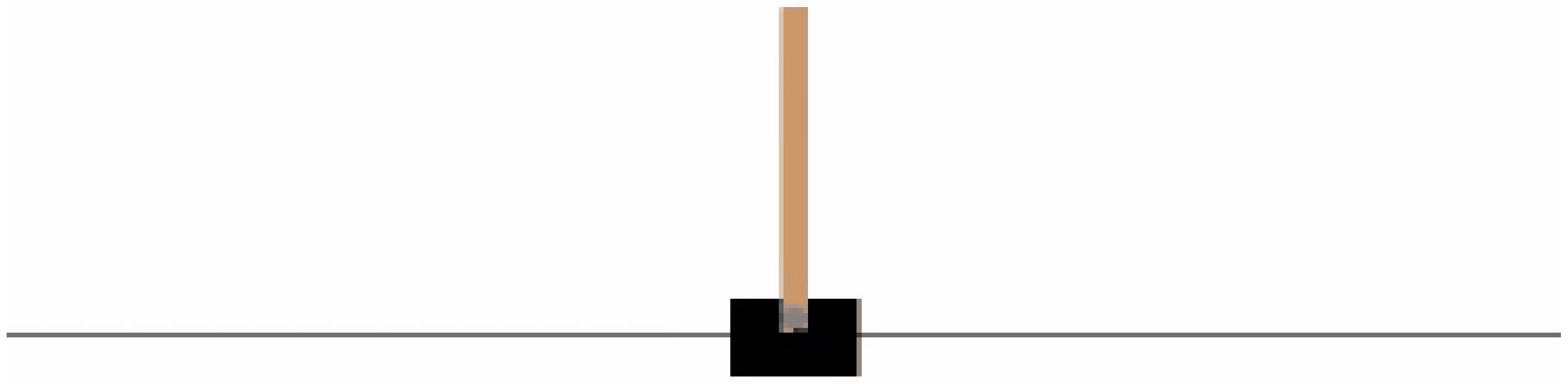


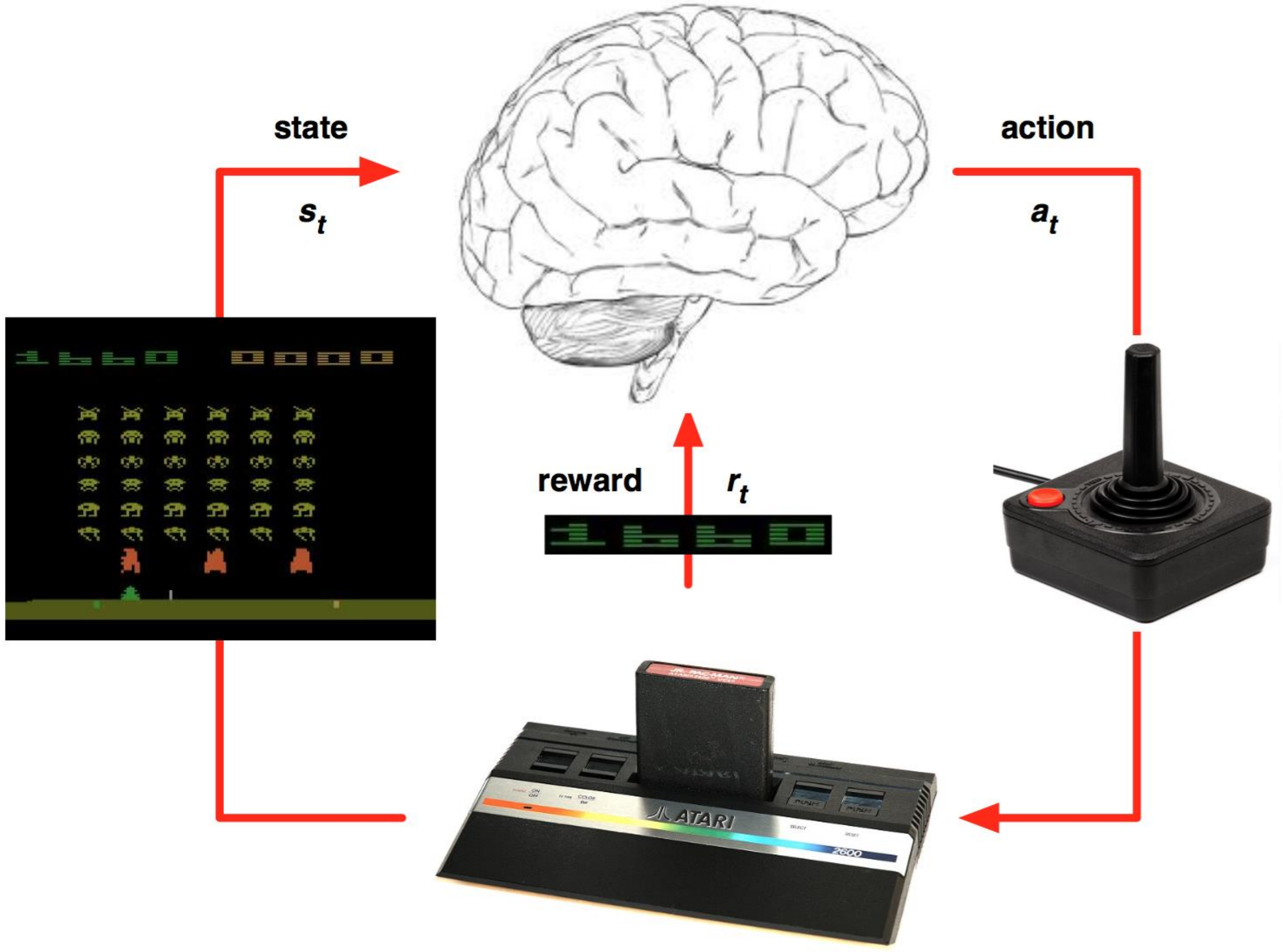
Deep Q Learning

By Comdet