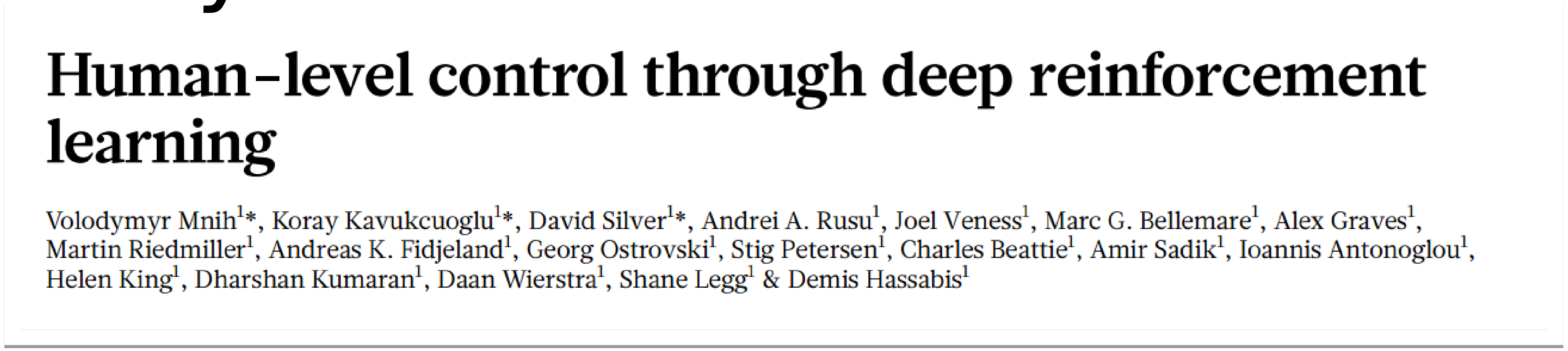
DQN Walkthrough

Dan Neil

16 April 2015

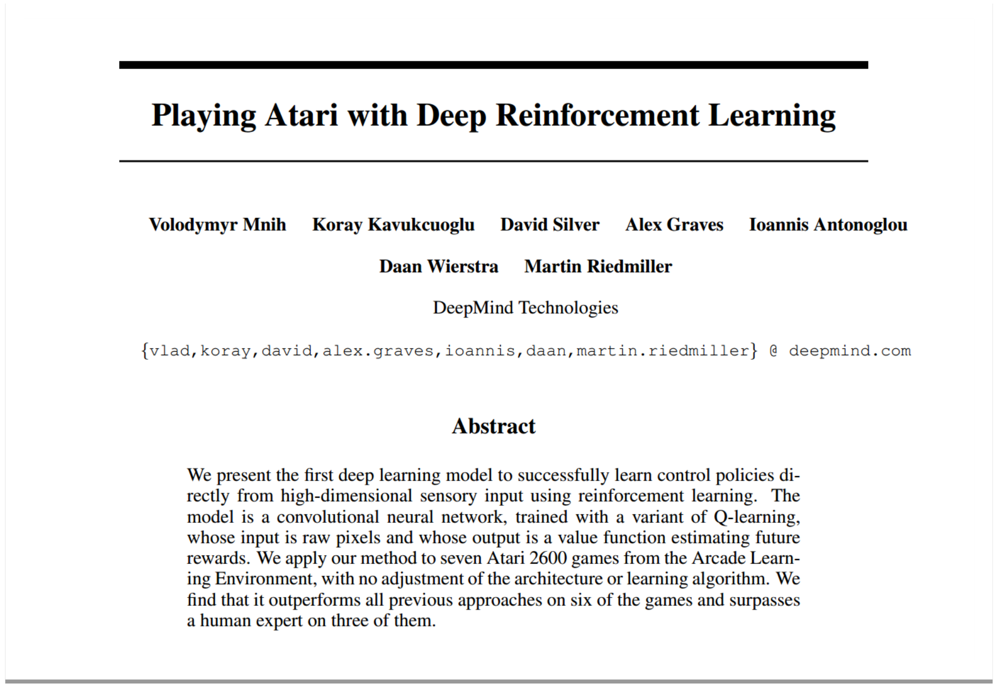
Quick Paper Overview

History



Human-level control through deep reinforcement learning

Volodymyr Mnih[1]*, Koray Kavukcuoglu[1]*, David Silver[1]*, Andrei A. Rusu[1], Joel Veness[1], Marc G. Bellemare[1], Alex Graves[1], Martin Riedmiller[1], Andreas K. Fidjeland[1], Georg Ostrovski[1], Stig Petersen[1], Charles Beattie[1], Amir Sadik[1], Ioannis Antonoglou[1], Helen King[1], Dharshan Kumaran[1], Daan Wierstra[1], Shane Legg[1] & Demis Hassabis[1]



Playing Atari with Deep Reinforcement Learning

Volodymyr Mnih Koray Kavukcuoglu David Silver Alex Graves Ioannis Antonoglou

Daan Wierstra Martin Riedmiller

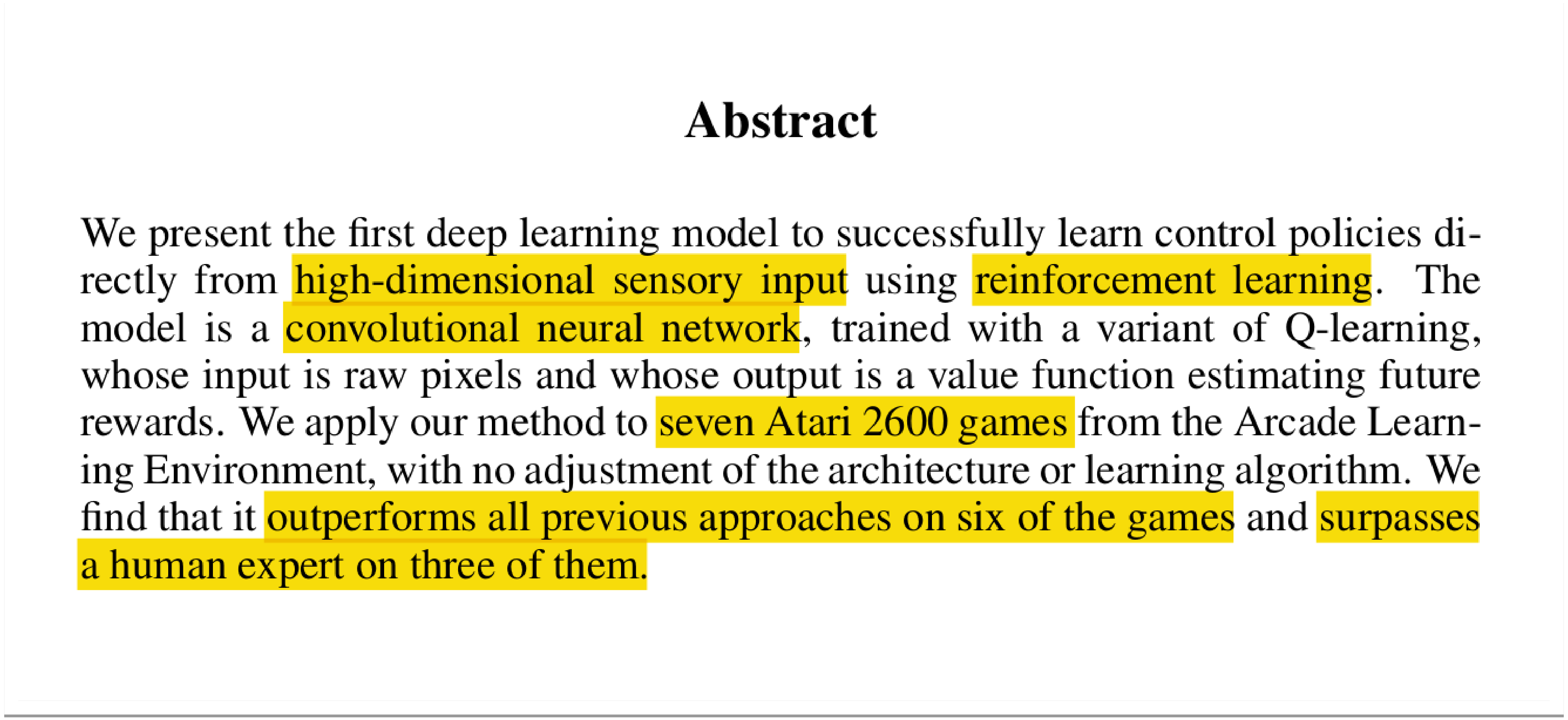
DeepMind Technologies

{vlad,koray,david,alex.graves,ioannis,daan,martin.riedmiller} @ deepmind.com

Abstract

We present the first deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. The model is a convolutional neural network, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards. We apply our method to seven Atari 2600 games from the Arcade Learning Environment, with no adjustment of the architecture or learning algorithm. We find that it outperforms all previous approaches on six of the games and surpasses a human expert on three of them.

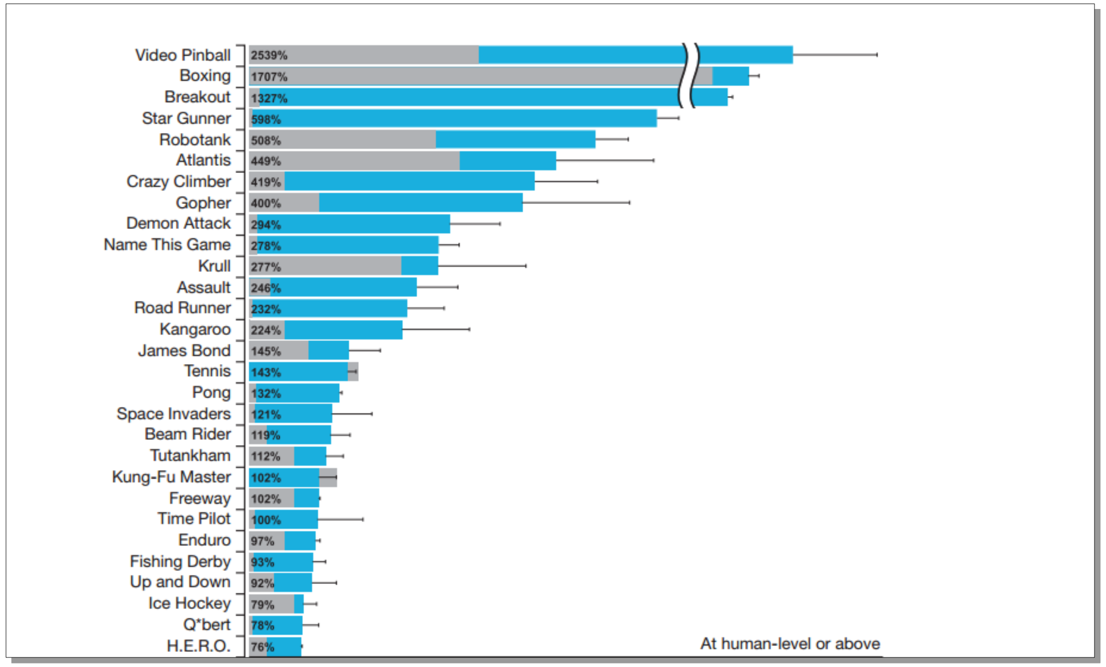
Abstract (Following from Jakob Buhmann)

