Deep Reinforcement Learning (DRL)

Alvaro Soto

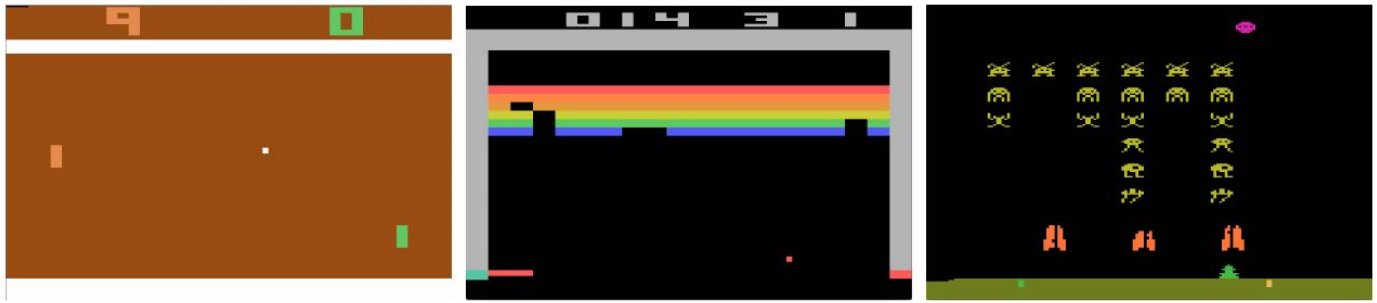
Computer Science Department (DCC), PUC

Intro

- A relevant challenge to RL techniques is how to represent value functions and/or policies.
- This is particularly relevant for the continous state case, where one can not learn a finite-state Q-table.
- Deep learning emerges as a highly flexible and powerful trainable tool to learn a Q-function.
- Unfortunately, in practice, classical RL techniques have been inestable when trained with non-linear learners such as NNs. The main difficulty is the lack of a proper label to train the models.
- The simultaneous challenge of building a Q-function and at the same time exploring the reward values from the environment, makes the learning task highly unestable.

DQN

- In 2014, Mnih et al. successfully train a deep convolutional NW to learn a Q-function for a player of classic Atari-2600 video games, such as Pong, Breakout, and Space Invaders.



- After training, the resulting model, called *Deep Q-NW* or DQN, outperforms all previous state-of-the-art techniques, achieving a performance level comparable to professional human gamers (Mnih et al. Nature 2015).

- DQN achieves this on a large list of 49 games, without relying on any game-specific modification, i.e., on each game it uses the same algorithm as well as NW architecture and hyperparameters. AWESOME!

Deep Q-Networks (DQN),
Mnih et al., NIPS-2014 and Mnih et al., Nature-2015.

Secret Ingredients:
Q-Learning + Deep Learning + Experience Replay Memory.

