing max operation (Action selection and evaluation)
  - Not fully decoupled!
  - Minimum computational overhead
- Architecture:
  - Online network: action selection
  - Target network: action evaluation (estimation)

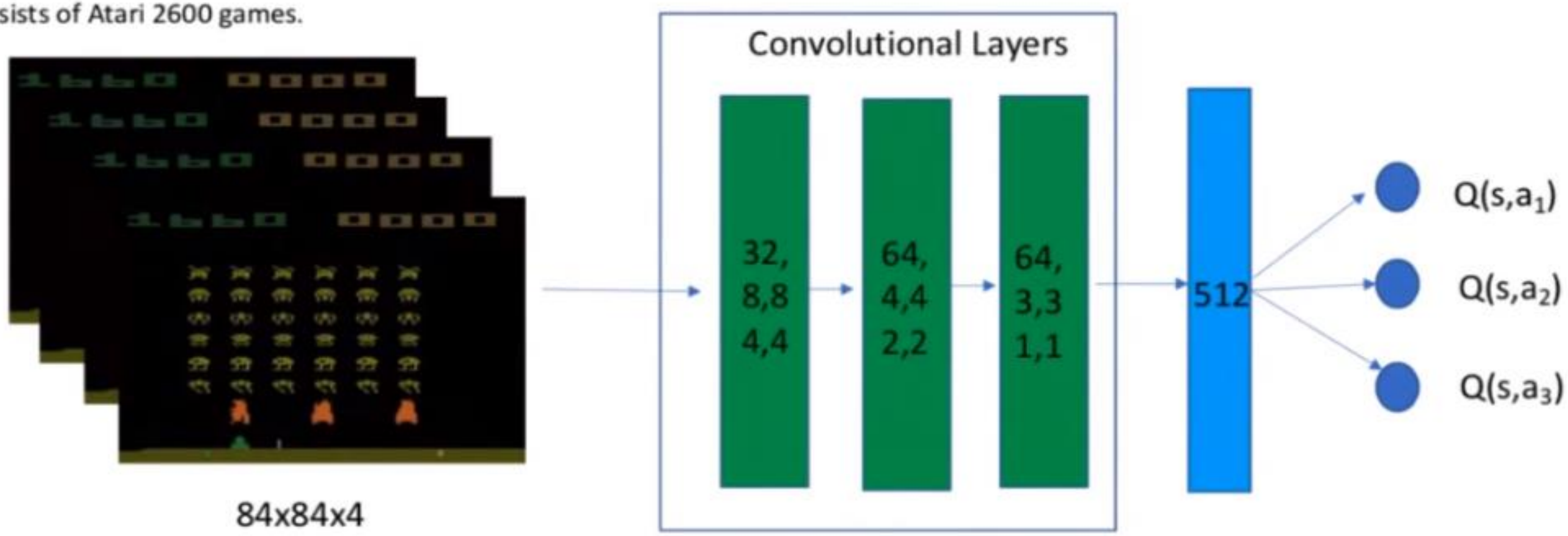
$$Y_t^{\text{DoubleDQN}} \equiv R_{t+1} + \gamma Q(S_{t+1}, \underset{a}{\operatorname{argmax}}\, Q(S_{t+1}, a; \boldsymbol{\theta}_t), \boldsymbol{\theta}_t^-)$$

Double DQN architecture



DDQN network architecture

- Testbed consists of Atari 2600 games.



- Trained for 200M frames (or approx. 1 week)
- Backprop using RMSProp algorithm
- Discount rate = 0.99
- learning rate = 0.00025, steps between target network update = 10,000, steps = 50M, size of memory = 1M tuples, mini-batch size =32, network update interval = 4, epsilon decreasing linearly from 1 to 0.1 over 1M steps.