Phaudphut, fb.com/comdet, comdet.p@gmail.com

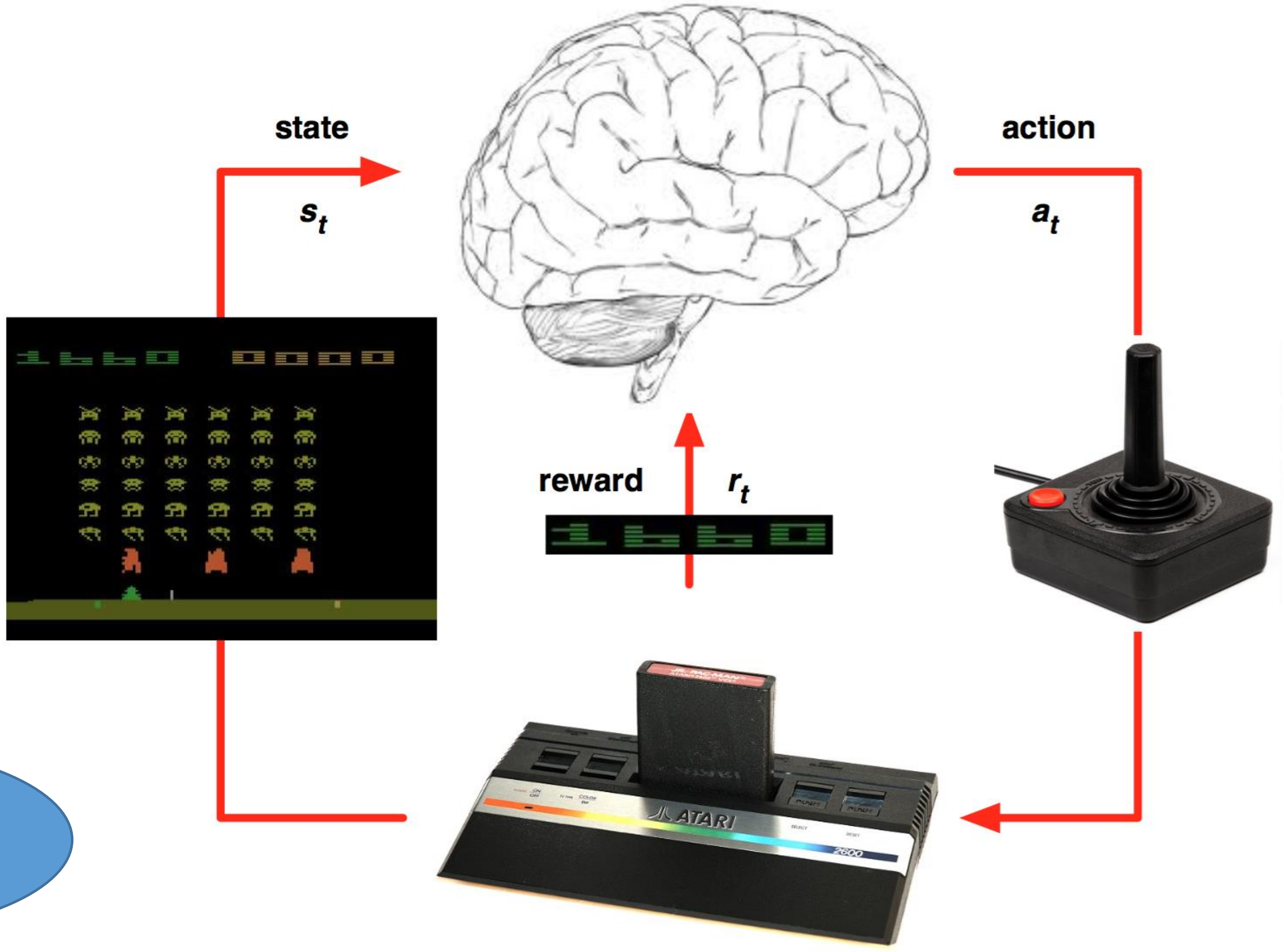
Review Prev Lab

- We use CNN and gain more over 99% of accuracy.
- We know how convolution and pooling work.
- BUT
  - In real life data didn’t represent as dataset ready to train

