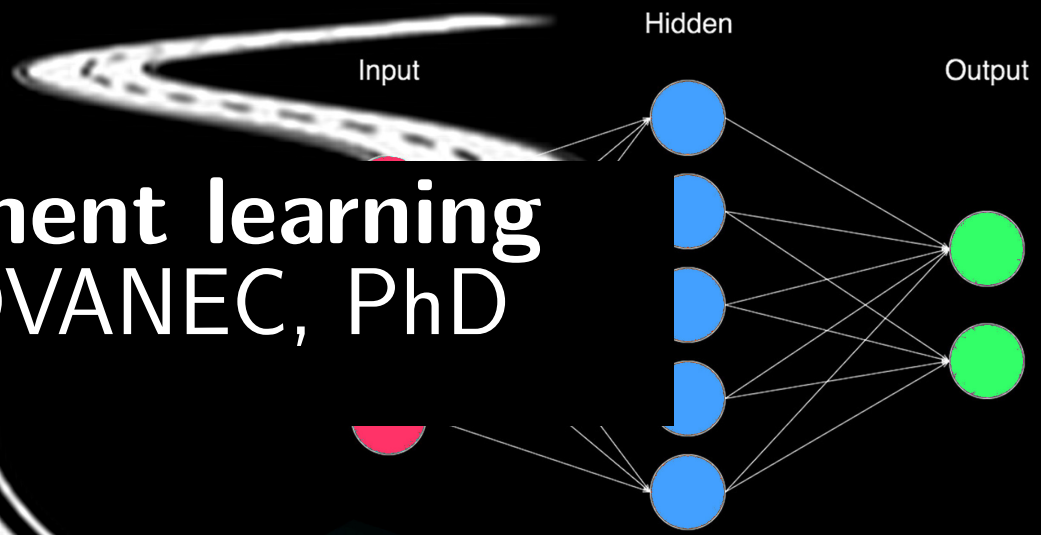
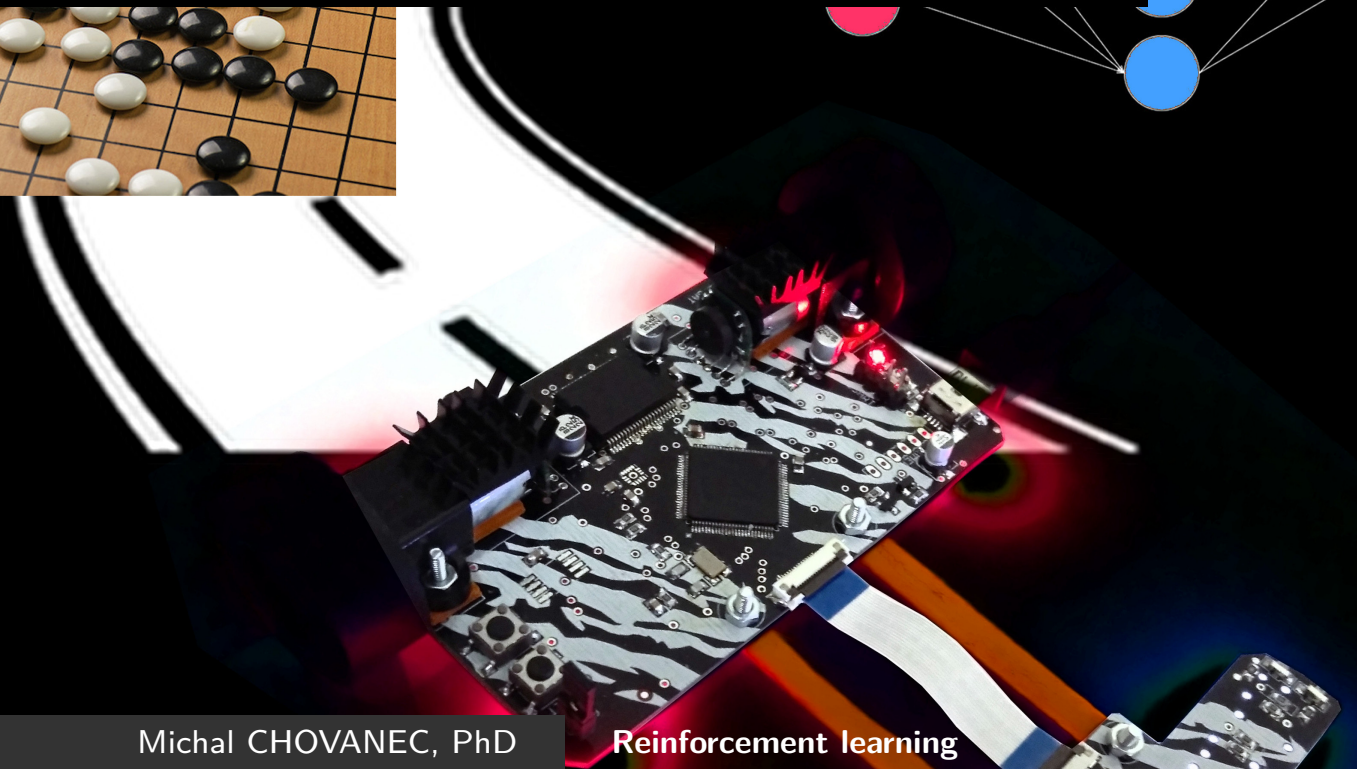
$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\left[r_{t+1} + \lambda \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\right]$$