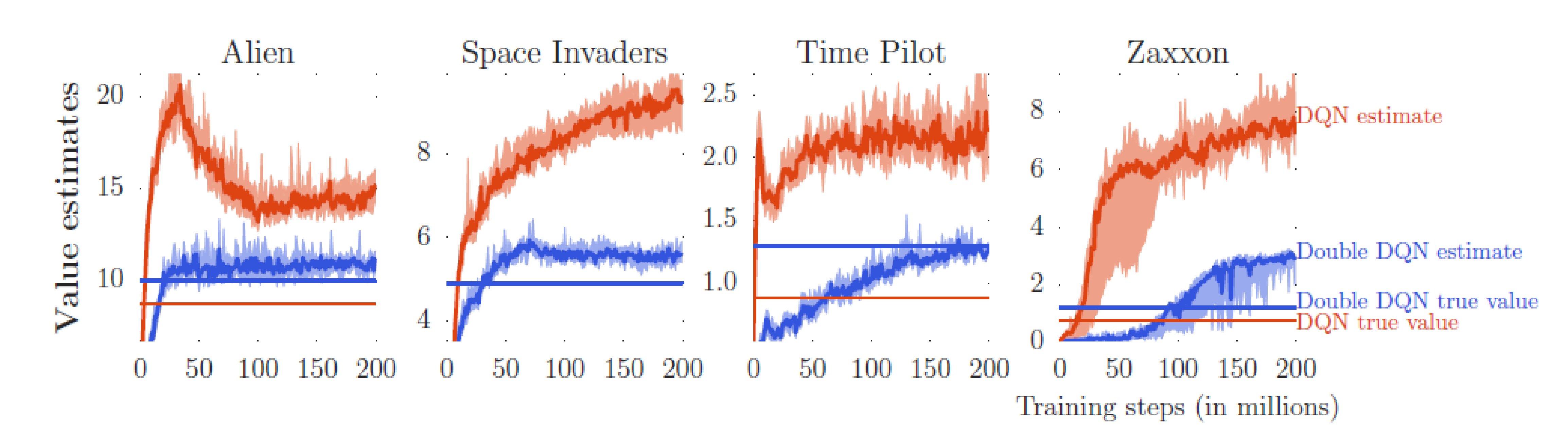
Empirical Results

- Comparison of DDQN vs DQN
  - Results on overoptimism
  - Quality of the learned policies
  - Robustness to Human starts