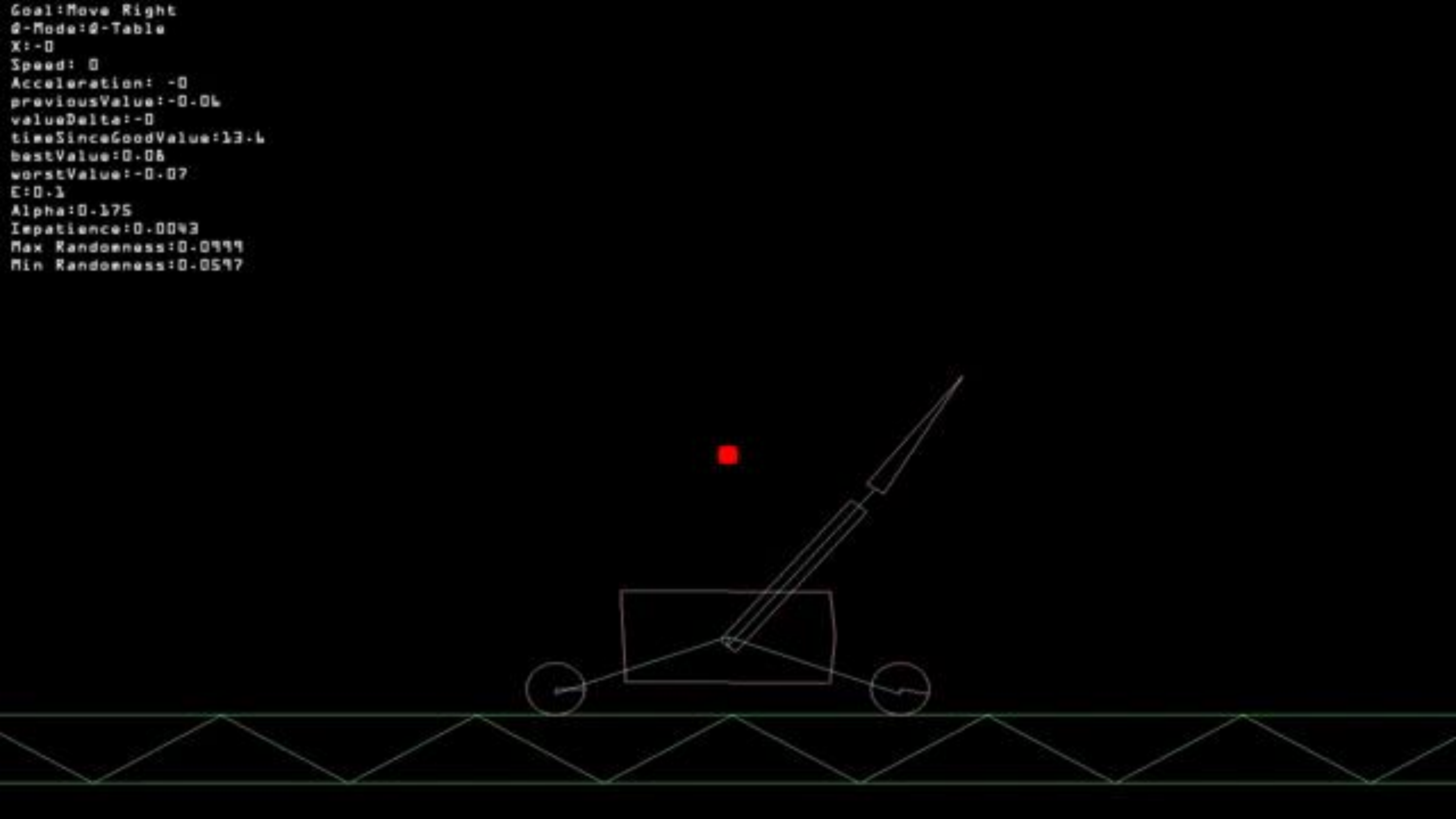
Reinforcement Learning



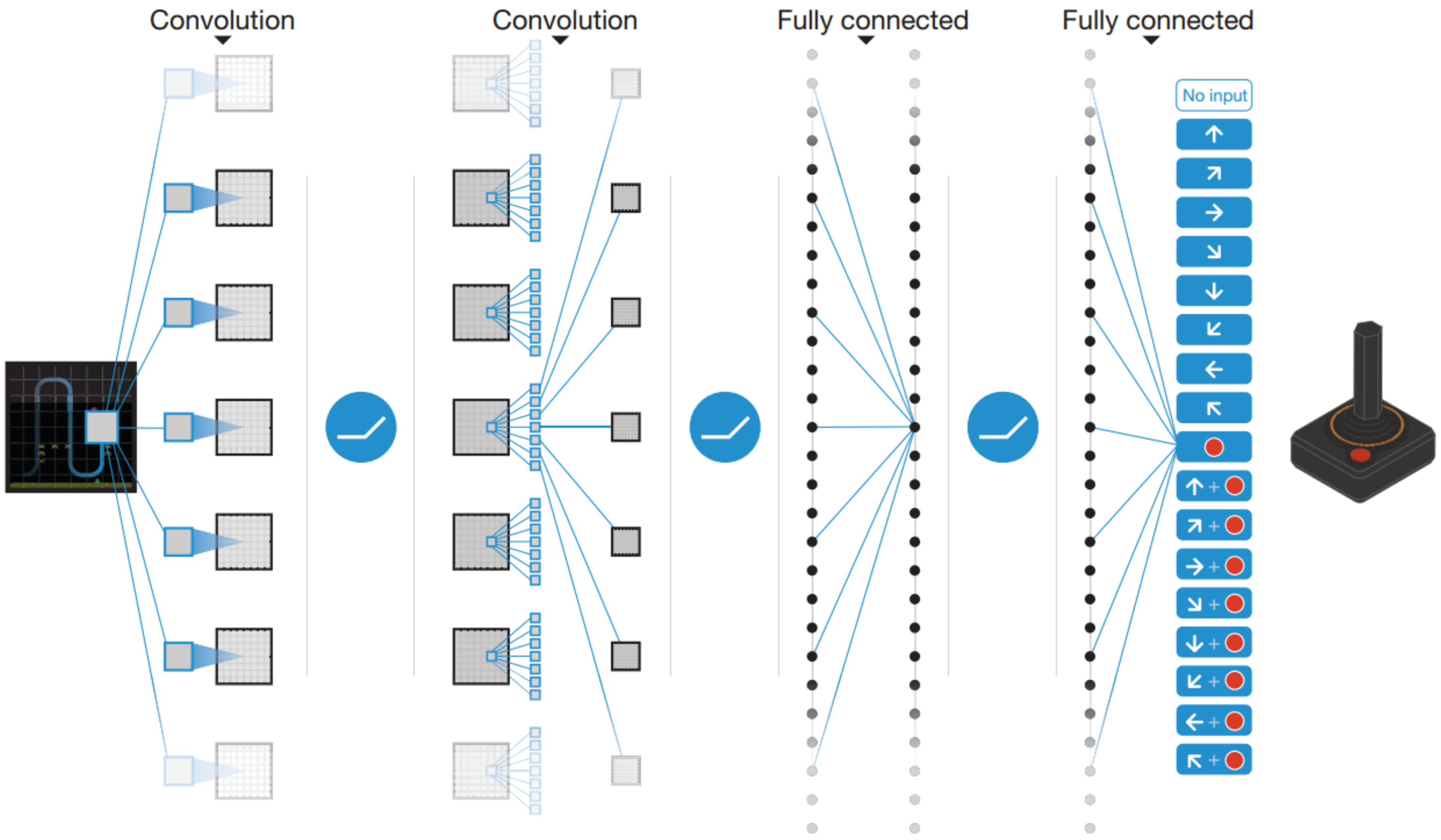
Reinforcement Learning



Maybe reward can get at the end of the game


Goal:Move Right
Q-Mode:Q-Table
X:-0
Speed: 0
Acceleration: -0
previousValue:-0.06
valueDelta:-0
timeSinceGoodValue:13.6
bestValue:0.08
worstValue:-0.07
E:0.1
Alpha:0.175
Impatience:0.0043
Max Randomness:0.0999
Min Randomness:0.0597

Algorithm



Convolution
Convolution
Fully connected
Fully connected
No input

Formulation

1 $$Q^*(s,a) = \mathbf{E}_{s'}\left[r + \gamma \max_{a'} Q(s',a'))\right]$$

2 $$Q'_t(s,a \mid \theta) \approx Q(s,a)$$

3 $$L(\theta) = \mathbf{E}_{s,a,r}\left[\left(\mathbf{E}_{s'}\left[r + \gamma \max_{a'} Q'_t(s',a' \mid \theta)\right] - Q_{t+1}(s,a \mid \theta)\right)^2\right]$$

Target

Variable

Algorithm

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    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 $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation $\boxed{3}$
  end for
end for

Our parameter

```
ACTIONS = 2 # number of valid actions
GAMMA = 0.99 # decay rate of past observations
OBSERVE = 1000. # timesteps to observe before training
EXPLORE = 2000000. # frames over which to anneal epsilon
FINAL_EPSILON = 0.0001 # final value of epsilon
INITIAL_EPSILON = 0.1 # starting value of epsilon
REPLAY_MEMORY = 1000 # number of previous transitions to remember
BATCH = 128 # size of minibatch
```