(The New Action Value = The Old Value) + The Learning Rate × (The New Information − the Old Information)

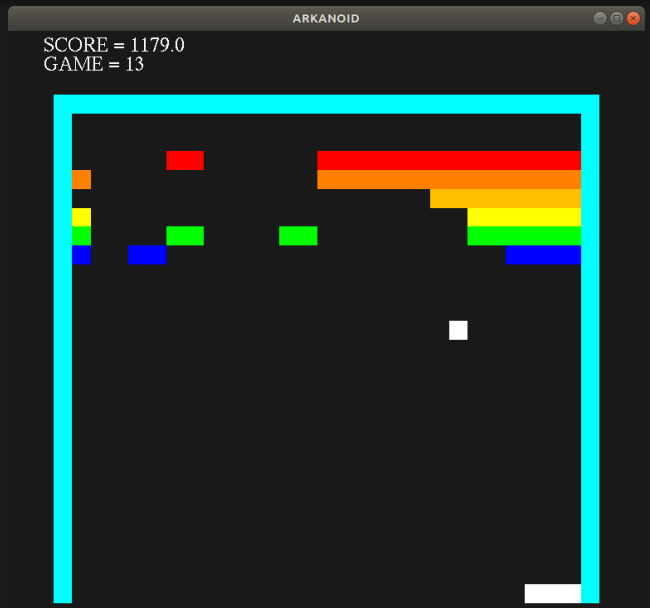


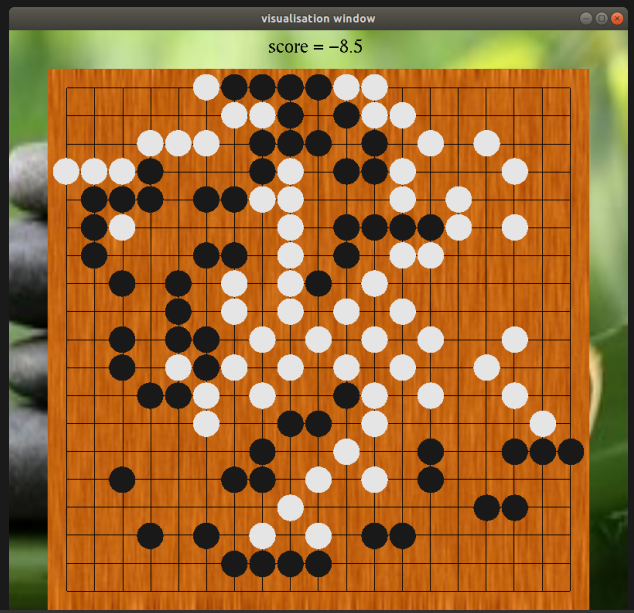
Reinforcement learning

Michal CHOVANEC, PhD

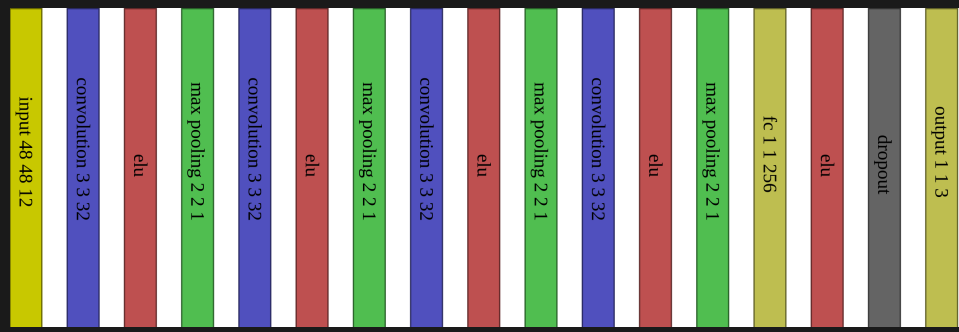


Reinforcement learning



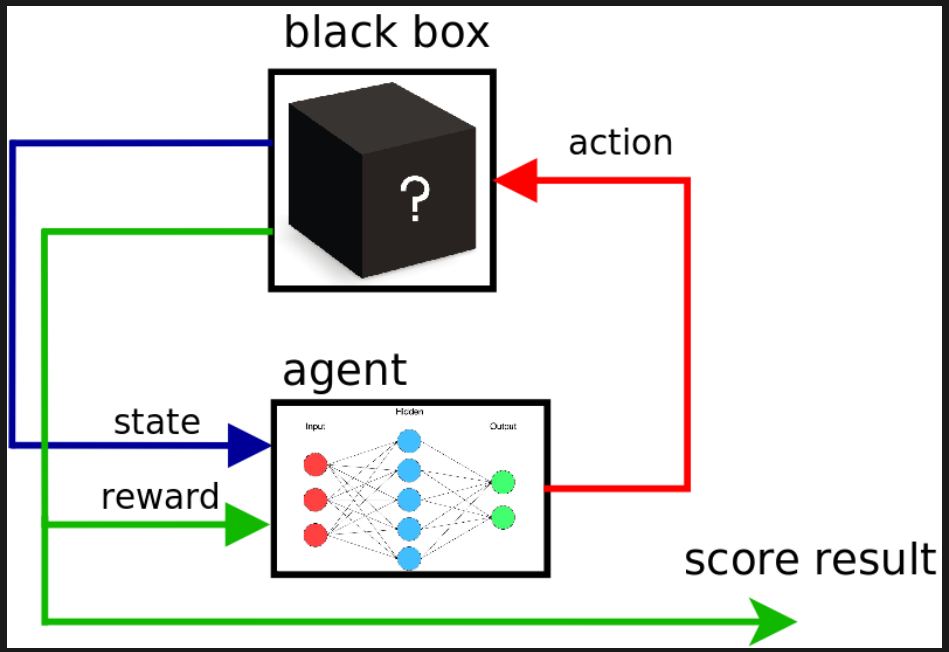


- playing Atari (2013)
- playing GO (2016)
- playing Doom (2018)
- playing StarCraft (2019)
- Stock market Agents, creating software, robotics



Reinforcement learning

- learning from punishments and rewards
  - obtain **state**
  - choose **action**
  - **execute** action
  - obtain **reward**
  - learn from **experiences**, $Q(s, a)$