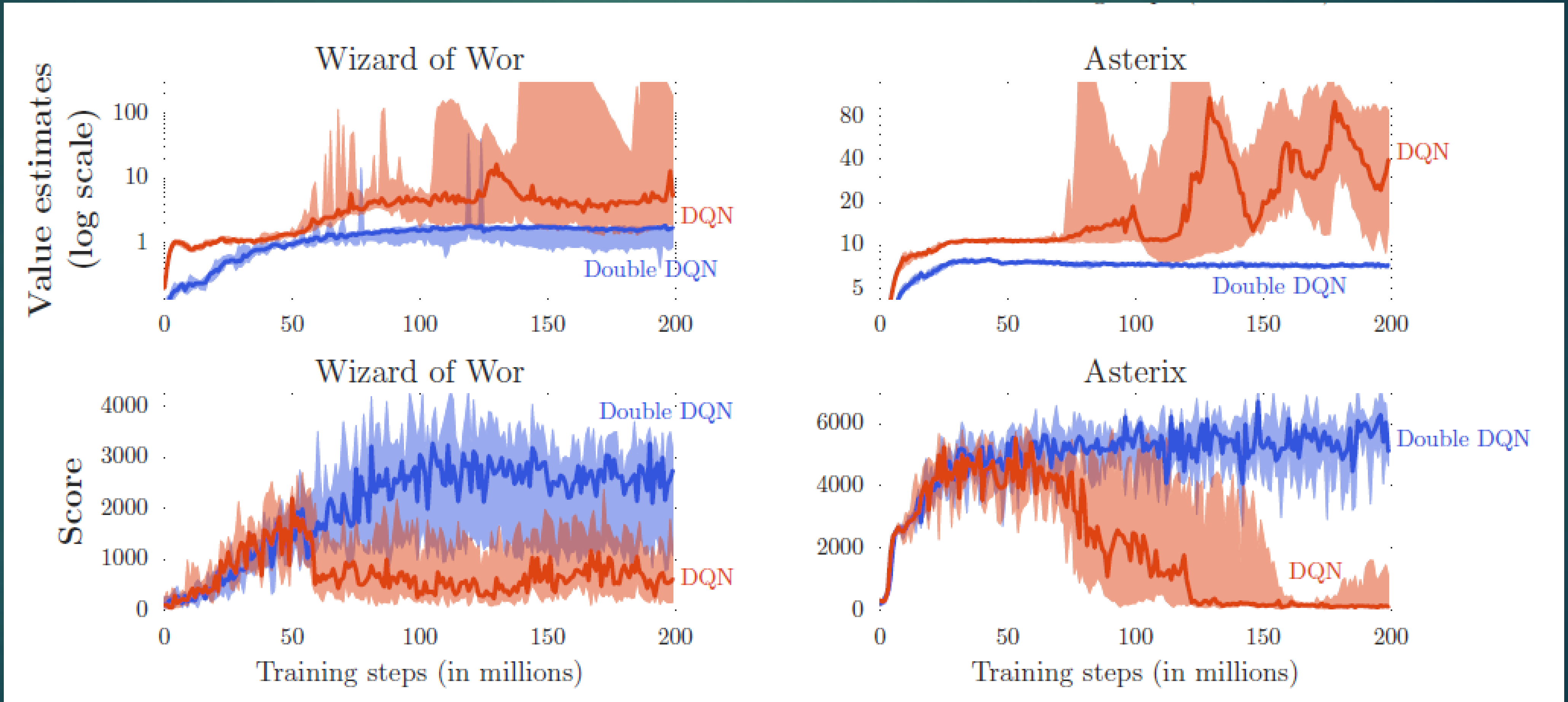
Results on overoptimism

$T = 125,000$ steps as

$$\frac{1}{T}\sum_{t=1}^{T} \operatorname*{argmax}_{a} Q(S_t, a; \boldsymbol{\theta})$$



Results on overoptimism



Results on overoptimism

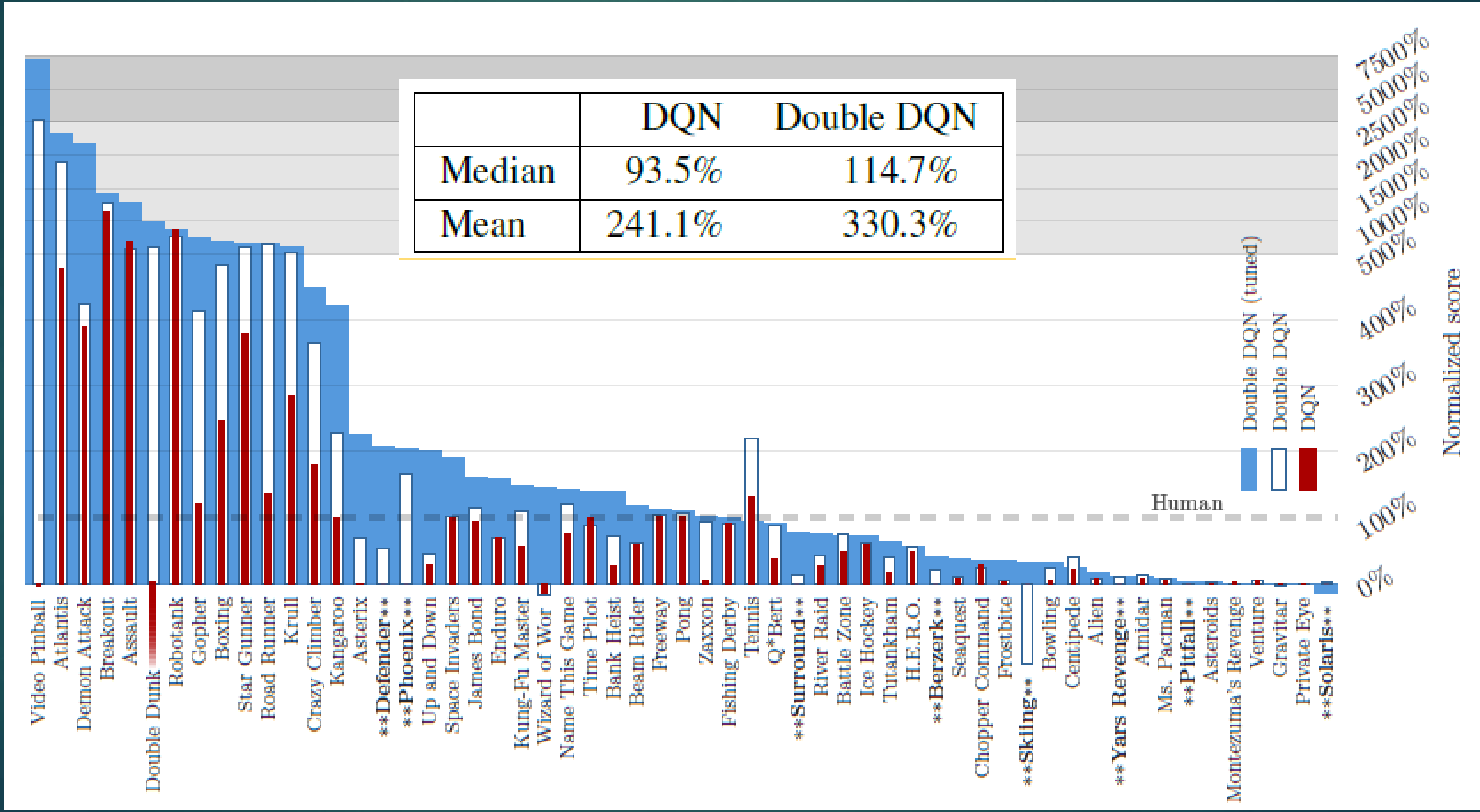
- Obtained results suggest followings:
  - Double DQN produces more accurate value estimates along with better policies
  - Double DQN is much more stable
  - Overestimation substantially harms resulting policy