We create function that easy to use

```
def weight_variable(shape):
    initial = tf.truncated_normal(shape, stddev = 0.01)
    return tf.Variable(initial)

def bias_variable(shape):
    initial = tf.constant(0.01, shape = shape)
    return tf.Variable(initial)

def conv2d(x, W, stride):
    return tf.nn.conv2d(x, W, strides = [1, stride, stride, 1], padding = "SAME")

def max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize = [1, 2, 2, 1], strides = [1, 2, 2, 1], padding = "SAME")
```

We create variables

```
#create model
W_conv1 = weight_variable([8, 8, 4, 32])
b_conv1 = bias_variable([32])

W_conv2 = weight_variable([4, 4, 32, 64])
b_conv2 = bias_variable([64])

W_conv3 = weight_variable([3, 3, 64, 64])
b_conv3 = bias_variable([64])

W_fc1 = weight_variable([1600, 512])
b_fc1 = bias_variable([512])

W_fc2 = weight_variable([512, ACTIONS])
b_fc2 = bias_variable([ACTIONS])
```

We create model

```
# input layer
s = tf.placeholder("float", [None, 80, 80, 4])

# hidden layers
h_conv1 = tf.nn.relu(conv2d(s, W_conv1, 4) + b_conv1) ### ===> 80x80x4 conv 4 ==> 20x20x32
h_pool1 = max_pool_2x2(h_conv1)                       ### ===> 20x20x32 maxpool => 10x10x64

h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2, 2) + b_conv2) ###=====10x10 conv2 ===>5x5x64
#h_pool2 = max_pool_2x2(h_conv2)

h_conv3 = tf.nn.relu(conv2d(h_conv2, W_conv3, 1) + b_conv3) ## 5x5 conv 1 padsame =>5x5x64
#h_pool3 = max_pool_2x2(h_conv3)

#h_pool3_flat = tf.reshape(h_pool3, [-1, 256])
h_conv3_flat = tf.reshape(h_conv3, [-1, 1600])              ##5x5x64 flaten =>1600

h_fc1 = tf.nn.relu(tf.matmul(h_conv3_flat, W_fc1) + b_fc1)

# readout layer
readout = tf.matmul(h_fc1, W_fc2) + b_fc2
```

We create and here loss and game and D?

```
# define the cost function
a = tf.placeholder("float", [None, ACTIONS])
y = tf.placeholder("float", [None])

readout_action = tf.reduce_sum(tf.multiply(readout, a), reduction_indices=1)
cost = tf.reduce_mean(tf.square(y - readout_action)) #rms root mean square for cost function
train_step = tf.train.AdamOptimizer(1e-6).minimize(cost)

# open up a game state to communicate with emulator
game_state = game.GameState()

# saving and loading networks
saver = tf.train.Saver()
checkpoint = tf.train.get_checkpoint_state("saved_networks")
#if checkpoint and checkpoint.model_checkpoint_path:
#    saver.restore(sess, checkpoint.model_checkpoint_path)
#    print("Successfully loaded:", checkpoint.model_checkpoint_path)
#else:
#    print("Could not find old network weights")

# store the previous observations in replay memory
D = deque()
```

Play game and saved in D

```
# start training
epsilon = INITIAL_EPSILON
rp = 0
while len(D) < REPLAY_MEMORY:

    # choose an action epsilon greedily
    readout_t = readout.eval(feed_dict={s : [s_t]})[0]
    a_t,action_index = play_action(epsilon,readout_t)

    ######### play ! ########
    s_t1,r_t,terminal = get_stage(s_t,a_t)

    # store the transition in D
    D.append((s_t, a_t, r_t, s_t1, terminal))

    s_t = s_t1
    rp += 1
    if(rp % 100 == 0):
        print("TRY PLAY and RECORD : %d max readout %.4f" % (rp,np.max(readout_t)))
```