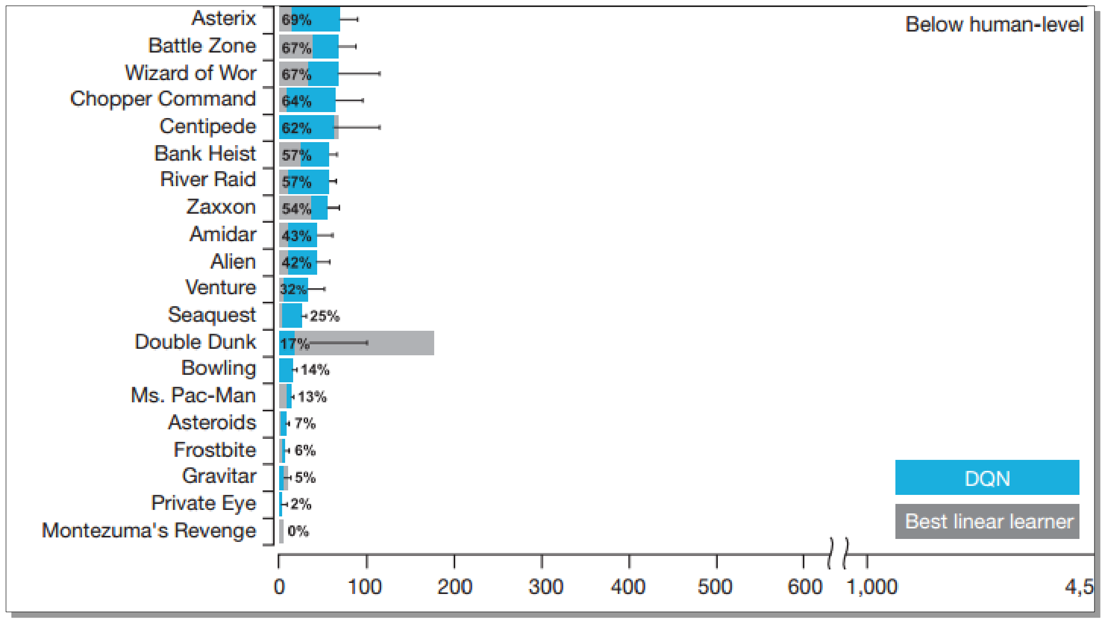


Abstract

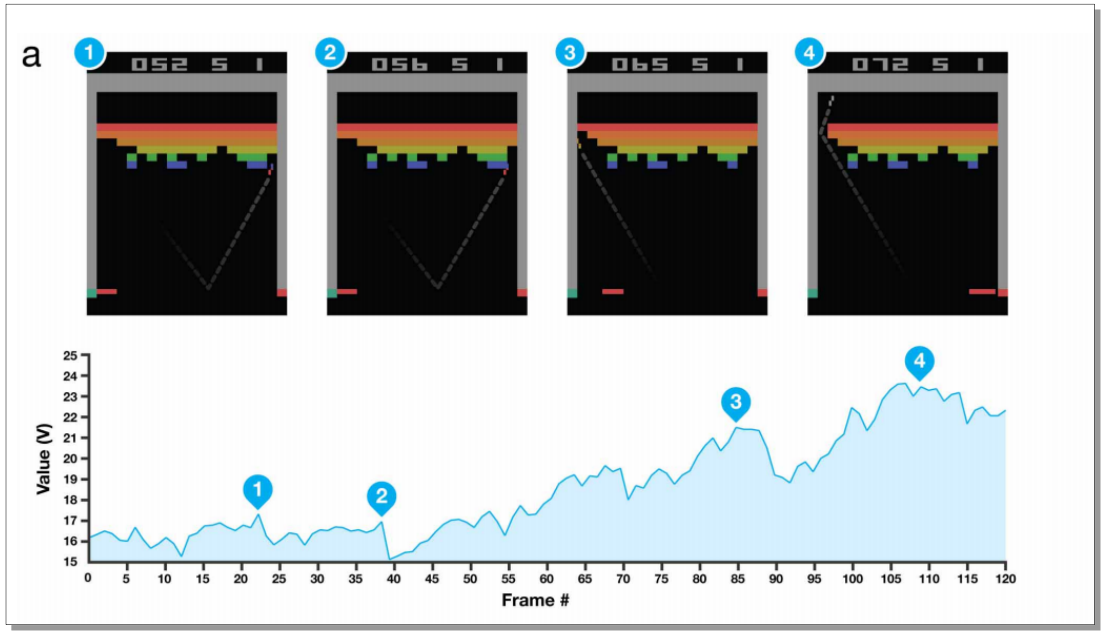
We present the first deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. The model is a convolutional neural network, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards. We apply our method to seven Atari 2600 games from the Arcade Learning Environment, with no adjustment of the architecture or learning algorithm. We find that it outperforms all previous approaches on six of the games and surpasses a human expert on three of them.

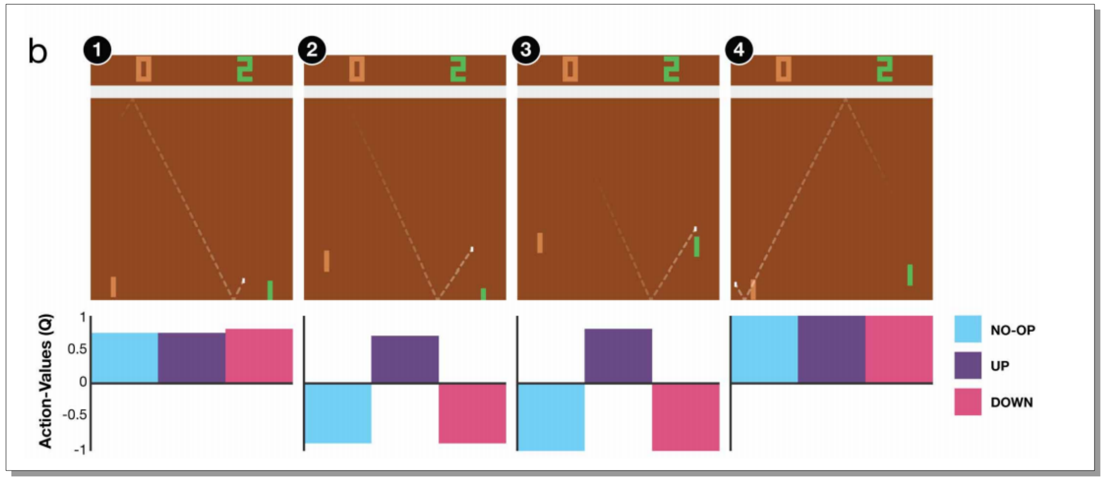
Why DQNs?


Video Pinball 2539%
Boxing 1707%
Breakout 1327%
Star Gunner 598%
Robotank 508%
Atlantis 449%
Crazy Climber 419%
Gopher 400%
Demon Attack 294%
Name This Game 278%
Krull 277%
Assault 246%
Road Runner 232%
Kangaroo 224%
James Bond 145%
Tennis 143%
Pong 132%
Space Invaders 121%
Beam Rider 119%
Tutankham 112%
Kung-Fu Master 102%
Freeway 102%
Time Pilot 100%
Enduro 97%
Fishing Derby 93%
Up and Down 92%
Ice Hockey 79%
Q*bert 78%
H.E.R.O. 76%
At human-level or above


Asterix 69%
Battle Zone 67%
Wizard of Wor 67%
Chopper Command 64%
Centipede 62%
Bank Heist 57%
River Raid 57%
Zaxxon 54%
Amidar 43%
Alien 42%
Venture 32%
Seaquest 25%
Double Dunk 17%
Bowling 14%
Ms. Pac-Man 13%
Asteroids 7%
Frostbite 6%
Gravitar 5%
Private Eye 2%
Montezuma's Revenge 0%
Below human-level
DQN
Best linear learner
0
100
200
300
400
500
600
1,000
4,5

Show it working





Reinforcement Learning Quick Recap

- Actor chooses an action
- Action is performed and influences the environment
- Based on the new state, the system receives a reward r_t
- System should learn to choose the action that maximizes the expected discounted reward
- Challenge is to learn in spite of sparse, noisy, or temporally shifted rewards

Reinforcement Learning

- Q-Learning:
  - State represented by a sequence $s_t = x_1, a_1, x_2, ..., a_t, x_t,$
  - Approximate action value function $Q(s, a)$
  - Find maximizing action $Q^*(s, a) = \max_a \mathbb{E}[R_t | s_t = s, a_t = a]$
  - Optimal solution fulfills Bellman equation

$$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q^*(s', a') | s, a \right]$$

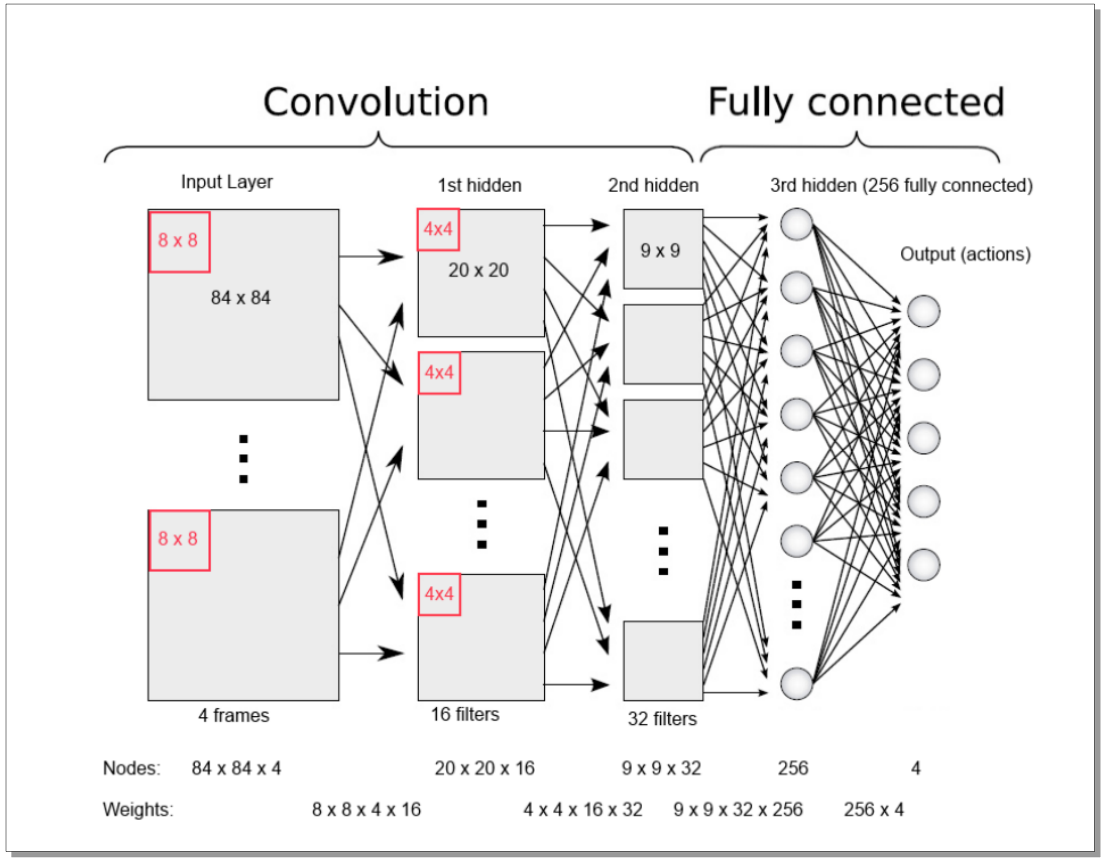
  - Approximate iteratively, s.t. $Q_i \rightarrow Q^* \ , \ i \rightarrow \infty$

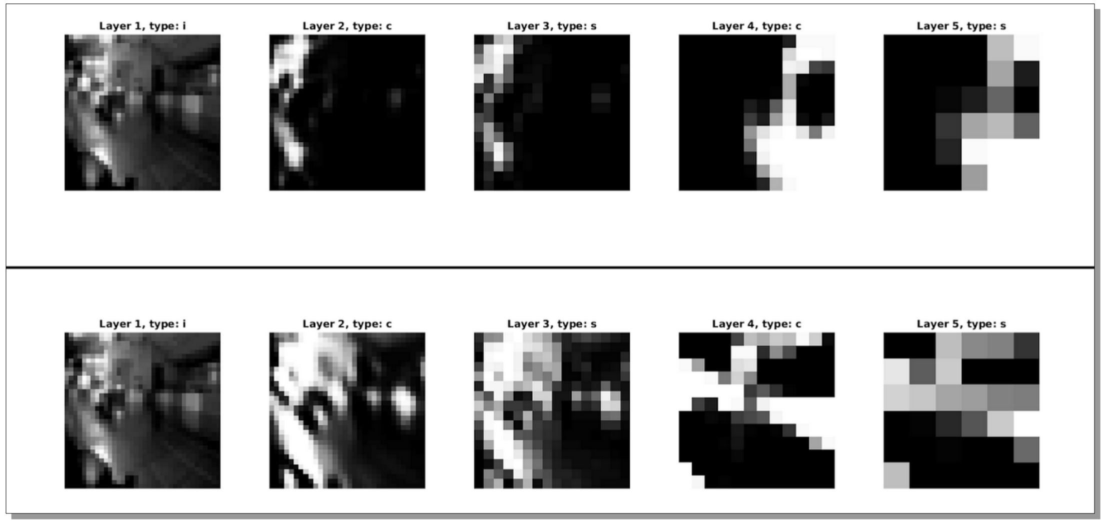
$$Q_{i+1}(s, a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q_i(s', a') | s, a \right]$$

Why function approximation?