3 Secret Ingredients:
Q-Learning + Deep Learning + Experience Replay Memory.

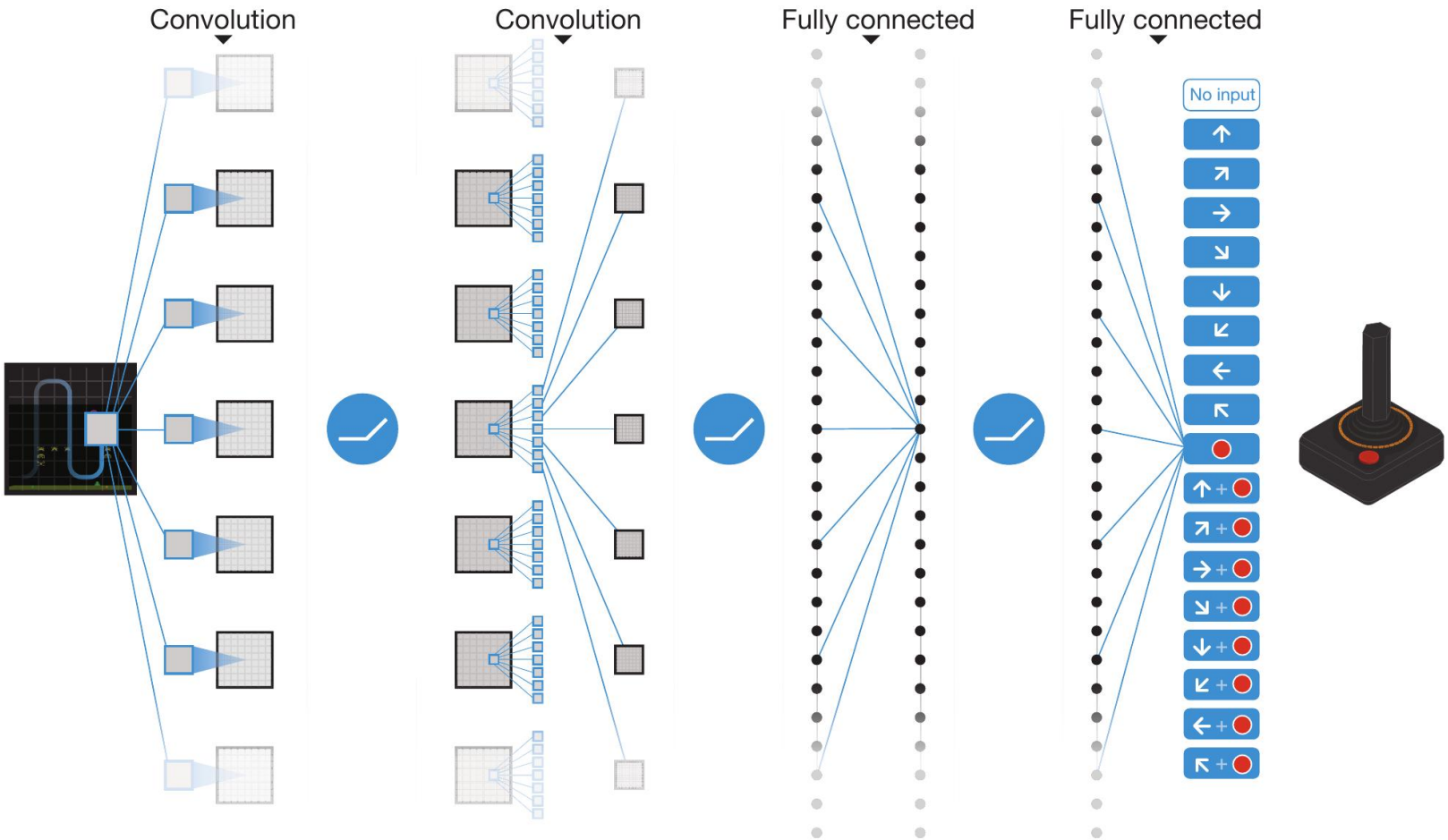
- As we discussed, Q-Learning is a model free and off-policy RL method to estimate the value function.
- Similarly to the value iteration algorithm, Q-Learning is based on Bellman's equation. It estimates the following function:

$$Q(s,a) = r(s,a) + \gamma V^*(s',a')$$
$$Q(s,a) = \mathbb{E}_{s' \sim env}[r(s,a) + \gamma \arg\max_{a'} Q(s',a')]$$

  where $\mathbb{E}$ denotes expected value ; (s', a') corresponds to next state s' and next action a' in environment *env*; and V^* is the optimal value function.
- In short, $Q(s,a)$ represents the maximum discounted cummulative rewards that can be achieve by starting from state s, executing action a, and following the optimal policy thereafter (maximizing value function).
- Let's see how we can use DL to learn a Q-function.