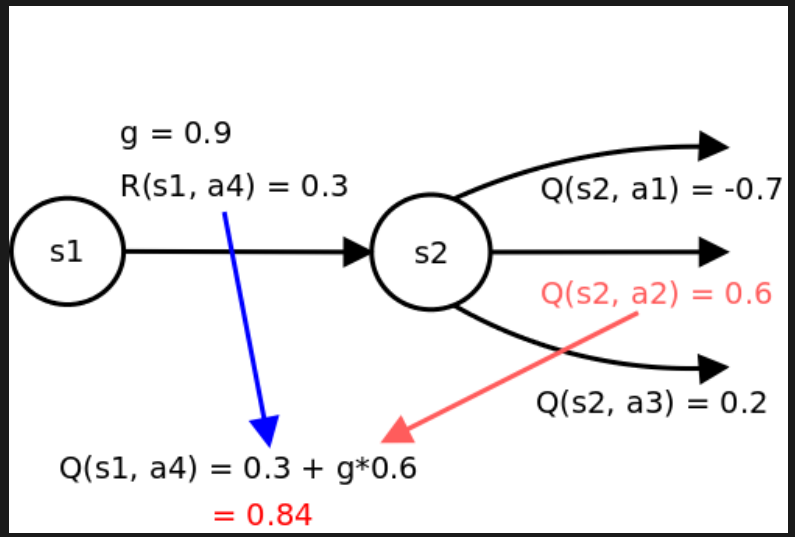


What is $Q(s, a)$

Q-learning algorithm

$Q(s, a)$ value of action a, executed in state s

$$Q(s, a) = R + \gamma \max_{\alpha'} Q(s', \alpha')$$



where

s is state

a is action

s' is next state

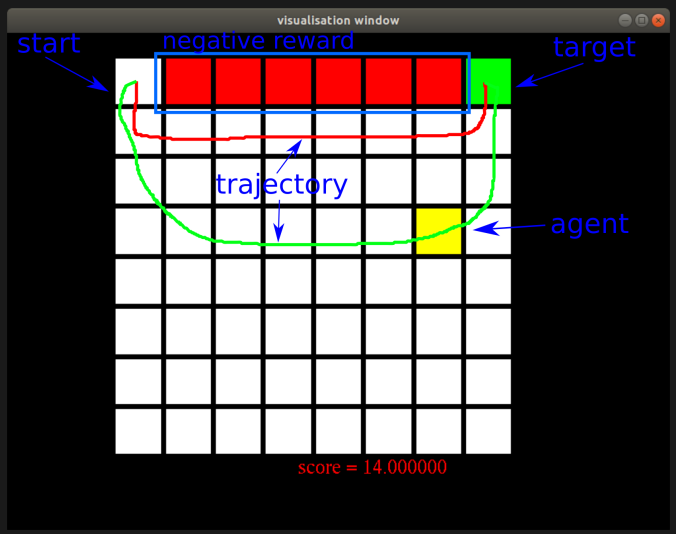
a' is best action in next state

$R(s, a)$ is reward

$\gamma \in \langle 0, 1 \rangle$ is discount factor

Q table example

- control agent (yellow) from start to target, avoiding cliff (red)
- state : one of 64 fields
- 4 actions, moving up, left, down, right
- reward : +1 on target field, -1 on cliff field



State	UP	LEFT	DOWN	RIGHT
0			0.43	-1
1, 2, 3, 4, 5, 6	T	T	T	T
7	T	T	T	T
8				0.49
9	-1			0.53
10	-1			0.59
11	-1			0.65
12	-1			0.72
13	-1			0.81
14	-1			0.9
15	1			

Q table agent

```
def __init__(self):
    #get state size, and actions count
    self.states_count  = self.env.get_size()
    self.actions_count = self.env.get_actions_count()
    #init Q table, using number of states and actions
    self.q_table = numpy.zeros((self.states_count, self.actions_count))

def main(self):
    #Q learning needs to remember current state, action and previous state and action
    self.state_prev = self.state
    self.state      = self.env.get_observation().argmax()

    self.action_prev    = self.action
    #select action is done by probality selection using epsilon
    self.action         = self.select_action(self.q_table[self.state], epsilon)

    #obtain reward from environment
    reward = self.env.get_reward()

    #process Q learning
    q_tmp = self.q_table[self.state].max()

    d = reward + self.gamma*q_tmp - self.q_table[self.state_prev][self.action_prev]

    self.q_table[self.state_prev][self.action_prev]+= self.alpha*d

    #execute action
    self.env.do_action(self.action)
```

Deep Q network - DQN (2013)