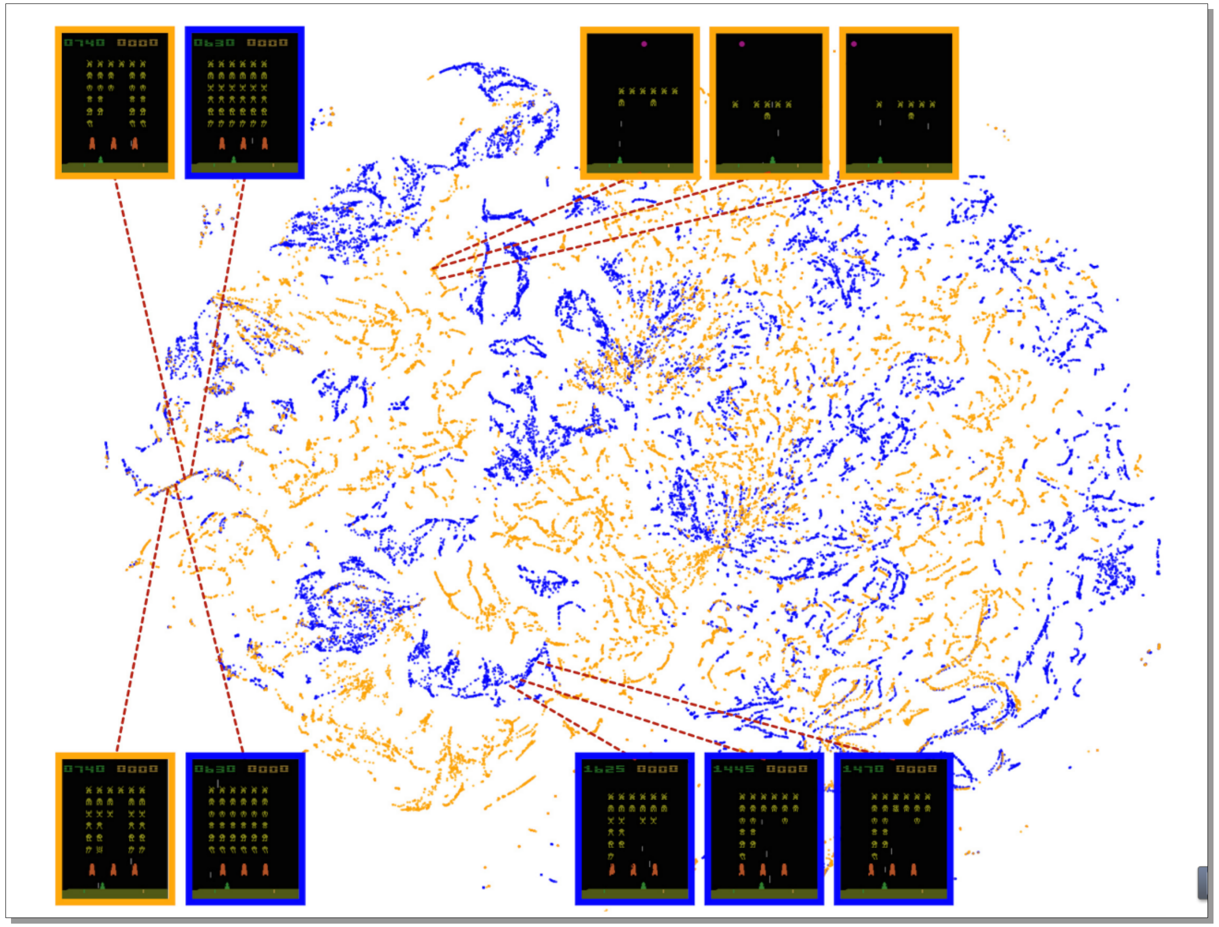
- Learn a mapping of states to true rewards
    - Sounds like neural networks (model-free generic function approximators)
- Solution: neural networks
    - Fuzzy mapping from arbitrary features to rewards
    - Many tools exist to optimize learning

Convolutional Networks





Convolutional Network Embedding



Challenges in NN as Function Approximators

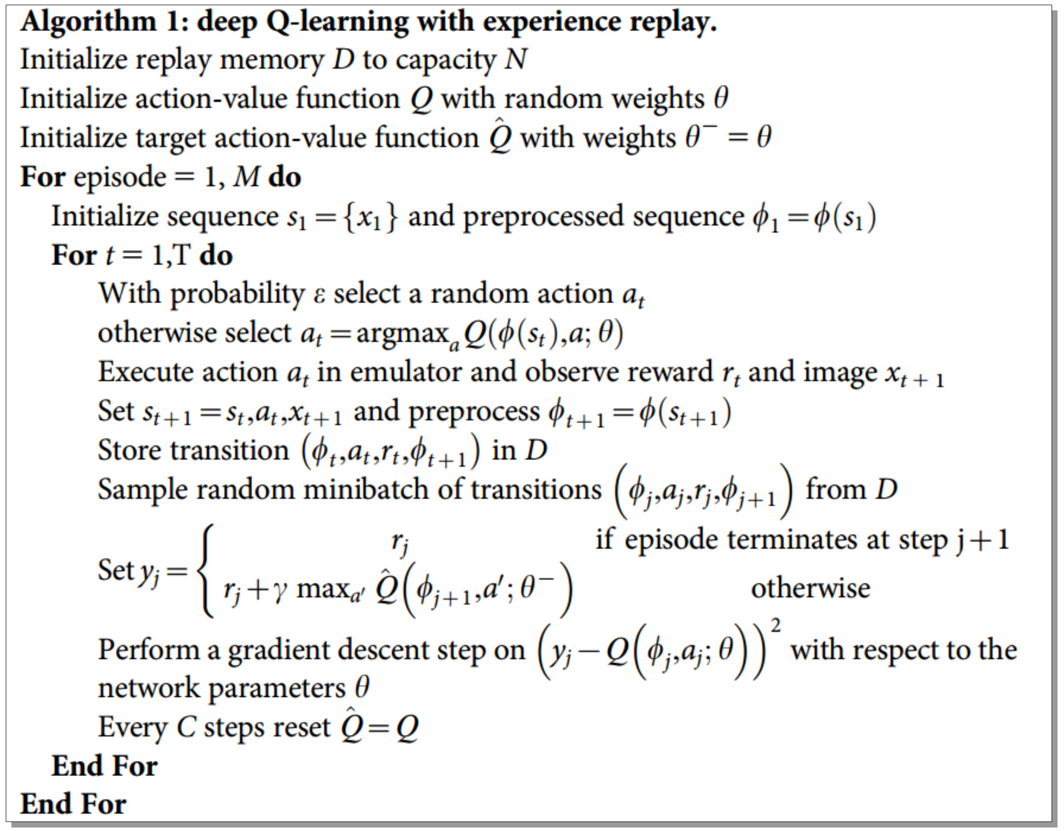
- Correlations in successive states
- Small changes to Q-estimator may significantly change the policy and the subsequent data distribution
- Strong correlations exist between the action-values and the target values
- ✉Leads to feedback loops and instability. How to solve?
- One previous approach: lots of randomly initialized networks
  - Very time costly

Enter DQNs

- Two (and a half) key ideas:
- Replay memory
  - Store some memories of the past, and randomly sample from these instead of just the current state
  - Each memory can influence training more than once (good, efficient)
  - Random sampling from past breaks consecutive correlations
  - Behaviour distribution is distributed over history
- Target is only updated periodically, allowing the action-value estimator to diverge and reduce correlations
  - No longer does an increase of $Q(s_t, a_t)$ increase $Q(s_{t+1}, a)$, avoiding feedback loop
- Clip the error rate to (-1, 1)

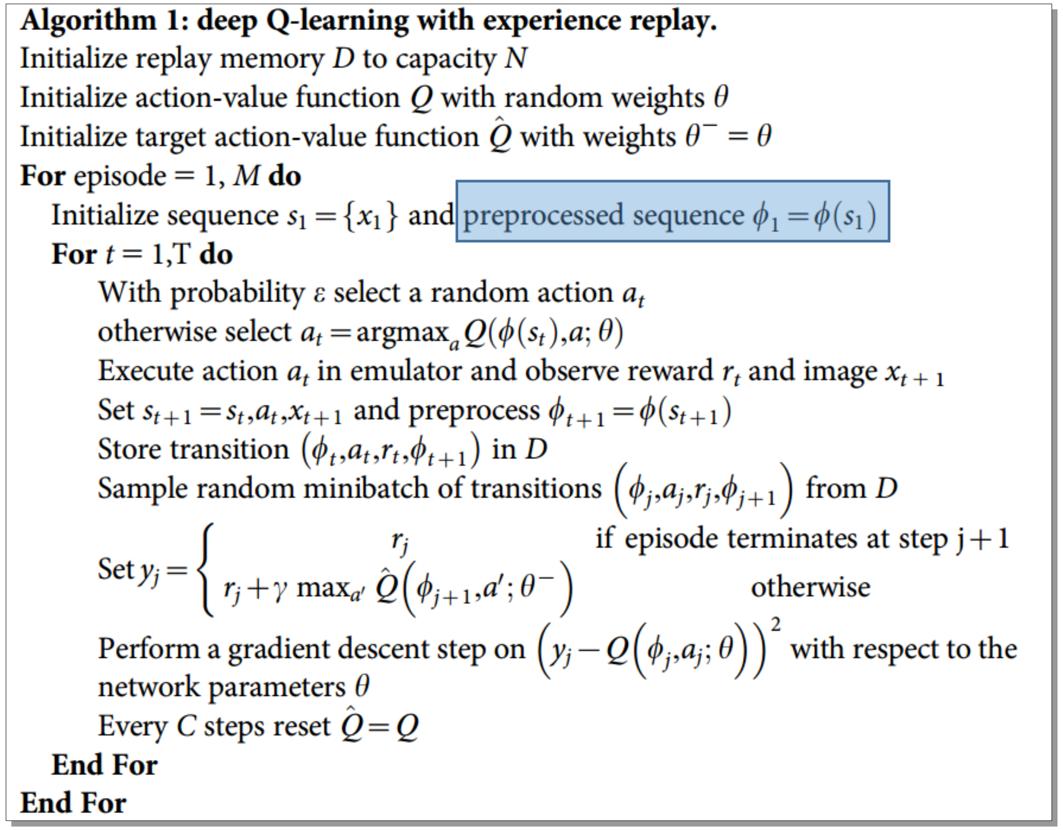
Code Deep Dive

Algorithm Overview

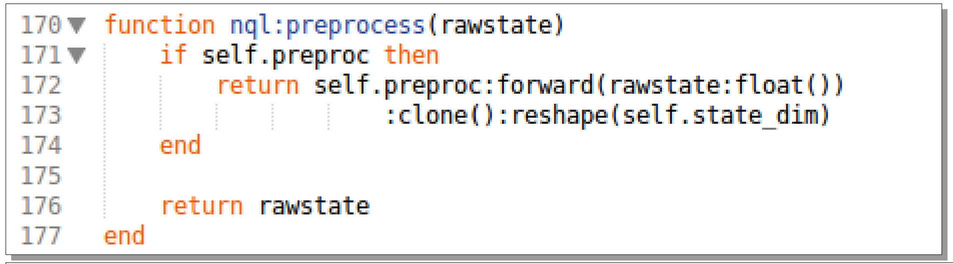


```
**Algorithm 1: deep Q-learning with experience replay.**
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
**For** episode = 1, $M$ **do**
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    **For** $t = 1$,T **do**
        With probability $\varepsilon$ select a random action $a_t$
        otherwise select $a_t = \mathrm{argmax}_a Q(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
        Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the
        network parameters $\theta$
        Every $C$ steps reset $\hat{Q} = Q$
    **End For**
**End For**
```

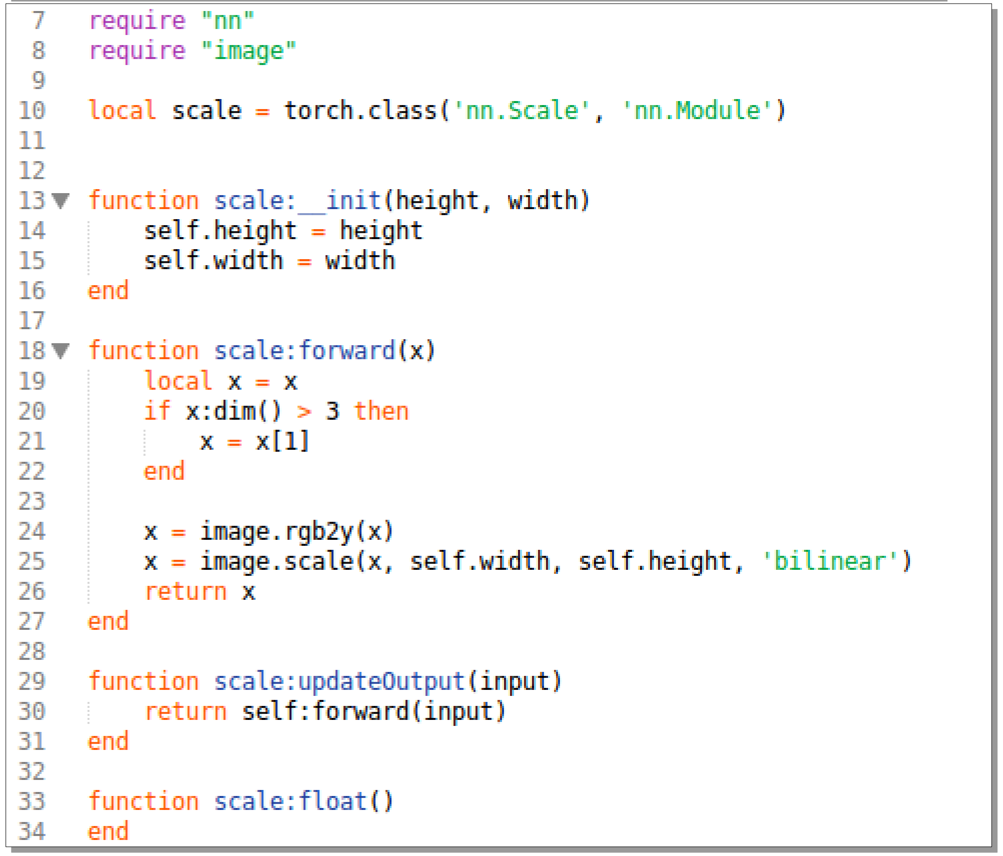
Algorithm Overview - Preprocess



```
Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, $M$ do
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    For $t$ = 1,T do
        With probability $\varepsilon$ select a random action $a_t$
        otherwise select $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
        Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the
        network parameters $\theta$
        Every $C$ steps reset $\hat{Q} = Q$
    End For
End For
```