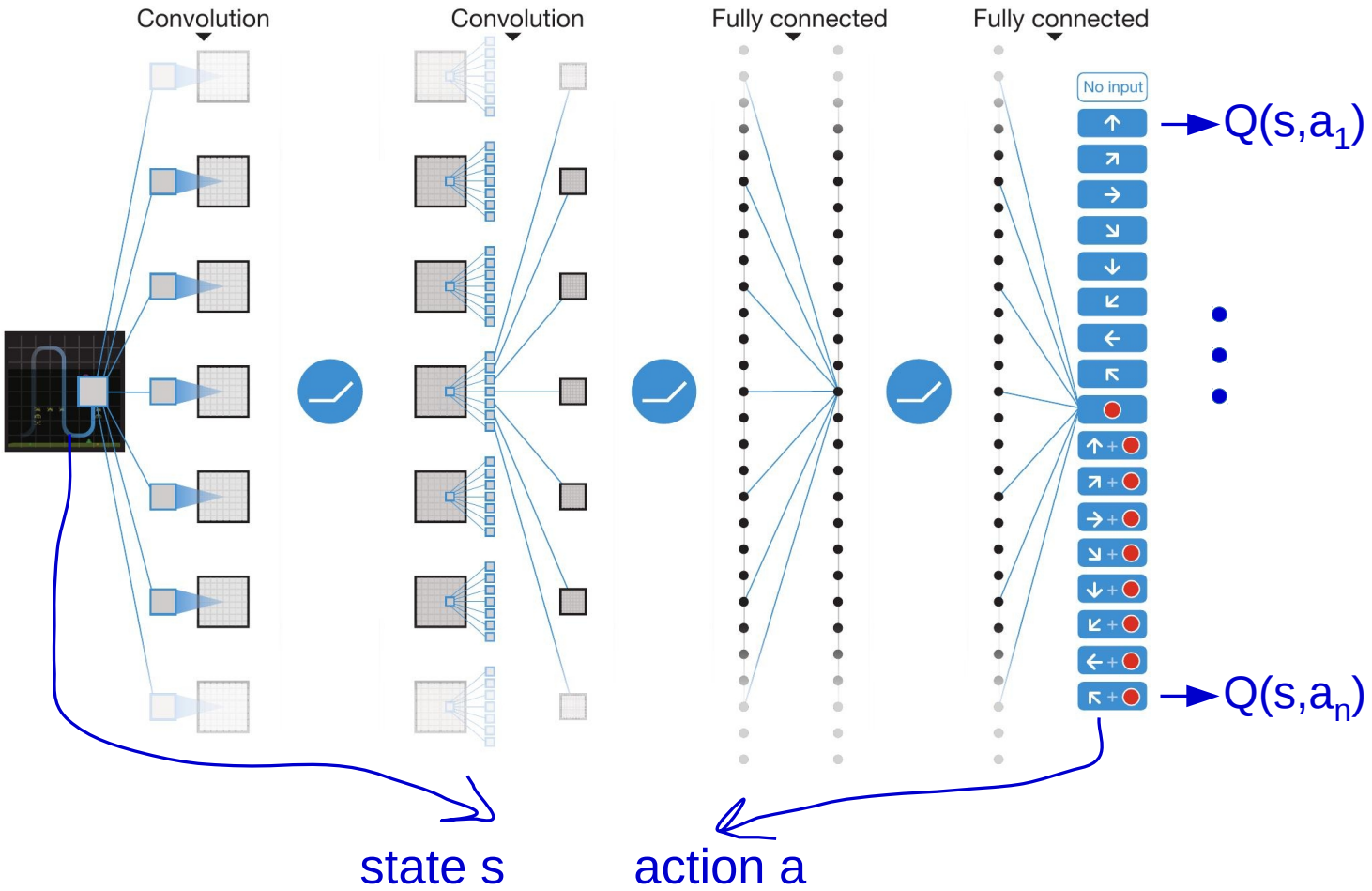
Secret Ingredients:
Q-Learning + Deep Learning + Experience Replay Memory.

- DQN approximates the Q-function using a DL architecture.

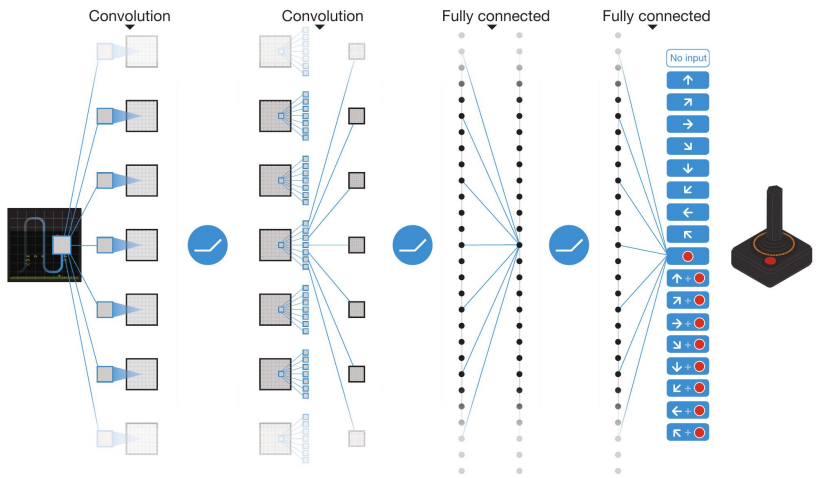


Secret Ingredients:
Q-Learning + Deep Learning + Experience Replay Memory.

This works by using the DL model to approximate or "read" the values of the Q-function: Q(s,a).