Approximate $Q(s, a)$ using deep neural network as $\hat{Q}(s, a; w)$, where w are learnable network parameters

$$Q(s, a) = R + \gamma \max_{\alpha'} Q(s', \alpha')$$
$$\hat{Q}(s, a; w) = R + \gamma \max_{\alpha'} \hat{Q}(s', \alpha'; w)$$

error to minimize

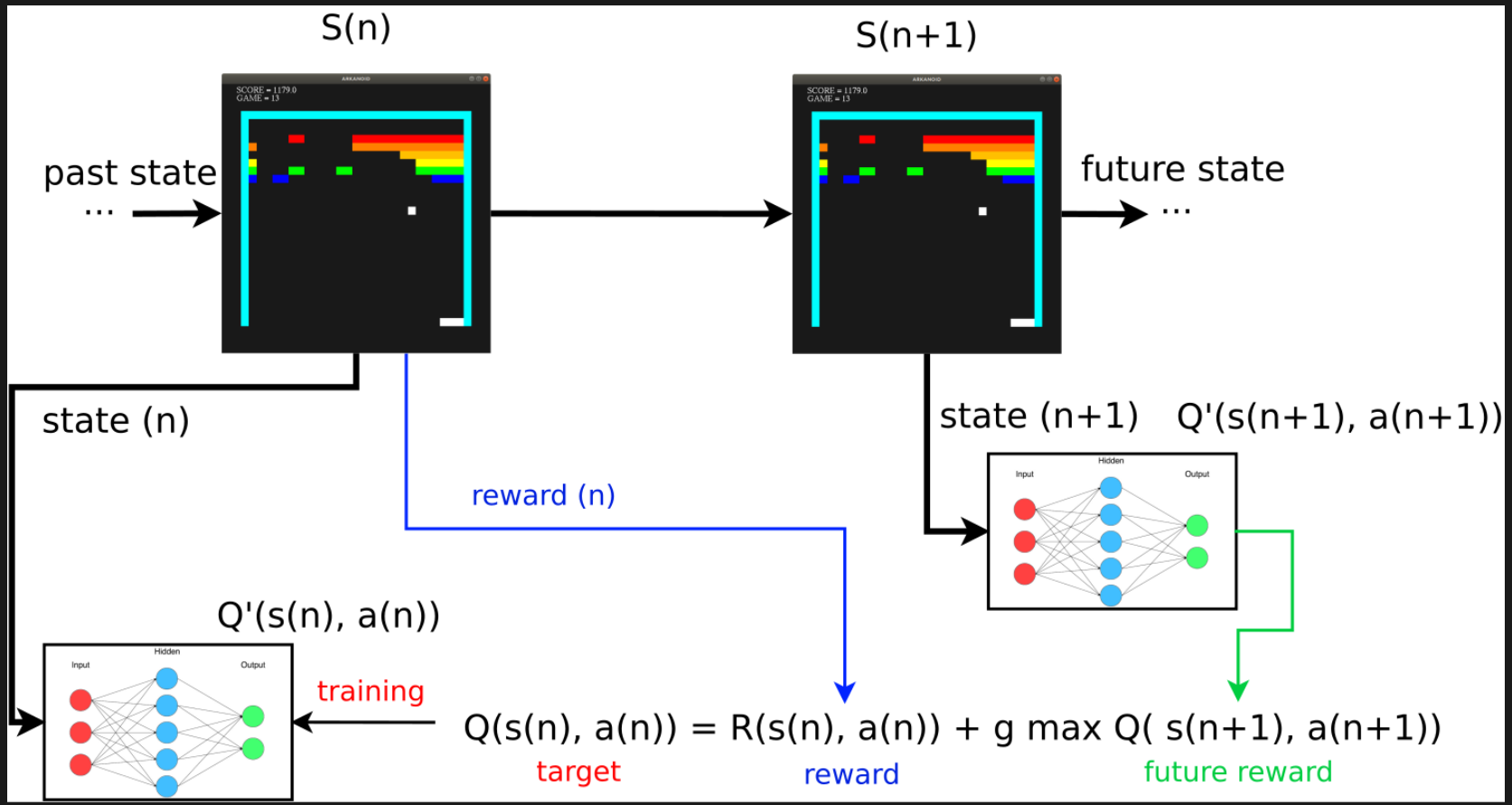
$$E = \underbrace{R + \gamma \max_{\alpha'} \hat{Q}(s', \alpha'; w)}_{\textbf{target value}} - \underbrace{\hat{Q}(s, a; w)}_{\textbf{predicted value}}$$