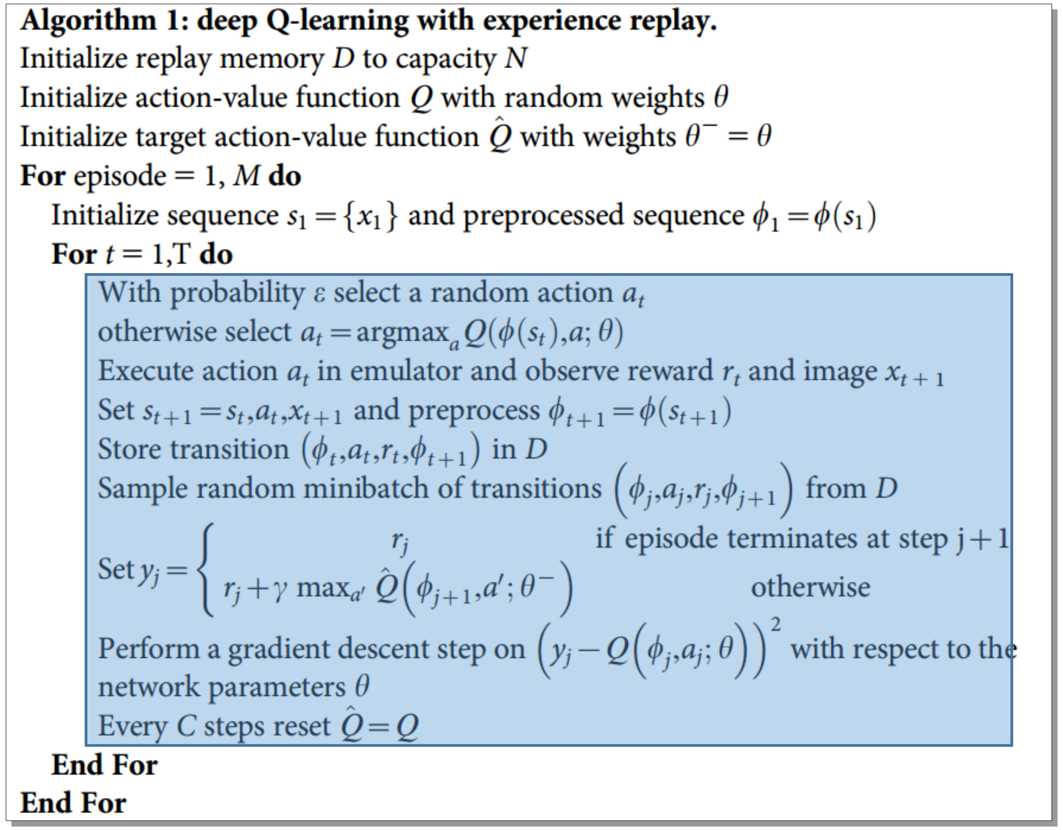


```lua
function nql:preprocess(rawstate)
    if self.preproc then
        return self.preproc:forward(rawstate:float())
                    :clone():reshape(self.state_dim)
    end

    return rawstate
end
```

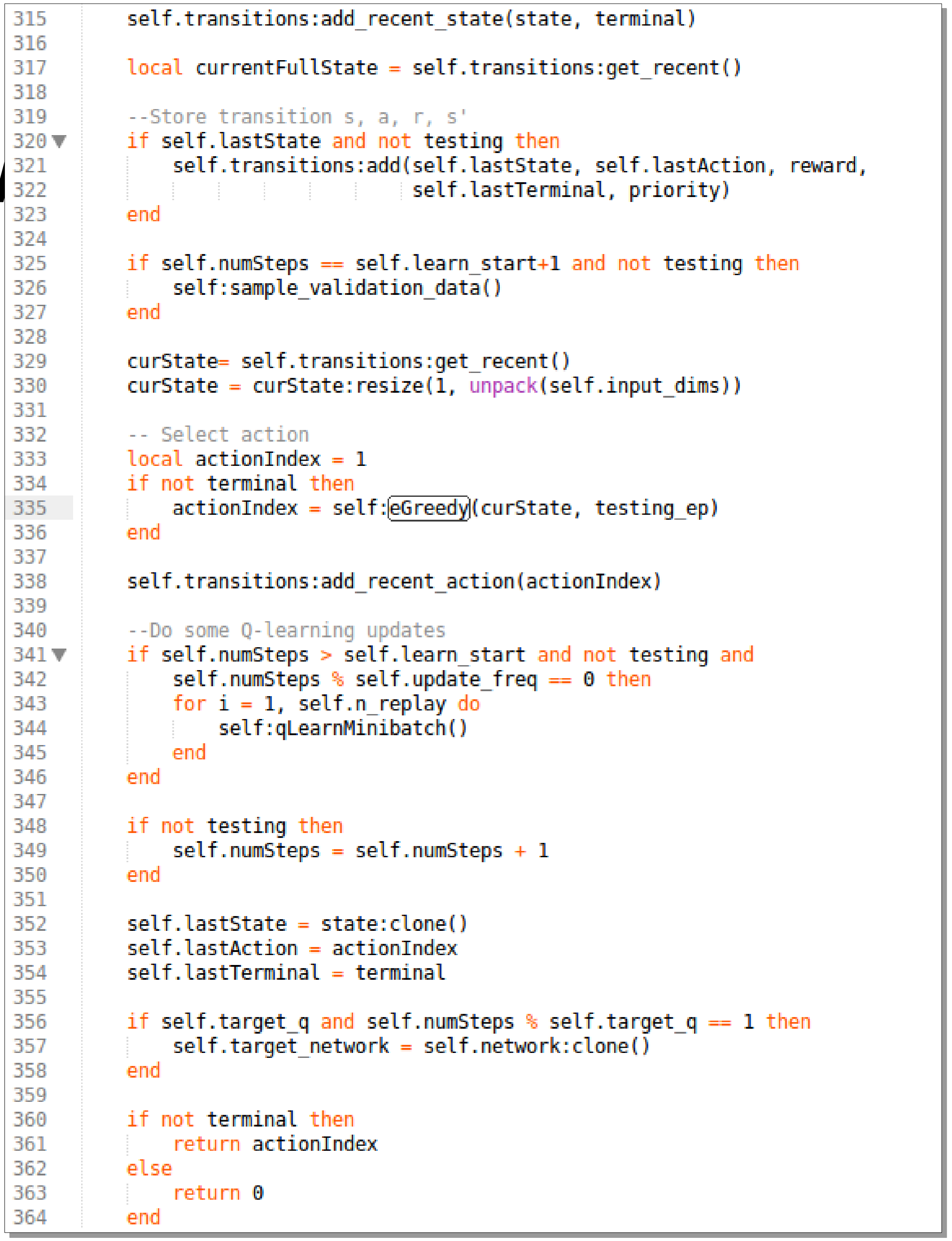


```lua
require "nn"
require "image"

local scale = torch.class('nn.Scale', 'nn.Module')


function scale:__init(height, width)
    self.height = height
    self.width = width
end

function scale:forward(x)
    local x = x
    if x:dim() > 3 then
        x = x[1]
    end

    x = image.rgb2y(x)
    x = image.scale(x, self.width, self.height, 'bilinear')
    return x
end

function scale:updateOutput(input)
    return self:forward(input)
end

function scale:float()
end
```

Algorithm Overview

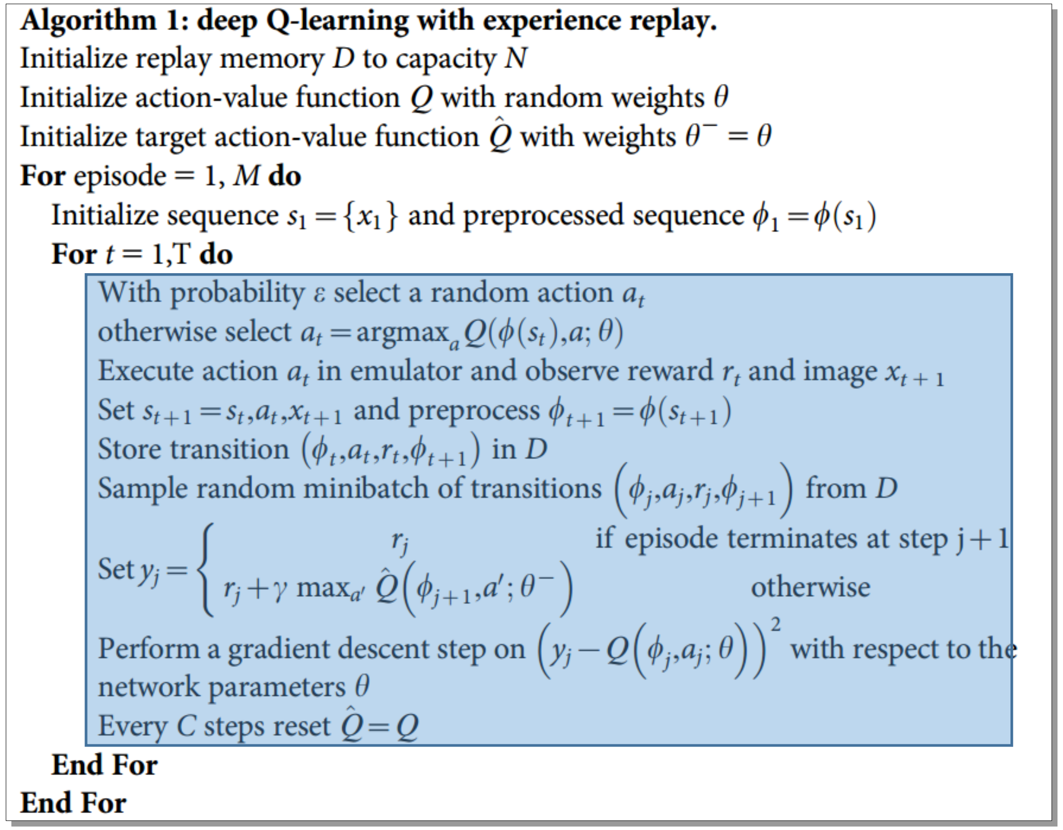


```
Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, $M$ do
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    For $t$ = 1,T do
        With probability $\varepsilon$ select a random action $a_t$
        otherwise select $a_t = \mathrm{argmax}_a Q(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
        Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
        Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the
        network parameters $\theta$
        Every $C$ steps reset $\hat{Q} = Q$
    End For
End For
```