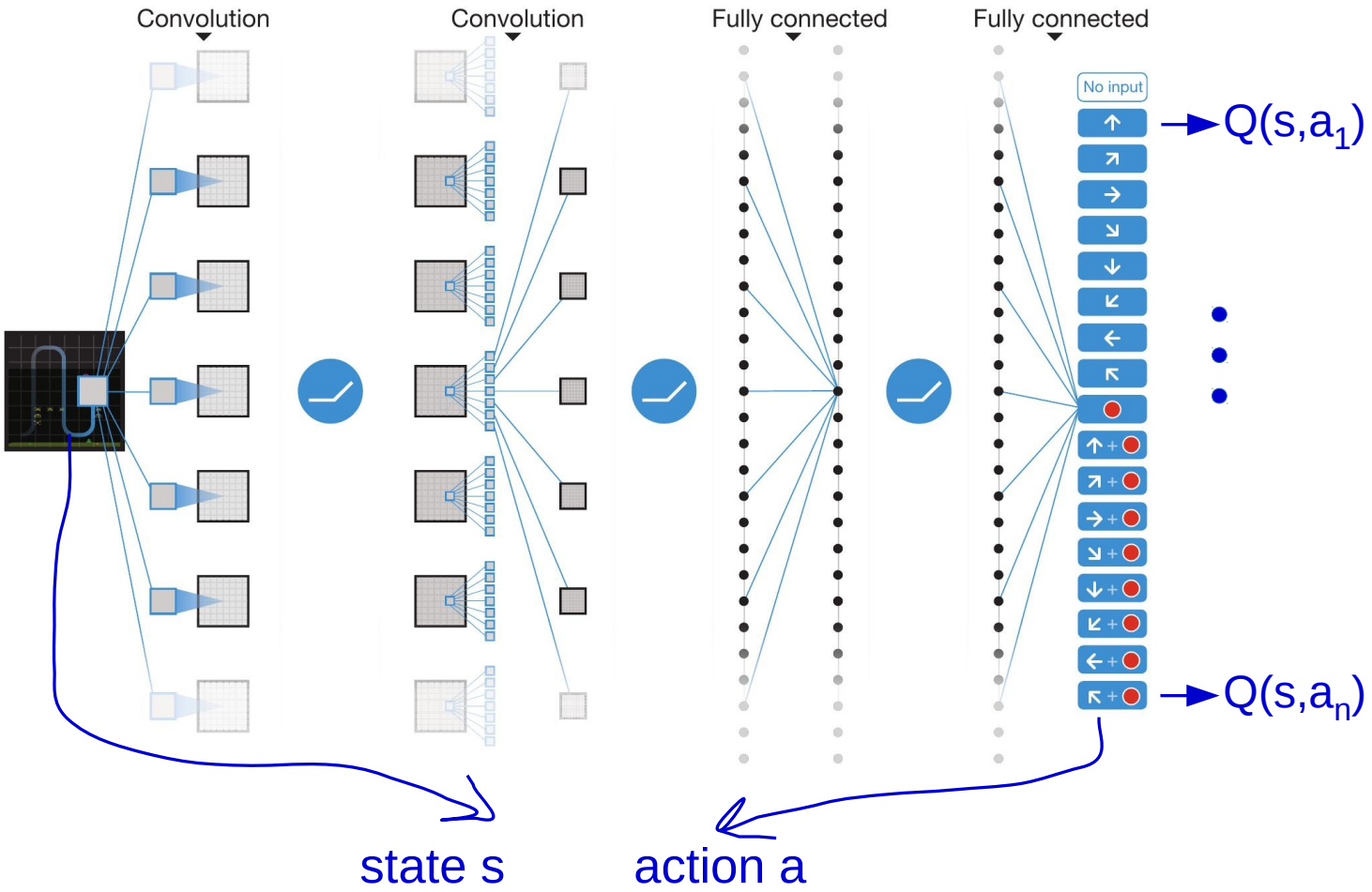
Secret Ingredients:
Q-Learning + Deep Learning + Experience Replay Memory.



Following Mnih et al. Nature 2015, NW structure is given by:

- Input: 84x84x4 image (they stack the 4 most recent frames to add temporal info).
- Layer-1 Convolutional: 32 filter, 8x8 with stride 4 + Relu.
- Layer-2 Convolutional: 64 filter, 4x4 filters with stride 2 + Relu.
- Layer-3 Convolutional: 64 filter, 3x3 filters with stride 1 + Relu.
- Layer-4 FC: 512 units + Relu (input is a flatted vector from convolutional layer-3: tensor of 64x7x7D is flatted to a vector of 3.136D).
- Output FC: linear layer with a single output for each valid action. Number of valid actions varied between 4 and 18 depending of the game.

Secret Ingredients:
Q-Learning + Deep Learning + Experience Replay Memory.



Problem: we do not know the right labels

Secret Ingredients:
Q-Learning + Deep Learning + Experience Replay Memory.

Training strategy: optimize NW parameters θ to improve at each iteration our estimation of the Q-function.

- At step i, we know the input state s_i and the executed action a_i. We also observe reward $r(s_i, a_i)$, next state s', and NW output $Q(s, a|\theta_i)$.
- The key strategy is to use the current NW estimation of the Q-function to estimate a dynamic target value y_i as follows:

$$y_i = \mathbb{E}_{s' \sim env}[r(s_i, a_i) + \gamma \arg\max_{a'} Q(s', a'|\theta_i^-)] \quad ,$$

  Notice that they use an estimation θ_i^- of the NW parameters to calculate $max_{a'} Q(s', a'|\theta_i^-)$. To avoid inestabilities, parameters θ_i^- are updated with NW parameters θ_i every C steps.
- This leads to the following loss function used to train the NW:

$$\mathbb{L}_i(\theta_i) = \mathbb{E}_{s,a \sim env}[(y_i - Q(s_i, a_i|\theta_i))^2]$$

- Note the dependency of the loss function $\mathbb{L}_i$ on index i. This is because target value y_i changes during training.
- For each training instance only output y_i corresponding to action a_i propagates an error function through the NW.