Quality of the learned policies

- Now, how Double DQN helps in terms of policy quality?
  - learned policies are evaluated for 5 mins of emulator time then the scores are averaged over 100 episodes (policies are learnt by parameters tuned for DQN!)
  - Scores are normalized

$$\text{score}_{\text{normalized}} = \frac{\text{score}_{\text{agent}} - \text{score}_{\text{random}}}{\text{score}_{\text{human}} - \text{score}_{\text{random}}}$$

Quality of the learned policies



	DQN	Double DQN
Median	93.5%	114.7%
Mean	241.1%	330.3%

Robustness to Human starts

- Due to concern of deterministic games with a unique starting point
  - 100 starting points sampled for each game from human experts
  - Only rewards accumulated after the starting points were considered

	DQN	Double DQN	Double DQN (tuned)
Median	47.5%	88.4%	116.7%
Mean	122.0%	273.1%	475.2%

Pseudo-code

1. Initialize replay memory capacity.
2. Initialize the policy network with random weights.
3. Clone the policy network, and call it the *target network*.
4. *For each episode:*
    1. Initialize the starting state.
    2. *For each time step:*
        1. Select an action.
            - *Via exploration or exploitation*
        2. Execute selected action in an emulator.
        3. Observe reward and next state.
        4. Store experience in replay memory.
        5. Sample random batch from replay memory.
        6. Preprocess states from batch.
        7. Pass batch of preprocessed states to policy network.
        8. Calculate loss between output Q-values and target Q-values.
            - Requires a pass to the target network for the next state
        9. Gradient descent updates weights in the policy network to minimize loss.
            - After x time steps, weights in the target network are updated to the weights in the policy network.

Implementation

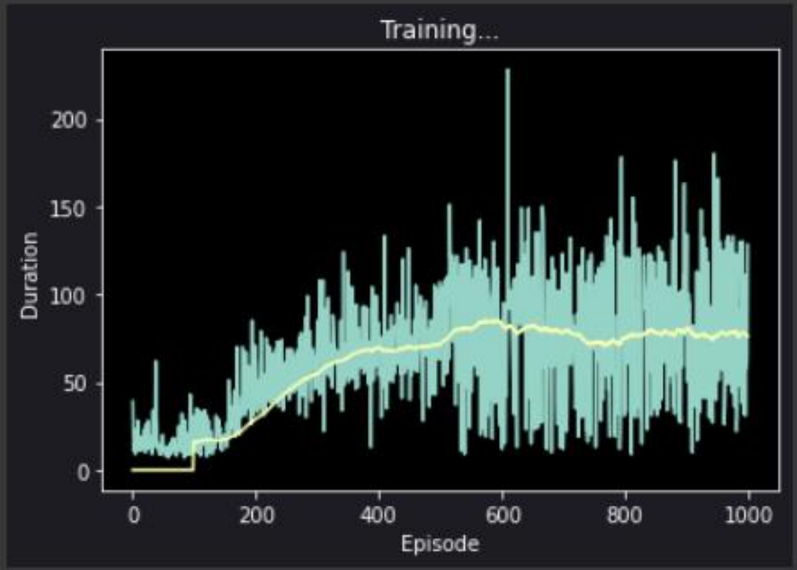
- Fully-Connected layers:
  - 4 hidden layers:

```
def __init__(self, img_height, img_width):
    super().__init__()

    self.fc1 = nn.Linear(in_features=img_height*img_width*3, out_features=24)
    self.fc2 = nn.Linear(in_features=24, out_features=32)
    self.fc3 = nn.Linear(in_features=32, out_features=48)
    self.fc4 = nn.Linear(in_features=48, out_features=16)
    self.out = nn.Linear(in_features=16, out_features=2)
def forward(self, t):
    t = t.flatten(start_dim=1)
    t = F.relu(self.fc1(t))
    t = F.relu(self.fc2(t))
    t = F.relu(self.fc3(t))
    t = F.relu(self.fc4(t))
    t = self.out(t)
```