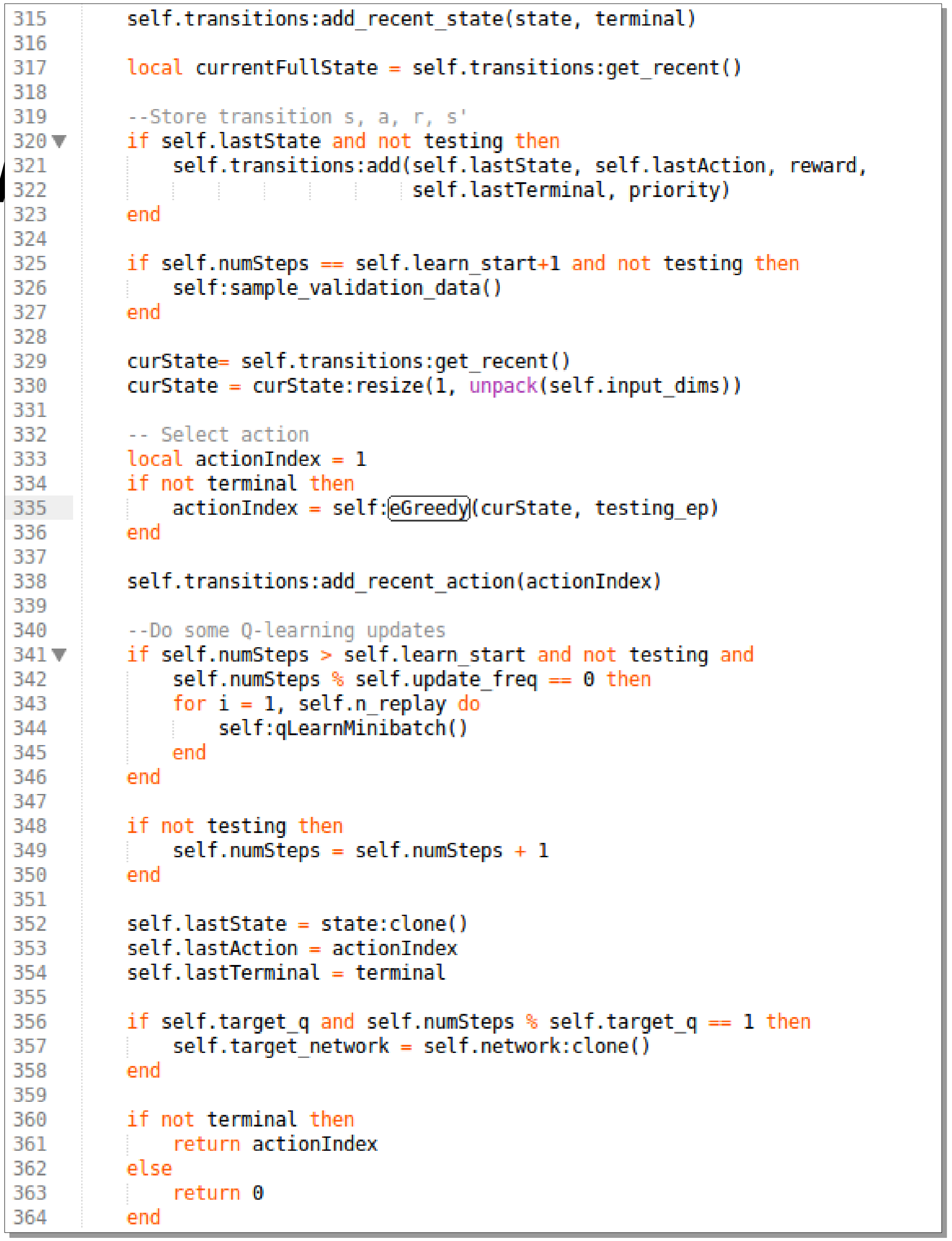


```lua
    self.transitions:add_recent_state(state, terminal)

    local currentFullState = self.transitions:get_recent()

    --Store transition s, a, r, s'
    if self.lastState and not testing then
        self.transitions:add(self.lastState, self.lastAction, reward,
                             self.lastTerminal, priority)
    end

    if self.numSteps == self.learn_start+1 and not testing then
        self:sample_validation_data()
    end

    curState= self.transitions:get_recent()
    curState = curState:resize(1, unpack(self.input_dims))

    -- Select action
    local actionIndex = 1
    if not terminal then
        actionIndex = self:eGreedy(curState, testing_ep)
    end

    self.transitions:add_recent_action(actionIndex)

    --Do some Q-learning updates
    if self.numSteps > self.learn_start and not testing and
        self.numSteps % self.update_freq == 0 then
        for i = 1, self.n_replay do
            self:qLearnMinibatch()
        end
    end

    if not testing then
        self.numSteps = self.numSteps + 1
    end

    self.lastState = state:clone()
    self.lastAction = actionIndex
    self.lastTerminal = terminal

    if self.target_q and self.numSteps % self.target_q == 1 then
        self.target_network = self.network:clone()
    end

    if not terminal then
        return actionIndex
    else
        return 0
    end
```

Algorithm Overview

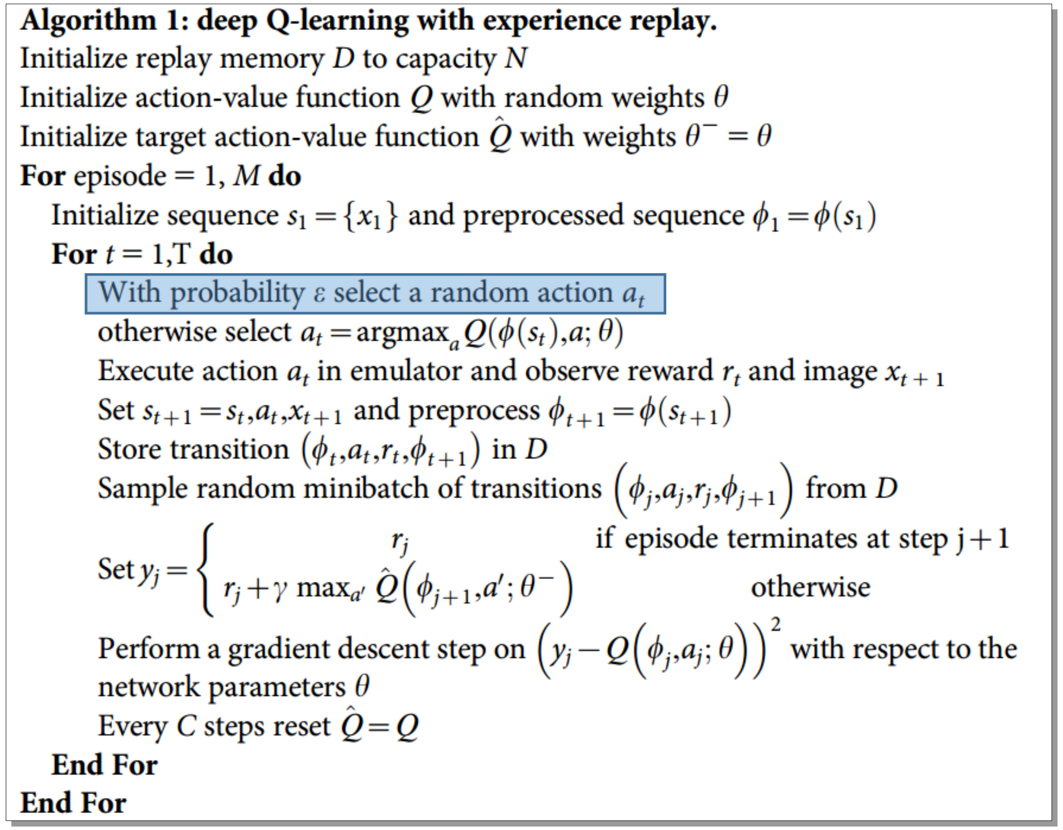


```
Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, $M$ do
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    For $t$ = 1,T do
        With probability $\varepsilon$ select a random action $a_t$
        otherwise select $a_t = \mathrm{argmax}_a Q(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
        Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the
        network parameters $\theta$
        Every $C$ steps reset $\hat{Q} = Q$
    End For
End For
```

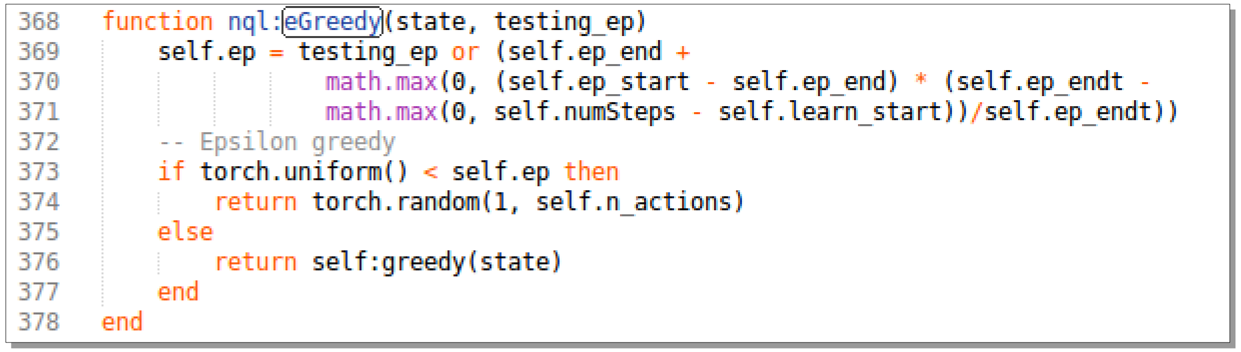


```lua
    self.transitions:add_recent_state(state, terminal)

    local currentFullState = self.transitions:get_recent()

    --Store transition s, a, r, s'
    if self.lastState and not testing then
        self.transitions:add(self.lastState, self.lastAction, reward,
                             self.lastTerminal, priority)
    end

    if self.numSteps == self.learn_start+1 and not testing then
        self:sample_validation_data()
    end

    curState= self.transitions:get_recent()
    curState = curState:resize(1, unpack(self.input_dims))

    -- Select action
    local actionIndex = 1
    if not terminal then
        actionIndex = self:eGreedy(curState, testing_ep)
    end

    self.transitions:add_recent_action(actionIndex)

    --Do some Q-learning updates
    if self.numSteps > self.learn_start and not testing and
        self.numSteps % self.update_freq == 0 then
        for i = 1, self.n_replay do
            self:qLearnMinibatch()
        end
    end

    if not testing then
        self.numSteps = self.numSteps + 1
    end

    self.lastState = state:clone()
    self.lastAction = actionIndex
    self.lastTerminal = terminal

    if self.target_q and self.numSteps % self.target_q == 1 then
        self.target_network = self.network:clone()
    end

    if not terminal then
        return actionIndex
    else
        return 0
    end
```

Algorithm Overview – Epsilon Greedy

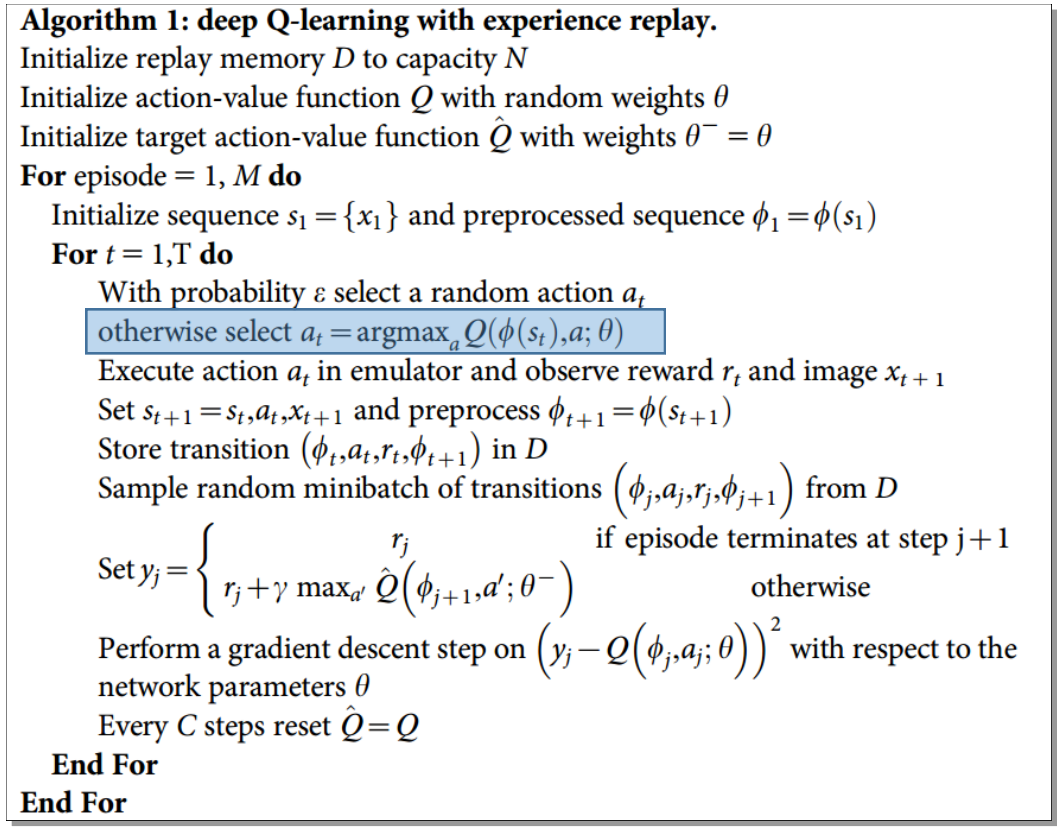


```
Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, $M$ do
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    For $t$ = 1,T do
        With probability $\varepsilon$ select a random action $a_t$
        otherwise select $a_t = \mathrm{argmax}_a Q(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
        Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the
        network parameters $\theta$
        Every $C$ steps reset $\hat{Q} = Q$
    End For
End For
```