Secret Ingredients:
Q-Learning + Deep Learning + Experience Replay Memory.

- The gradient of the loss function with respect to the parameters θ_i is given by:

$$\nabla_{\theta_i}(\mathbb{L}_i(\theta_i)) = \nabla_{\theta_i}\left(\mathbb{E}_{s,a\sim env}[(y_i - Q(s,a|\theta_i))^2]\right)$$

$$\nabla_{\theta_i}(\mathbb{L}_i(\theta_i)) = \mathbb{E}_{s,a,s'\sim env}\left[\left(r(s,a) + \gamma \arg\max_{a'} Q(s',a'|\theta_i^-) - Q(s,a|\theta_i)\right)\nabla_{\theta_i}(Q(s,a|\theta_i))\right]$$

- Rather than computing the full expectations, one can optimize the parameters of the NW using the current estimations.
- Actions are selected according to a ϵ-greedy strategy, i.e., select random action with probability ϵ or a greedy action with probability $1-\epsilon$.

Secret Ingredients:
Q-Learning + Deep Learning + Experience Replay Memory.

- In classical RL, it is known that learning the Q-function with non-linear models often diverge.
- DQN avoids this problem by using an Experience Replay Memory.
- DQN stores the agent's experiences at each time-step, $e_t = (s_t, r_t, a_t, s_{t+1})$ in a data set $D = \{e_1, \ldots, e_t\}$, pooled over many episodes (ex. 1M memories).
- During each training minibatch, DQN samples training data uniformly from D.
- These scheme offers several advantages:
  - Experience is potentially used in many weight updates.
  - Learning from consecutive samples is inefficient, owing to the strong correlations between the samples; randomizing the samples breaks these correlations and therefore reduces the variance of the updates.
  - The behaviour distribution is averaged over many of its previous states, smoothing out learning and avoiding oscillations or divergence in the parameters.

Secret Ingredients:
Q-Learning + Deep Learning + Experience Replay Memory.

Further training tricks

- Use of parameters θ_i^- to generate y_i: every C updates, they clone NW Q to obtain $\hat{Q}$ and use $\hat{Q}$ for generating y_i for the next C updates of function Q. This modification makes the algorithm more stable compared to standard online Q-learning.
- Generating the targets using an older set of parameters adds a delay between the time an update to Q is made and the time the update affects the targets y_i, making divergence oscillations much more unlikely.
- They clip the values of the loss function $\mathbb{L}$ to the range -1 and 1. This further improves the stability of the algorithm.

DQN.

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N — N in the order of 1M

Initialize action-value function Q with random weights θ

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 ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$ — e-greedy policy

    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 $\left(\phi_t, a_t, r_t, \phi_{t+1}\right)$ in D — keep updating Reply Memory D

    Sample random minibatch of transitions $\left(\phi_j, a_j, r_j, \phi_{j+1}\right)$ 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 θ