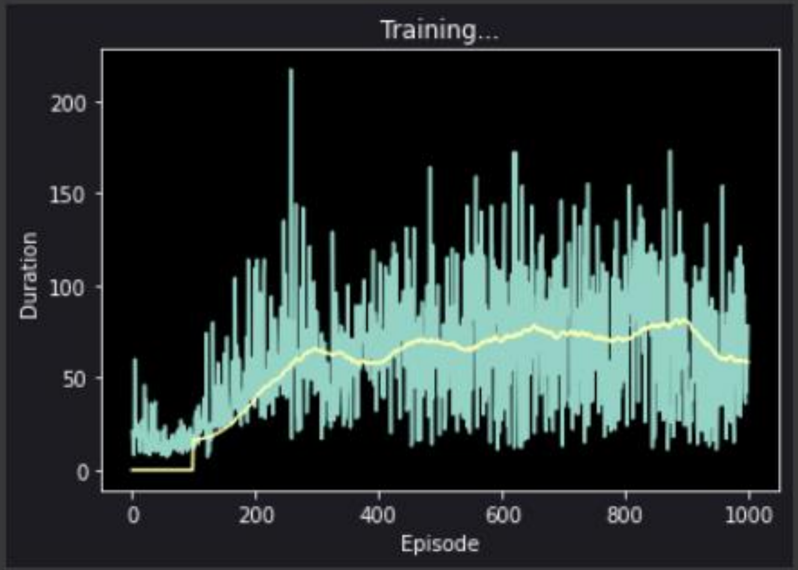
```
batch_size = 256
gamma = 0.999
eps_start = 1
eps_end = 0.01
eps_decay = 0.001
target_update = 20
target_update2 = 10
memory_size = 100000
lr = 0.001
num_episodes = 1000
```

Implementation

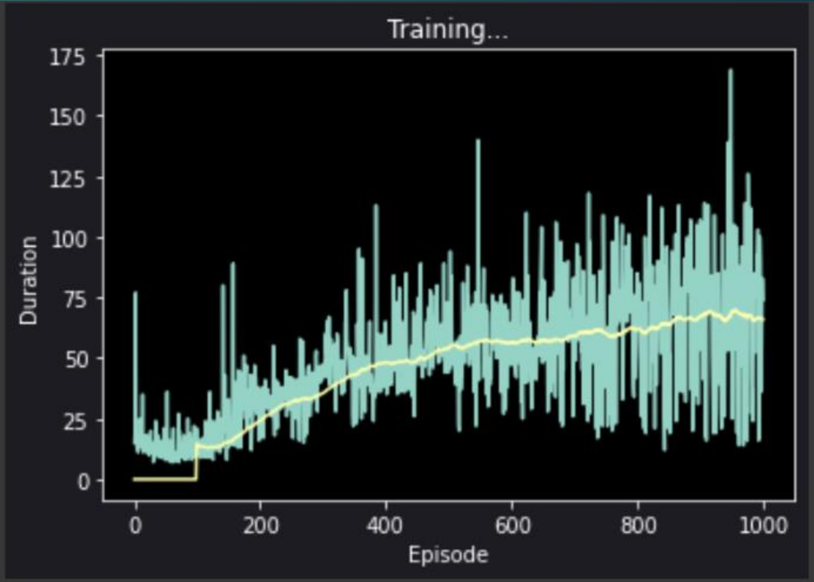
- Fully-Connected layers:
  - 4 hidden layers:




```
Episode 1000
 100 episode moving avg: 76.23
```