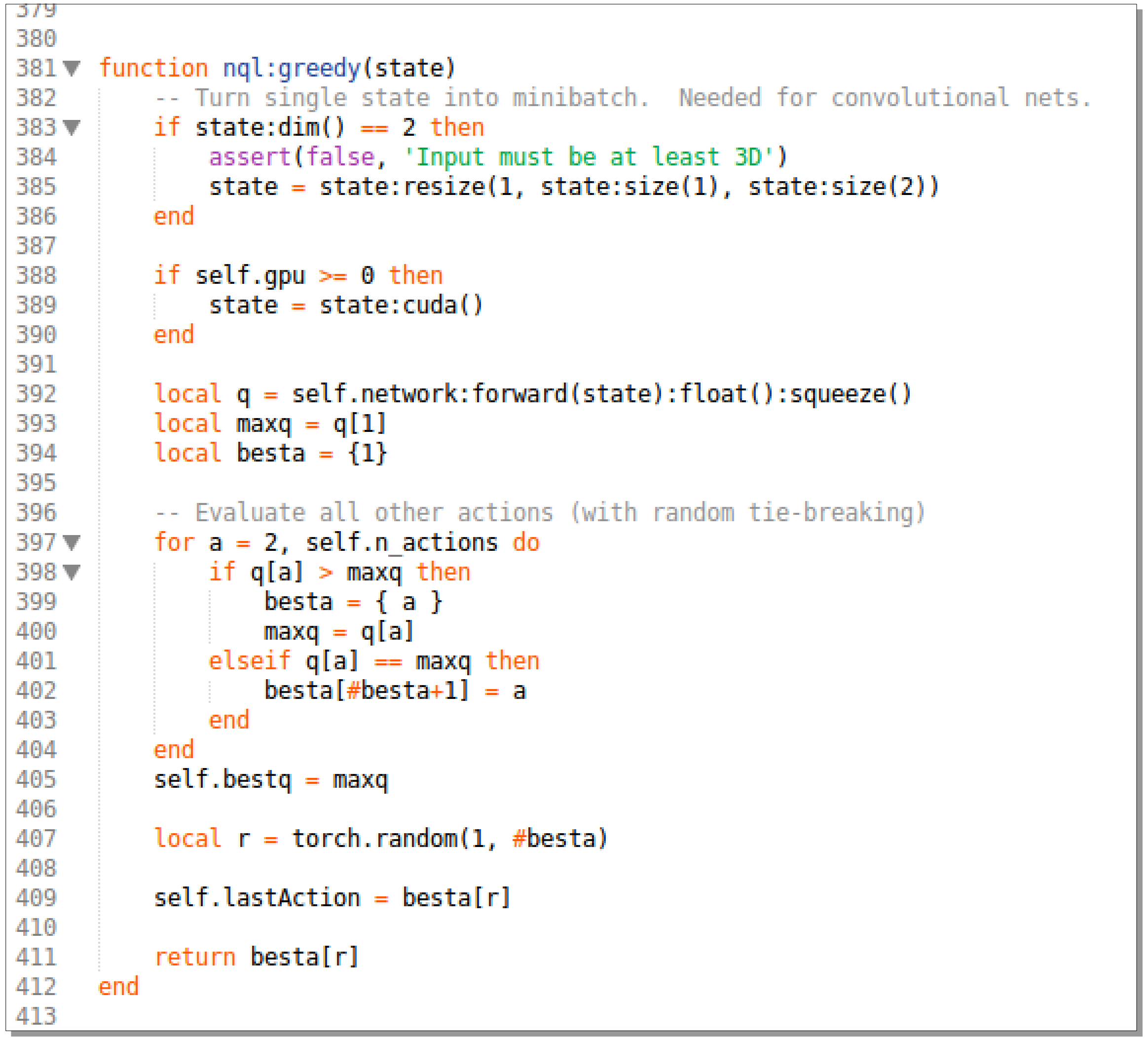


```lua
function nql:eGreedy(state, testing_ep)
    self.ep = testing_ep or (self.ep_end +
                math.max(0, (self.ep_start - self.ep_end) * (self.ep_endt -
                math.max(0, self.numSteps - self.learn_start))/self.ep_endt))
    -- Epsilon greedy
    if torch.uniform() < self.ep then
        return torch.random(1, self.n_actions)
    else
        return self:greedy(state)
    end
end
```

Algorithm Overview – Greedy

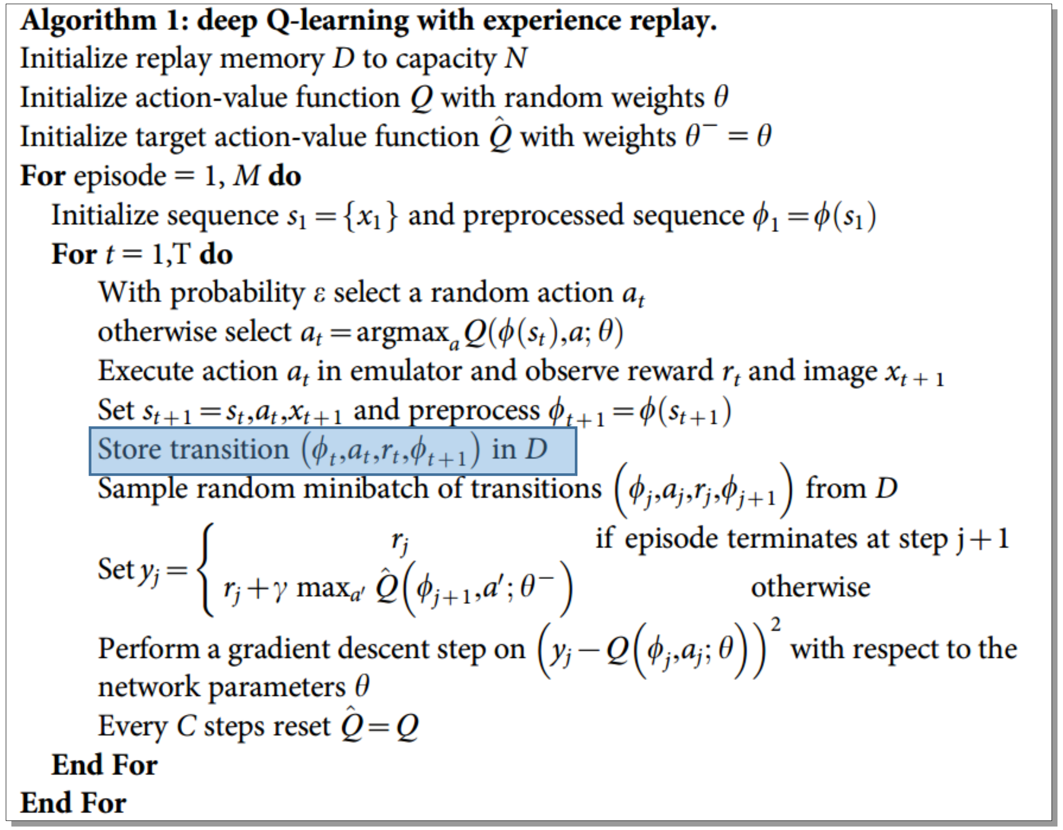


```
Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, $M$ do
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    For $t$ = 1,T do
        With probability $\varepsilon$ select a random action $a_t$
        otherwise select $a_t = \mathrm{argmax}_a Q(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
        Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the
        network parameters $\theta$
        Every $C$ steps reset $\hat{Q} = Q$
    End For
End For
```

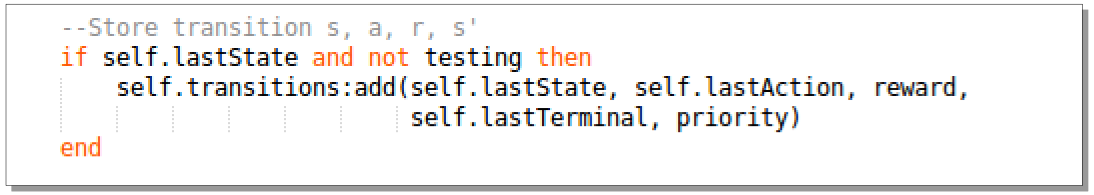


```lua

function nql:greedy(state)
    -- Turn single state into minibatch.  Needed for convolutional nets.
    if state:dim() == 2 then
        assert(false, 'Input must be at least 3D')
        state = state:resize(1, state:size(1), state:size(2))
    end

    if self.gpu >= 0 then
        state = state:cuda()
    end

    local q = self.network:forward(state):float():squeeze()
    local maxq = q[1]
    local besta = {1}

    -- Evaluate all other actions (with random tie-breaking)
    for a = 2, self.n_actions do
        if q[a] > maxq then
            besta = { a }
            maxq = q[a]
        elseif q[a] == maxq then
            besta[#besta+1] = a
        end
    end
    self.bestq = maxq

    local r = torch.random(1, #besta)

    self.lastAction = besta[r]

    return besta[r]
end

```

Algorithm Overview – Store in Replay

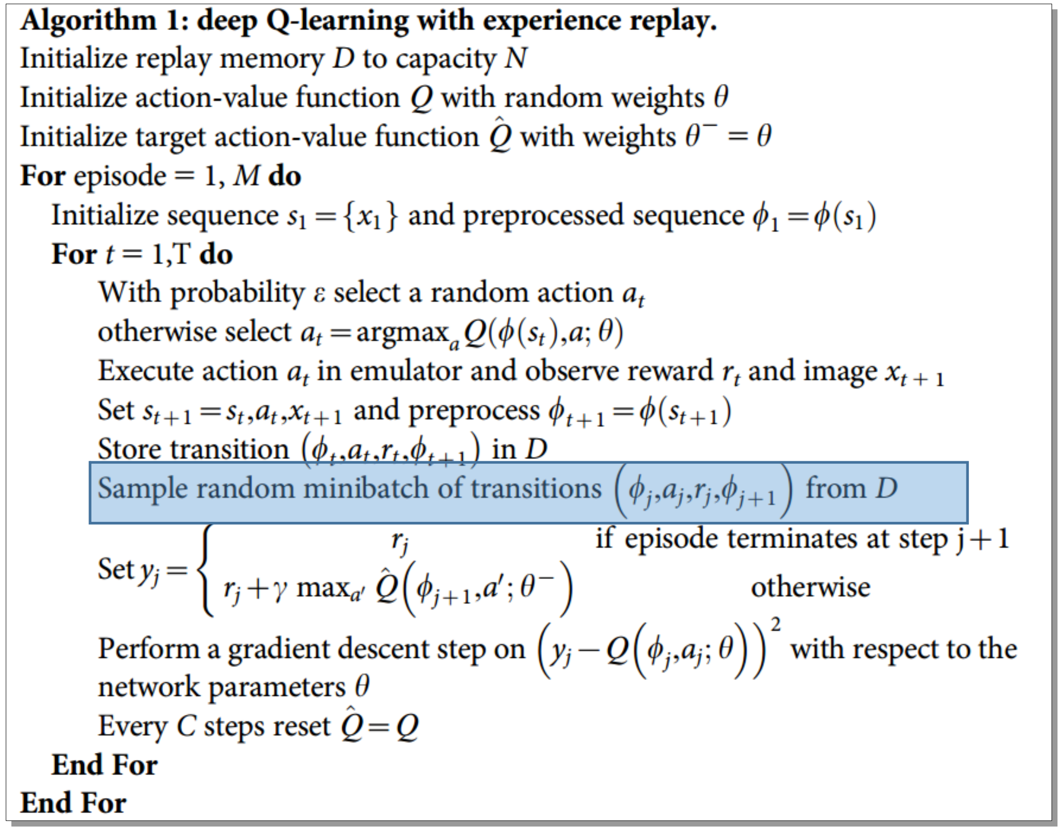


```
**Algorithm 1: deep Q-learning with experience replay.**
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, $M$ do
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t$ = 1,T do
    With probability $\varepsilon$ select a random action $a_t$
    otherwise select $a_t = \mathrm{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the
    network parameters $\theta$
    Every $C$ steps reset $\hat{Q} = Q$
  End For
End For
```