    Every C steps reset $\hat{Q} = Q$

  End For

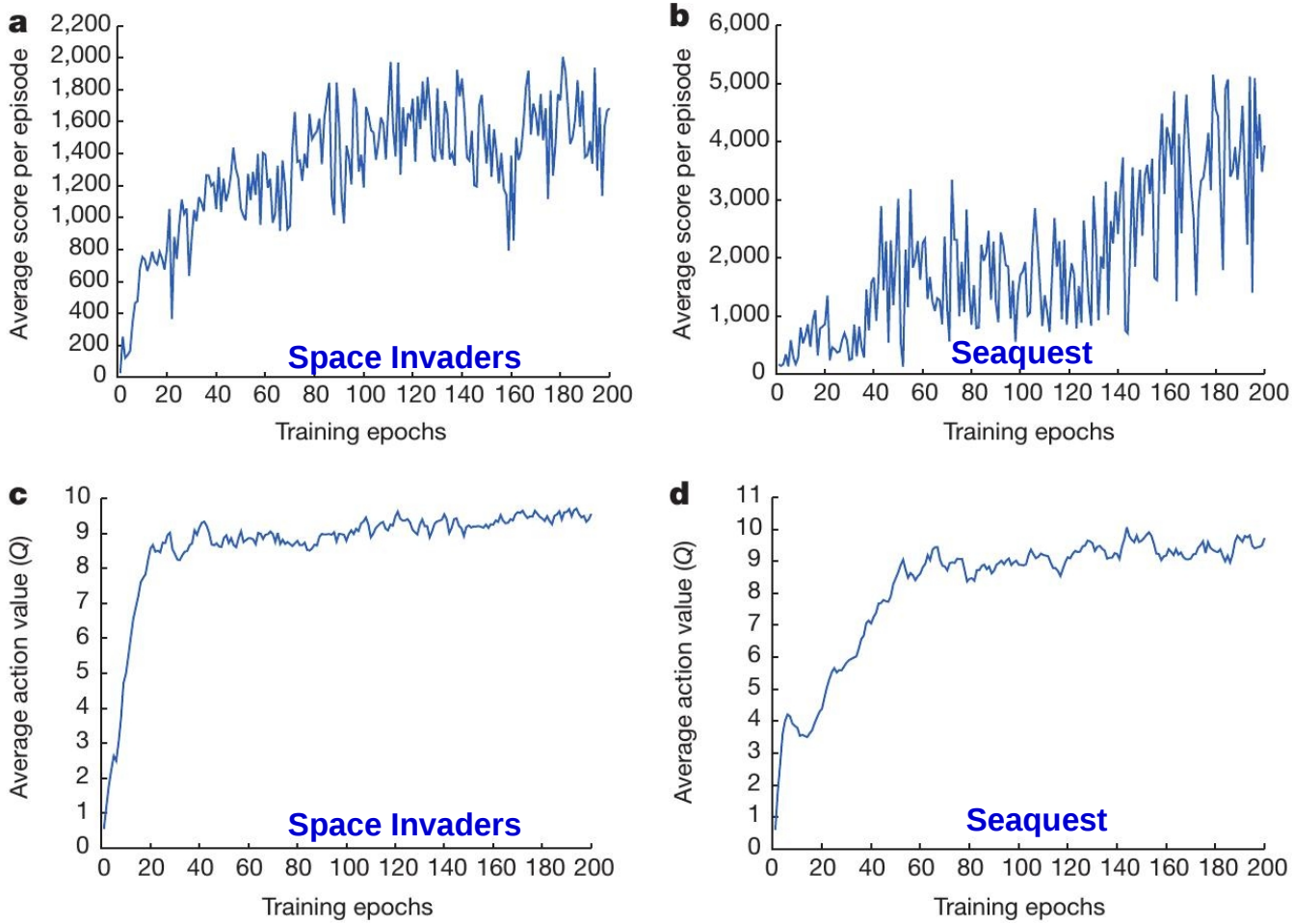
End For

Implementation

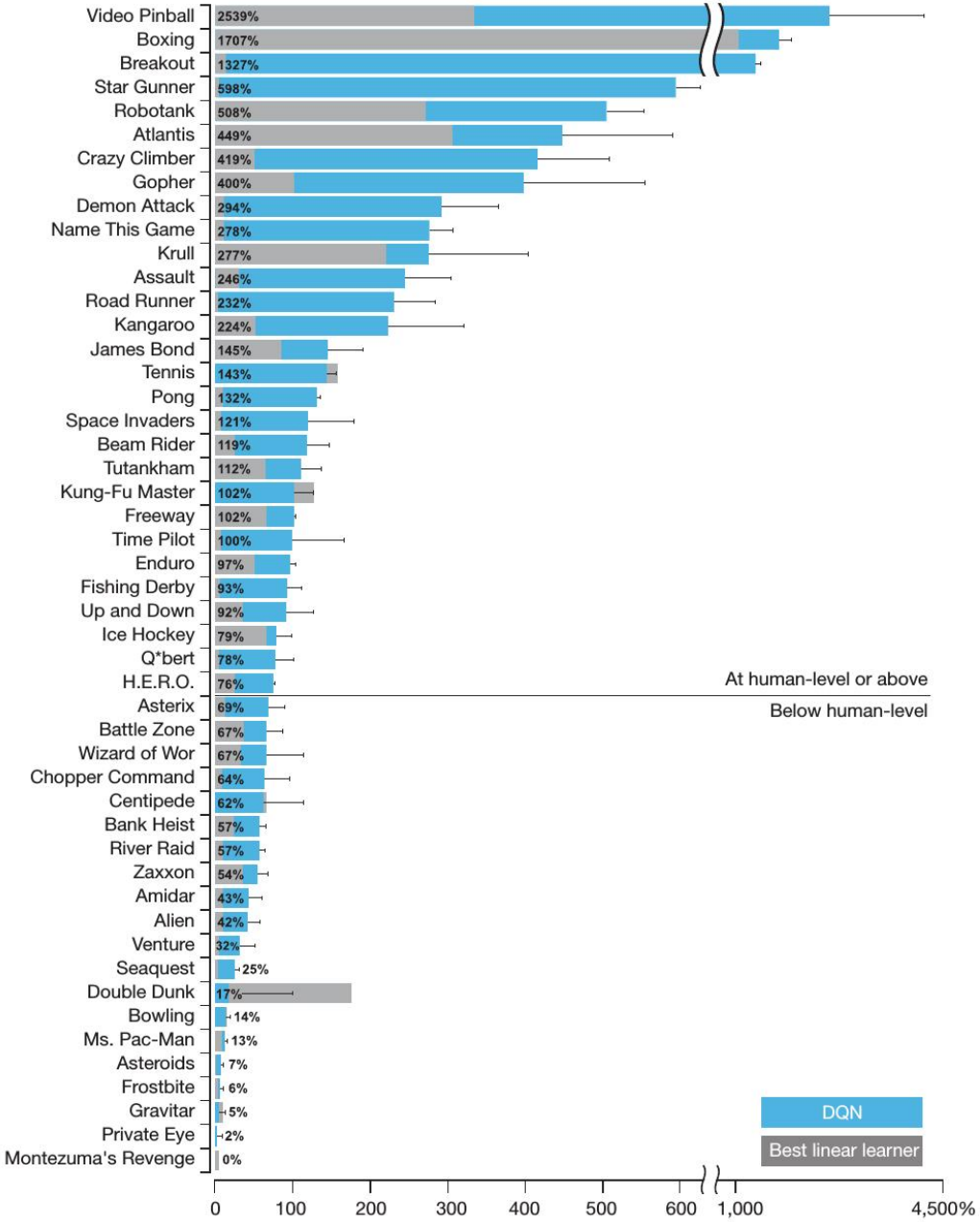
- In practice, they start by choosing actions uniformly at random for the first T time steps (ex. T=500.000), without updating the network weights. This allows the system to populate the replay memory before training begins.
- They use frame-skipping, i.e., agent sees and selects actions on every kth frame instead of every frame, and its last action is repeated on skipped frames. They use k=4. Since running the emulator forward for one step requires much less computation than having the agent select an action, this technique allows the agent to play roughly k times more games without signicantly increasing the runtime.
- Minibatches of size 32, discount factor 0.99, train with RMSProp.
- e-greedy annealed linearly from 1.0 to 0.1 over the first million frames, and fixed at 0.1 thereafter.
- Total of 50 M training frames (this is around 38 days of game experience in total).
- Replay memory 1M most recent frames.
- One epoch corresponds to 50000 minibatch weight updates, roughly 30 minutes of training time.
- Normalize reward signal across games according if score changed from one time step to the next: +1 whenever it increased, 1 whenever it decreased, and 0 otherwise.

Hyperparameter	Value	Description
minibatch size	32	Number of training cases over which each stochastic gradient descent (SGD) update is computed.
replay memory size	1000000	SGD updates are sampled from this number of most recent frames.