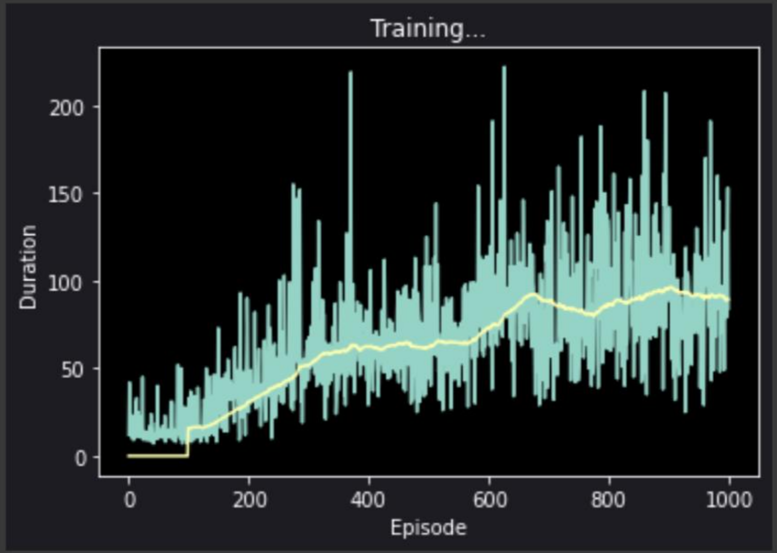

```
Episode 1000
 100 episode moving avg: 58.46
```



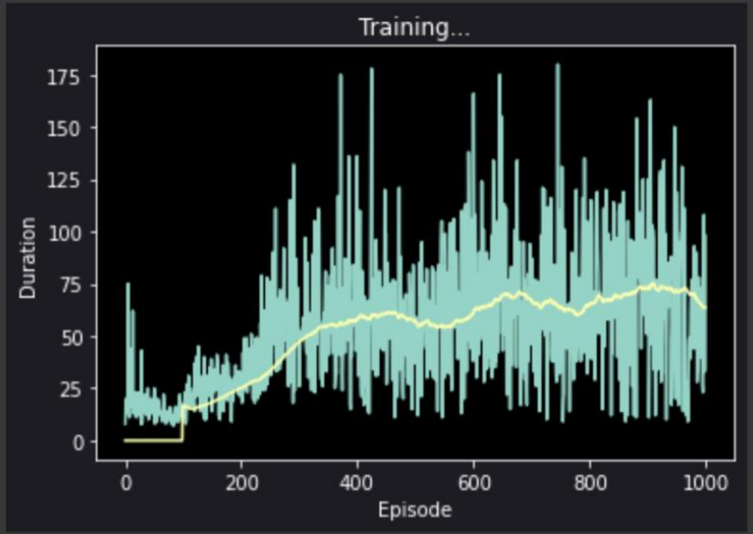

```
Episode 1000
 100 episode moving avg: 65.92
```

Implementation

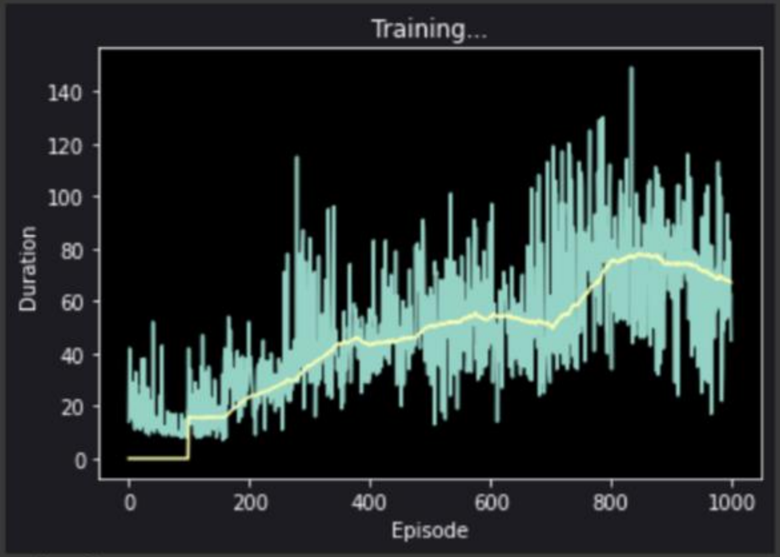
- Fully-Connected layers:
  - 2 hidden layers:




```
Episode 1000
 100 episode moving avg: 89.4
```




```
Episode 1000
 100 episode moving avg: 63.71
```




```
Episode 1000
 100 episode moving avg: 67.1
```