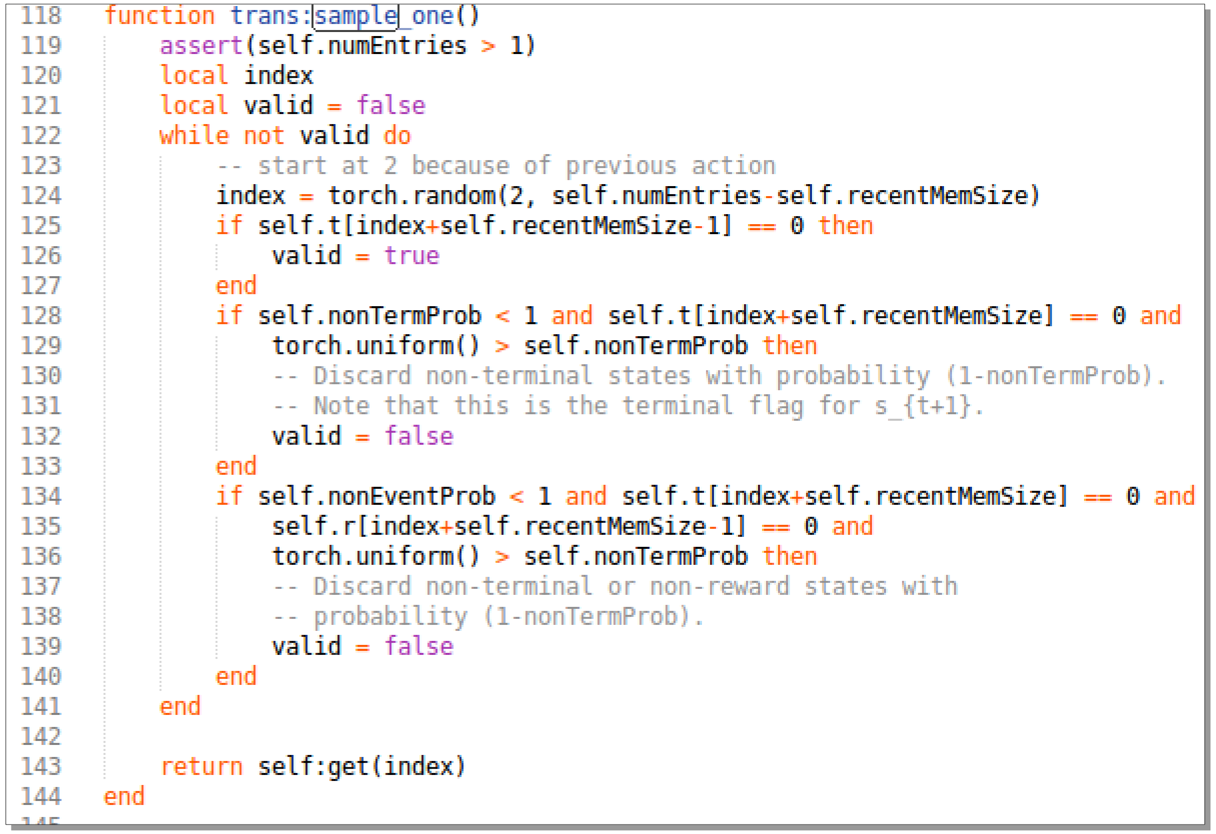


```lua
--Store transition s, a, r, s'
if self.lastState and not testing then
    self.transitions:add(self.lastState, self.lastAction, reward,
                         self.lastTerminal, priority)
end
```

Algorithm Overview – Get Minibatch

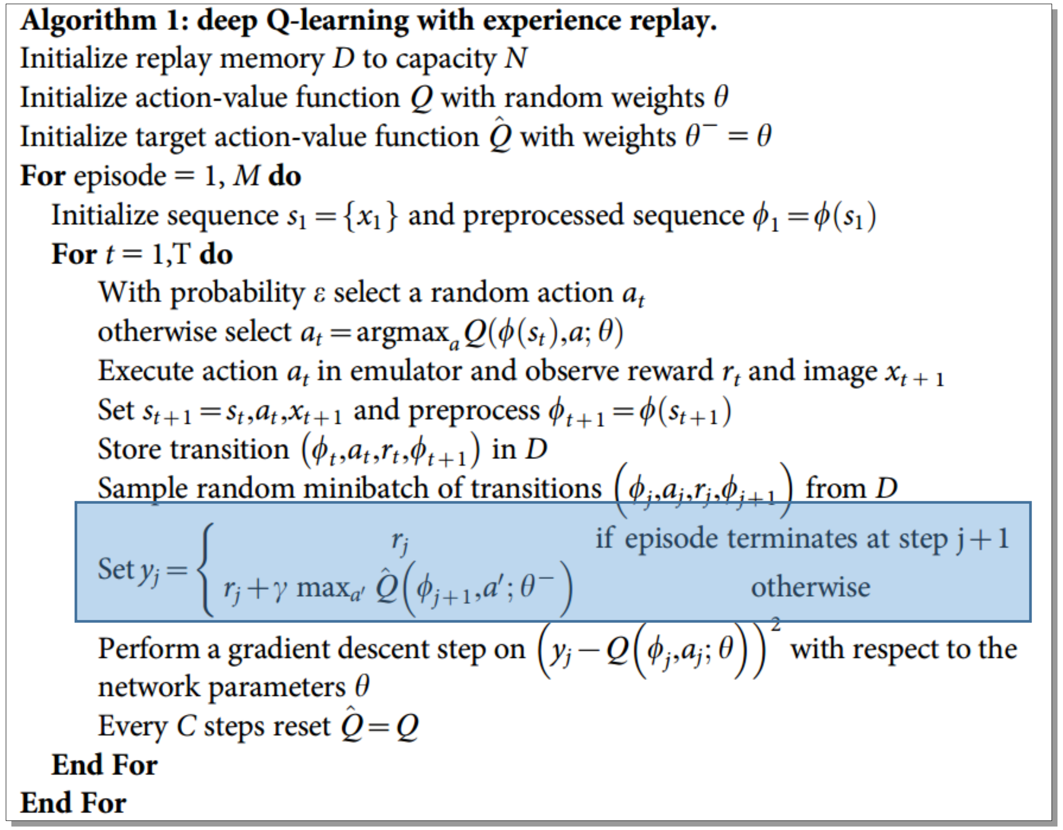


```
Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, $M$ do
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    For $t$ = 1,T do
        With probability $\varepsilon$ select a random action $a_t$
        otherwise select $a_t = \mathrm{argmax}_a Q(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
        Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the
        network parameters $\theta$
        Every $C$ steps reset $\hat{Q} = Q$
    End For
End For
```

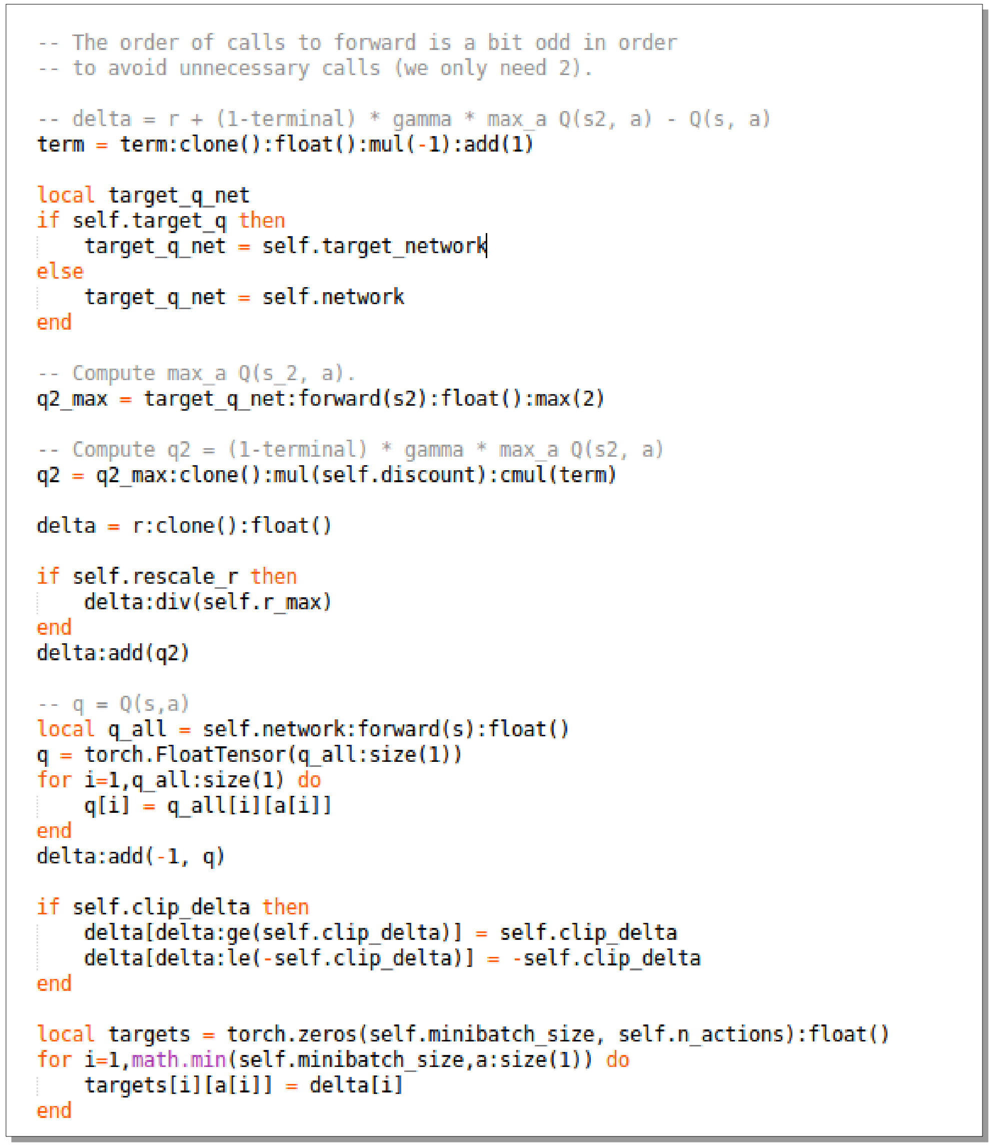


```lua
function trans:sample_one()
    assert(self.numEntries > 1)
    local index
    local valid = false
    while not valid do
        -- start at 2 because of previous action
        index = torch.random(2, self.numEntries-self.recentMemSize)
        if self.t[index+self.recentMemSize-1] == 0 then
            valid = true
        end
        if self.nonTermProb < 1 and self.t[index+self.recentMemSize] == 0 and
            torch.uniform() > self.nonTermProb then
            -- Discard non-terminal states with probability (1-nonTermProb).
            -- Note that this is the terminal flag for s_{t+1}.
            valid = false
        end
        if self.nonEventProb < 1 and self.t[index+self.recentMemSize] == 0 and
            self.r[index+self.recentMemSize-1] == 0 and
            torch.uniform() > self.nonTermProb then
            -- Discard non-terminal or non-reward states with
            -- probability (1-nonTermProb).
            valid = false
        end
    end

    return self:get(index)
end
```

Algorithm Overview – Get Rewards

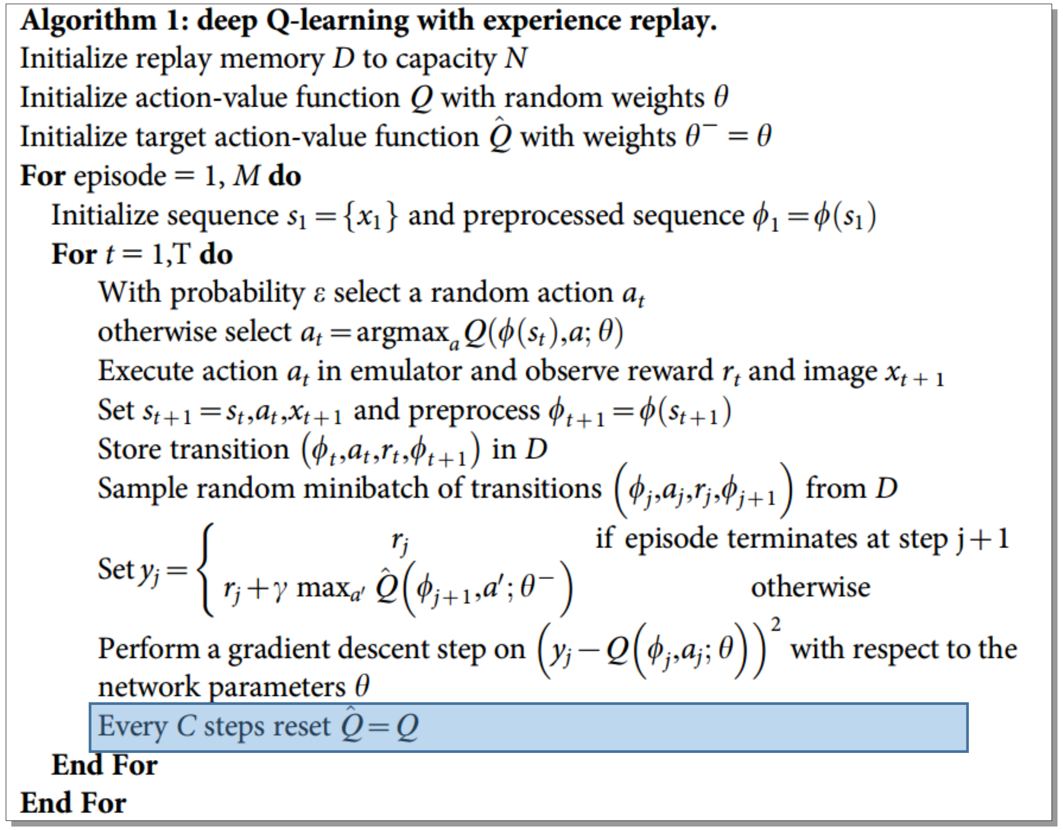


```
Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, $M$ do
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    For $t$ = 1,T do
        With probability $\varepsilon$ select a random action $a_t$
        otherwise select $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
        Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^-\right) & \text{otherwise} \end{cases}$
        Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the
        network parameters $\theta$
        Every $C$ steps reset $\hat{Q} = Q$
    End For
End For
```