agent history length	4	The number of most recent frames experienced by the agent that are given as input to the Q network.
target network update frequency	10000	The frequency (measured in the number of parameter updates) with which the target network is updated (this corresponds to the parameter *C* from Algorithm 1).
discount factor	0.99	Discount factor gamma used in the Q-learning update.
action repeat	4	Repeat each action selected by the agent this many times. Using a value of 4 results in the agent seeing only every 4th input frame.
update frequency	4	The number of actions selected by the agent between successive SGD updates. Using a value of 4 results in the agent selecting 4 actions between each pair of successive updates.
learning rate	0.00025	The learning rate used by RMSProp.
gradient momentum	0.95	Gradient momentum used by RMSProp.
squared gradient momentum	0.95	Squared gradient (denominator) momentum used by RMSProp.
min squared gradient	0.01	Constant added to the squared gradient in the denominator of the RMSProp update.
initial exploration	1	Initial value of ε in ε-greedy exploration.
final exploration	0.1	Final value of ε in ε-greedy exploration.
final exploration frame	1000000	The number of frames over which the initial value of ε is linearly annealed to its final value.
replay start size	50000	A uniform random policy is run for this number of frames before learning starts and the resulting experience is used to populate the replay memory.
no-op max	30	Maximum number of "do nothing" actions to be performed by the agent at the start of an episode.