Implementation

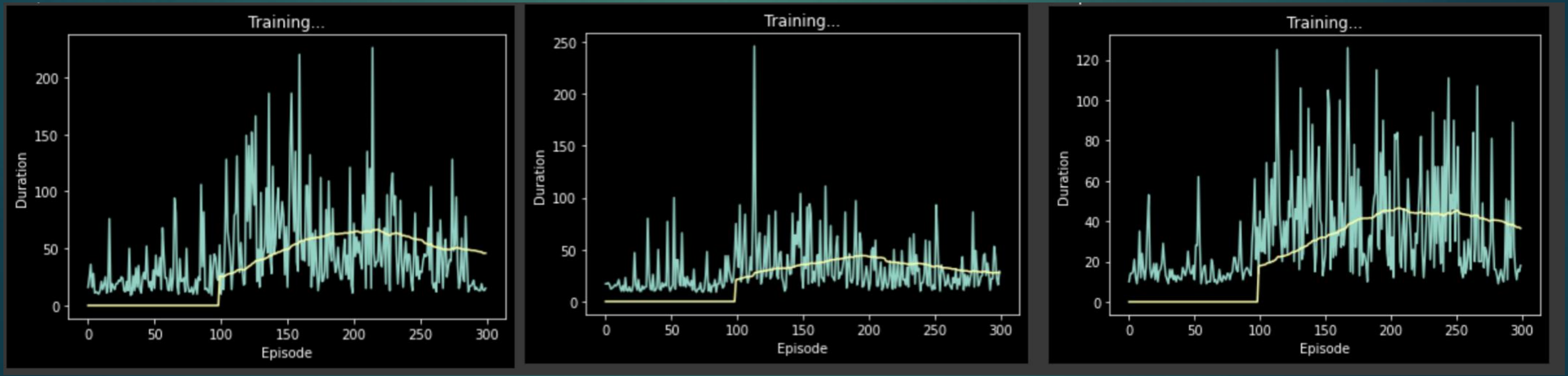
- Convolutional layers:

```
def __init__(self, h, w, outputs):
    super(DQN, self).__init__()
    self.conv1 = nn.Conv2d(3, 16, kernel_size=5, stride=2)
    self.bn1 = nn.BatchNorm2d(16)
    self.conv2 = nn.Conv2d(16, 32, kernel_size=5, stride=2)
    self.bn2 = nn.BatchNorm2d(32)
    self.conv3 = nn.Conv2d(32, 32, kernel_size=5, stride=2)
    self.bn3 = nn.BatchNorm2d(32)

    # Number of Linear input connections depends on output of conv2d layers
    # and therefore the input image size, so compute it.
    def conv2d_size_out(size, kernel_size = 5, stride = 2):
        return (size - (kernel_size - 1) - 1) // stride  + 1
    convw = conv2d_size_out(conv2d_size_out(conv2d_size_out(w)))
    convh = conv2d_size_out(conv2d_size_out(conv2d_size_out(h)))
    linear_input_size = convw * convh * 32
    self.head = nn.Linear(linear_input_size, outputs)

# Called with either one element to determine next action, or a batch
# during optimization. Returns tensor([[left0exp,right0exp]...]).
def forward(self, x):
    x = x.to(device)
    x = F.relu(self.bn1(self.conv1(x)))
    x = F.relu(self.bn2(self.conv2(x)))
    x = F.relu(self.bn3(self.conv3(x)))
    return self.head(x.view(x.size(0), -1))
```

```
BATCH_SIZE = 128
GAMMA = 0.999
EPS_START = 0.9
EPS_END = 0.05
EPS_DECAY = 200
TARGET_UPDATE = 10
TARGET_UPDATE2 = 5
```

Implementation

- Convolutional layers:



Future Works:

- Deployment of an intermediate network similar to Target network for action selection

$$Y_t^{\text{TripleDQN}} \equiv R_{t+1} + \gamma Q(S_{t+1}, \operatorname{argmax} Q(S_{t+1}, a; \boldsymbol{\theta}^{+}{}_{t}), \boldsymbol{\theta}_t^{-})$$

$$\boldsymbol{\theta} \quad \rightarrow \quad \boldsymbol{\theta}_t^{+} \quad \rightarrow \quad \boldsymbol{\theta}_t^{-}$$

- Similar to Double Q-network, define two set of weights and networks to further tackle overestimation


THANK YOU
FOR YOUR ATTENTION
fotolia