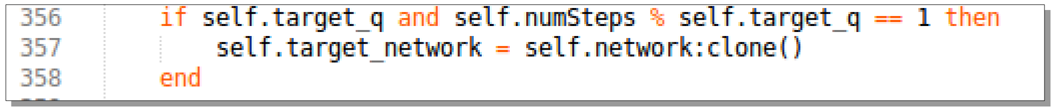


```lua
-- The order of calls to forward is a bit odd in order
-- to avoid unnecessary calls (we only need 2).

-- delta = r + (1-terminal) * gamma * max_a Q(s2, a) - Q(s, a)
term = term:clone():float():mul(-1):add(1)

local target_q_net
if self.target_q then
    target_q_net = self.target_network
else
    target_q_net = self.network
end

-- Compute max_a Q(s_2, a).
q2_max = target_q_net:forward(s2):float():max(2)

-- Compute q2 = (1-terminal) * gamma * max_a Q(s2, a)
q2 = q2_max:clone():mul(self.discount):cmul(term)

delta = r:clone():float()

if self.rescale_r then
    delta:div(self.r_max)
end
delta:add(q2)

-- q = Q(s,a)
local q_all = self.network:forward(s):float()
q = torch.FloatTensor(q_all:size(1))
for i=1,q_all:size(1) do
    q[i] = q_all[i][a[i]]
end
delta:add(-1, q)

if self.clip_delta then
    delta[delta:ge(self.clip_delta)] = self.clip_delta
    delta[delta:le(-self.clip_delta)] = -self.clip_delta
end

local targets = torch.zeros(self.minibatch_size, self.n_actions):float()
for i=1,math.min(self.minibatch_size,a:size(1)) do
    targets[i][a[i]] = delta[i]
end
```

Algorithm Overview – Update Target Network



```
**Algorithm 1: deep Q-learning with experience replay.**
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
**For** episode $= 1, M$ **do**
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    **For** $t = 1$,T **do**
        With probability $\varepsilon$ select a random action $a_t$
        otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
        Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
        Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the
        network parameters $\theta$
        Every $C$ steps reset $\hat{Q} = Q$
    **End For**
**End For**
```



```lua
    if self.target_q and self.numSteps % self.target_q == 1 then
        self.target_network = self.network:clone()
    end
```

What We Can Do

DQNs for Game Playing

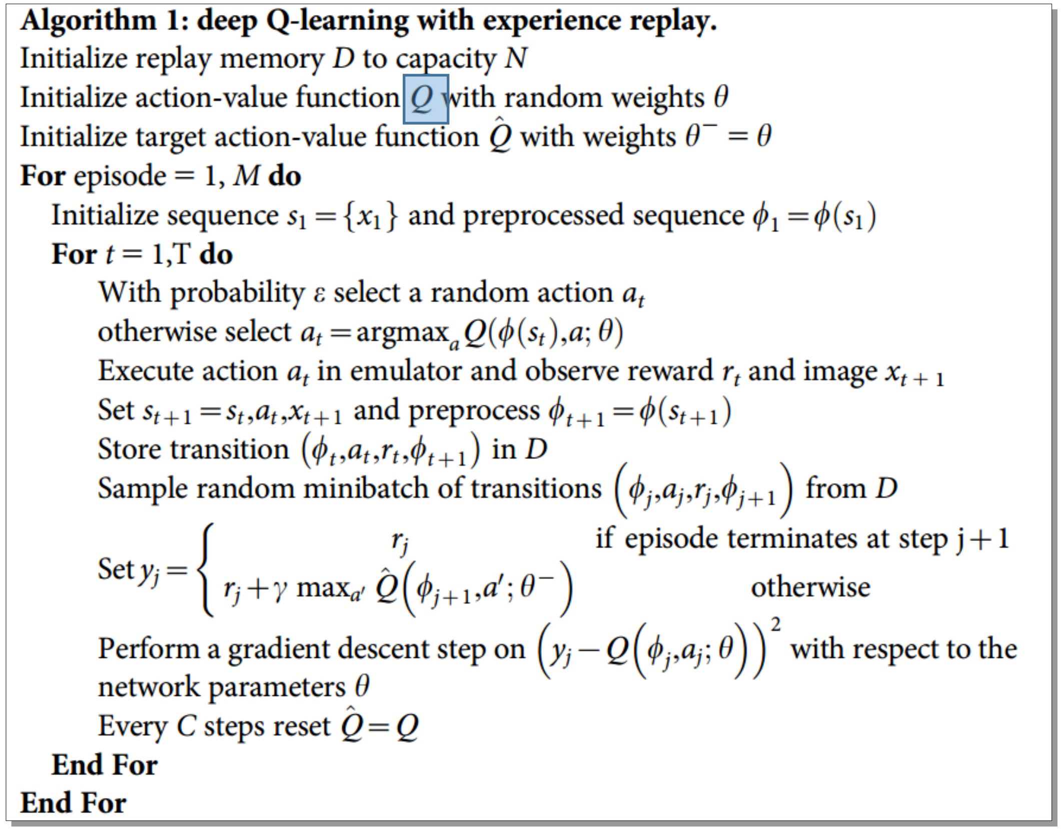
- For now, train a model using their default architecture
- Their parameters are identical across 49 games, indicating some robustness of these parameters
- In the future:
  - Pretrain visual model and reuse between games
  - Use model selection to pre-initialize game depending on type (puzzle, maze hack-and-slash, Mario-like, racing game, etc.)

Possible Labor Division

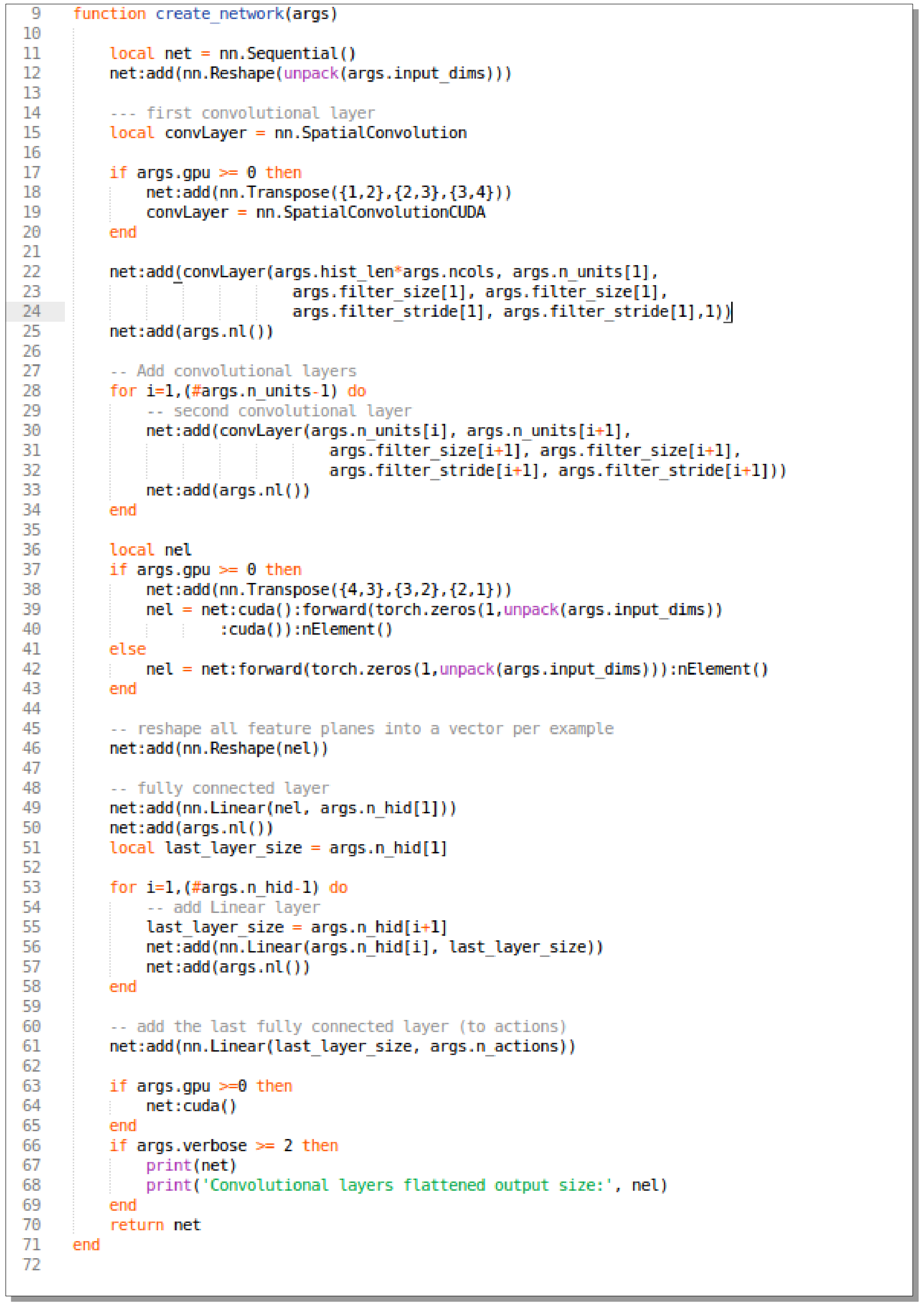
- Input and Preprocessing
  - Interface to the game simulator
  - Represent game state as simplified image
- DQN Network without the deep part
  - Setup epsilon-greedy framework, replay memory, and Q-target copying
  - Build generalized DQN without the q-estimator part
- ConvNet Function Approximation
  - Set up a simple interface to use the convnet to estimate the rewards

Appendix

Alg. Overview ConvNet



```
Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, $M$ do
   Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
   For $t = 1$,T do
      With probability $\varepsilon$ select a random action $a_t$
      otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
      Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
      Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
      Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
      Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
      Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
      Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the
      network parameters $\theta$
      Every $C$ steps reset $\hat{Q} = Q$
   End For
End For
```



```lua
function create_network(args)

    local net = nn.Sequential()
    net:add(nn.Reshape(unpack(args.input_dims)))

    --- first convolutional layer
    local convLayer = nn.SpatialConvolution

    if args.gpu >= 0 then
        net:add(nn.Transpose({1,2},{2,3},{3,4}))
        convLayer = nn.SpatialConvolutionCUDA
    end

    net:add(convLayer(args.hist_len*args.ncols, args.n_units[1],
                        args.filter_size[1], args.filter_size[1],
                        args.filter_stride[1], args.filter_stride[1],1))
    net:add(args.nl())

    -- Add convolutional layers
    for i=1,(#args.n_units-1) do
        -- second convolutional layer
        net:add(convLayer(args.n_units[i], args.n_units[i+1],
                            args.filter_size[i+1], args.filter_size[i+1],
                            args.filter_stride[i+1], args.filter_stride[i+1]))
        net:add(args.nl())
    end

    local nel
    if args.gpu >= 0 then
        net:add(nn.Transpose({4,3},{3,2},{2,1}))
        nel = net:cuda():forward(torch.zeros(1,unpack(args.input_dims))
                :cuda()):nElement()
    else
        nel = net:forward(torch.zeros(1,unpack(args.input_dims))):nElement()
    end

    -- reshape all feature planes into a vector per example
    net:add(nn.Reshape(nel))

    -- fully connected layer
    net:add(nn.Linear(nel, args.n_hid[1]))
    net:add(args.nl())
    local last_layer_size = args.n_hid[1]

    for i=1,(#args.n_hid-1) do
        -- add Linear layer
        last_layer_size = args.n_hid[i+1]
        net:add(nn.Linear(args.n_hid[i], last_layer_size))
        net:add(args.nl())
    end

    -- add the last fully connected layer (to actions)
    net:add(nn.Linear(last_layer_size, args.n_actions))

    if args.gpu >=0 then
        net:cuda()
    end
    if args.verbose >= 2 then
        print(net)
        print('Convolutional layers flattened output size:', nel)
    end
    return net
end
```