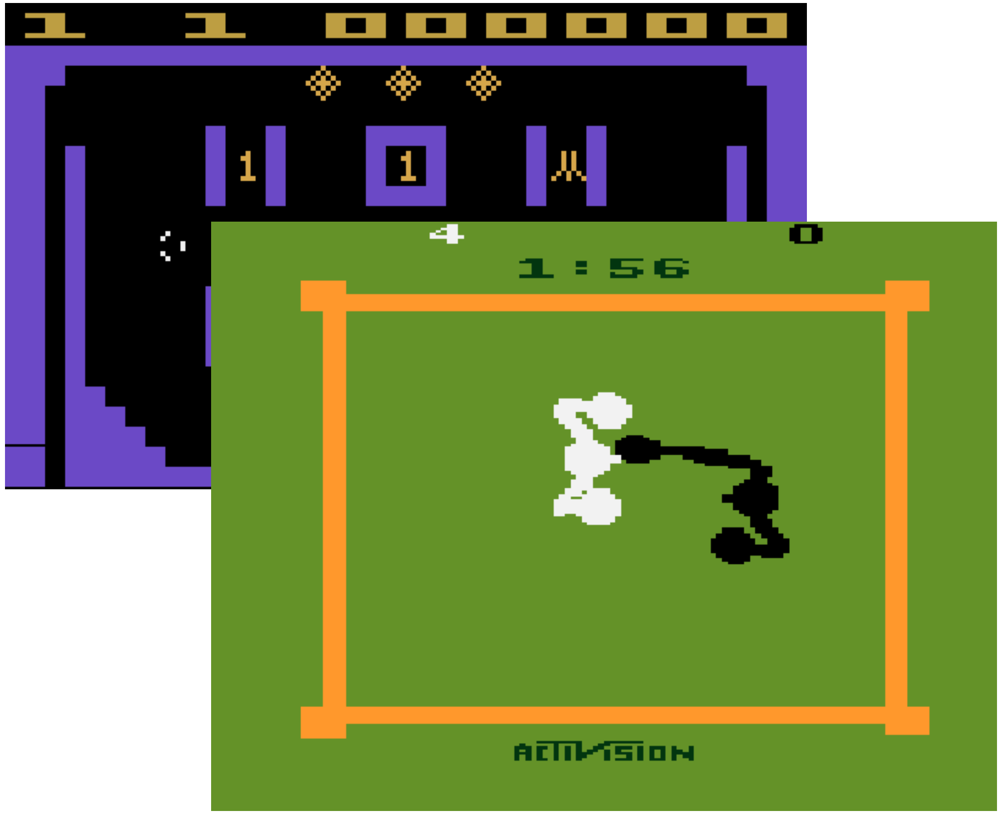
Learning!

```
# run the selected action and observe next state and reward
s_t1,r_t,terminal = get_stage(s_t,a_t)
# store the transition in D
D.append((s_t, a_t, r_t, s_t1, terminal))

if len(D) > REPLAY_MEMORY:
    D.popleft()
# sample a minibatch to train on
minibatch = random.sample(D, BATCH)

# get the batch variables
s_j_batch = [d[0] for d in minibatch]
a_batch = [d[1] for d in minibatch]
r_batch = [d[2] for d in minibatch]
s_j1_batch = [d[3] for d in minibatch]

y_batch = []
readout_j1_batch = readout.eval(feed_dict = {s : s_j1_batch})
for i in range(0, len(minibatch)):
    terminal = minibatch[i][4]
    # if terminal, only equals reward
    if terminal:
        y_batch.append(r_batch[i])
    else:
        y_batch.append(r_batch[i] + GAMMA * np.max(readout_j1_batch[i]))

# perform gradient step
train_step.run(feed_dict = {
    y : y_batch,
    a : a_batch,
    s : s_j_batch}
)

# update the old values
s_t = s_t1
rp += 1
```

Let skip to final!

Results Analysis

DQN is good at …



DQN is bad at …