weights gradient

$$\Delta w = \eta E \nabla_w \hat{Q}(s, \alpha, w)$$

This naive NN doesn't works - solution DQN

Naive solutuion

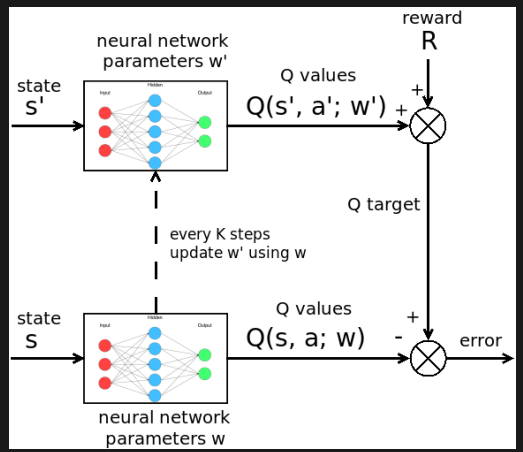


Deep Q network - DQN

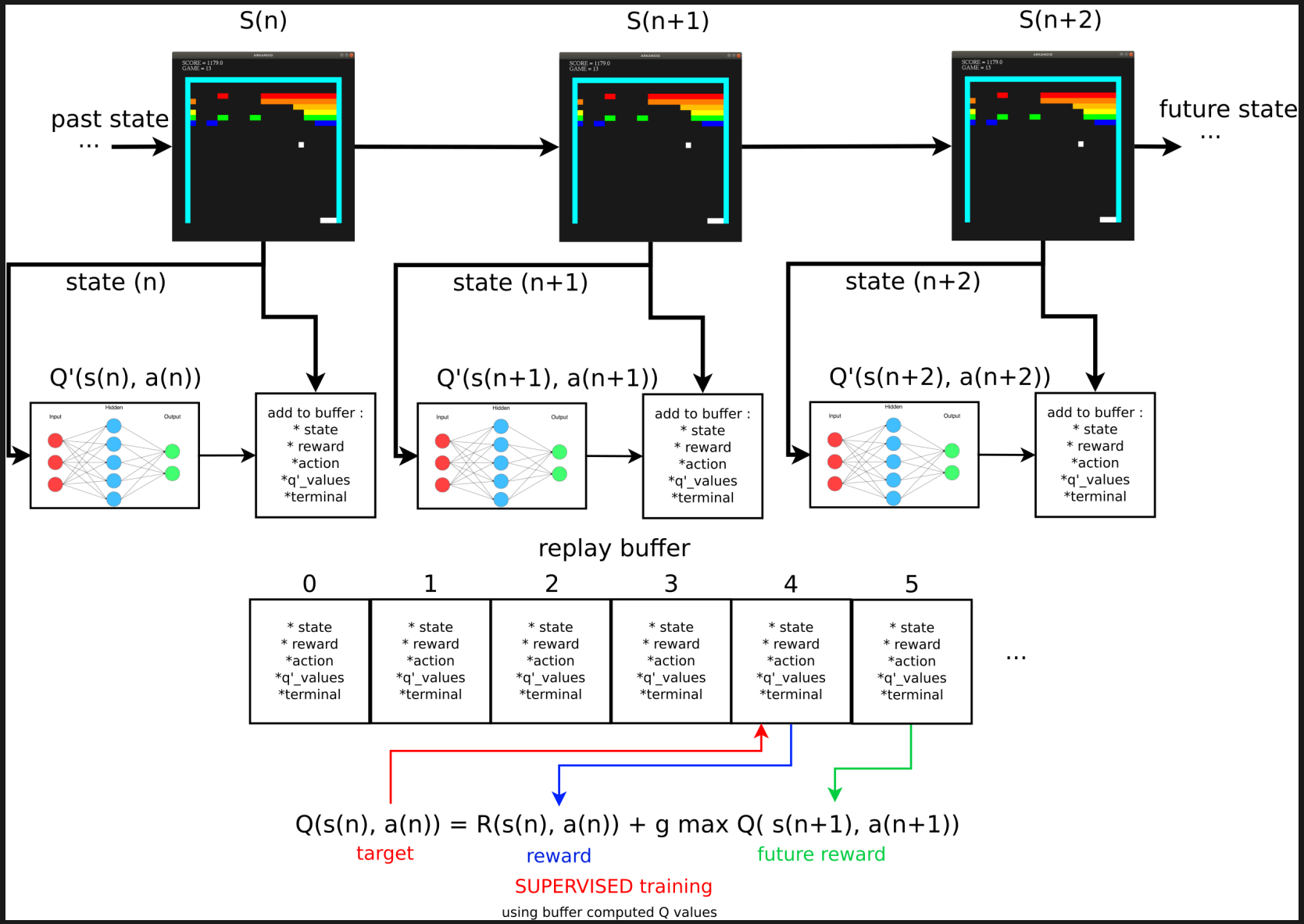
- **correlated states** :
  experience replay buffer
- **unstable training** :
  non-stationary target value $\hat{Q}(s, a; w)$, depends on w, use temporary fixed weights w'
- **unknow gradients values** :
  clip or normalise gradients into $\langle -1, 1 \rangle$

DQN equation

$$\hat{Q}(s, a; w) = R + \gamma \max_{\alpha'} \hat{Q}(s', \alpha'; w')$$



Experience replay buffer solution



DQN best practices

- **avoid** poor CNN, without FC layer, FC layer increase training speed, **avoid** global average pooling (GAP)
- for **short time sequence** use frame stacking, for **long time sequence** time sequences use one LSTM (GRU) layer instead of FC
- **use** 3x3 kernels and deeper architecture

layer	kernel size	output size
input	-	48x48x(3x4)
conv	3x3x32	48x48x32
max pooling	2x2	24x24x32
conv	3x3x32	24x24x32
max pooling	2x2	12x12x32
conv	3x3x32	12x12x32
max pooling	2x2	6x6x32
conv	3x3x32	6x6x32
max pooling	2x2	3x3x32
flatten		1x1x288
fc	256	1x1x256
fc	actions_count	actions_count

Experience replay buffer

store : state, q_values, action, reward, terminal state

```
#init buffer
self.replay_buffer = []

#perform add at each step
def add(self):
    buffer_item  = {
        "state"          : state_vector,
        "q_values"       : q_values,
        "action"         : self.action,
        "reward"         : self.reward,
        "terminal"       : self.env.is_done()
    }
    self.replay_buffer.append(buffer_item)
```

Experience replay buffer - computing Q values

```
if self.replay_buffer.is_full():
    #compute buffer Q values, using Q learning
    for n in reversed(range(self.replay_buffer_size-1)):
        #choose zero gamme if current state is terminal
        if self.replay_buffer[n]["terminal"] == True:
            gamma = 0.0
        else:
            gamma = self.gamma

        action_id = self.replay_buffer[n]["action"]

        #Q-learning : Q(s[n], a[n]) = R[n] + gamma*max(Q(s[n+1]))
        q_next = max(self.replay_buffer[n+1]["q_values"])

        self.replay_buffer[n]["q_values"][action_id] = self.replay_buffer[n]["reward"] +

        #clamp Q values into range <-10, 10> to prevent divergence
        for action in range(self.env.get_actions_count()):
            self.replay_buffer[n]["q_values"][action] = self.__clamp(self.replay_buffer[
```

Training CNN

we converted reinfrocement learning problem to supervised (regression) learning problem

- replay_buffer[n]["state"]: network input
- replay_buffer[n]["q_values"] : target values

```
self.cnn.set_training_mode()

for i in range(self.replay_buffer_size):

    #choose random item, to break correlations
    idx = random.randint(0, self.replay_buffer_size - 1)

    state = self.replay_buffer[idx]["state"]
    target_q_values = self.replay_buffer[idx]["q_values"]

    #fit network
    self.cnn.train(target_q_values, state)

self.cnn.unset_training_mode()

#clear buffer
self.replay_buffer = []
```

Q&A - consultations on Rysy Hut 2250 m.a.s.l.