Results


a
Average score per episode
2,200
2,000
1,800
1,600
1,400
1,200
1,000
800
600
400
200
0
0 20 40 60 80 100 120 140 160 180 200
Training epochs
Space Invaders
b
Average score per episode
6,000
5,000
4,000
3,000
2,000
1,000
0
0 20 40 60 80 100 120 140 160 180 200
Training epochs
Seaquest
c
Average action value (Q)
10
9
8
7
6
5
4
3
2
1
0
0 20 40 60 80 100 120 140 160 180 200
Training epochs
Space Invaders
d
Average action value (Q)
11
10
9
8
7
6
5
4
3
2
1
0
0 20 40 60 80 100 120 140 160 180 200
Training epochs
Seaquest

Results

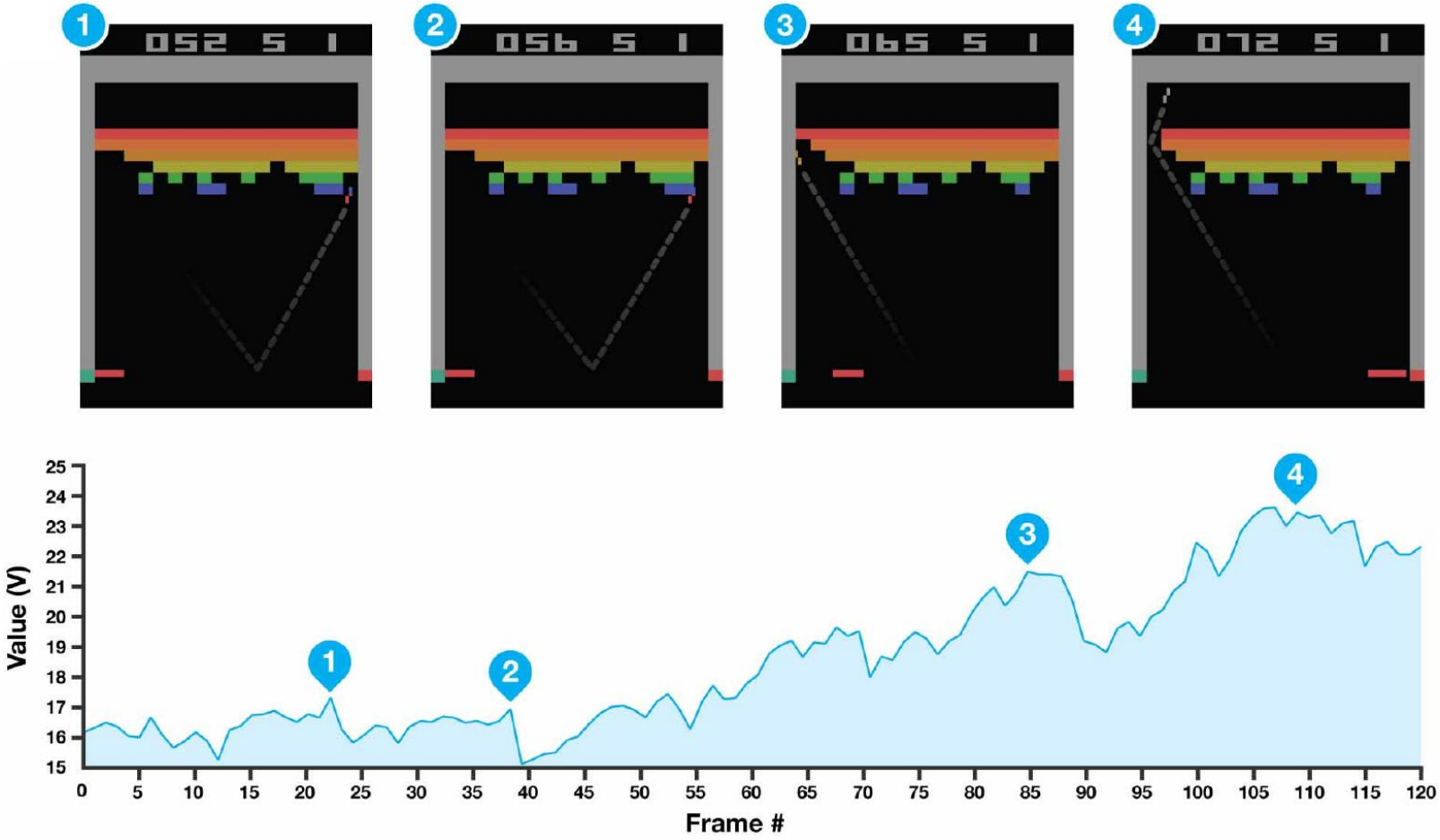

Video Pinball 2539%
Boxing 1707%
Breakout 1327%
Star Gunner 598%
Robotank 508%
Atlantis 449%
Crazy Climber 419%
Gopher 400%
Demon Attack 294%
Name This Game 278%
Krull 277%
Assault 246%
Road Runner 232%
Kangaroo 224%
James Bond 145%
Tennis 143%
Pong 132%
Space Invaders 121%
Beam Rider 119%
Tutankham 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%
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%
At human-level or above
Below human-level
DQN
Best linear learner
0
100
200
300
400
500
600
1,000
4,500%

Results

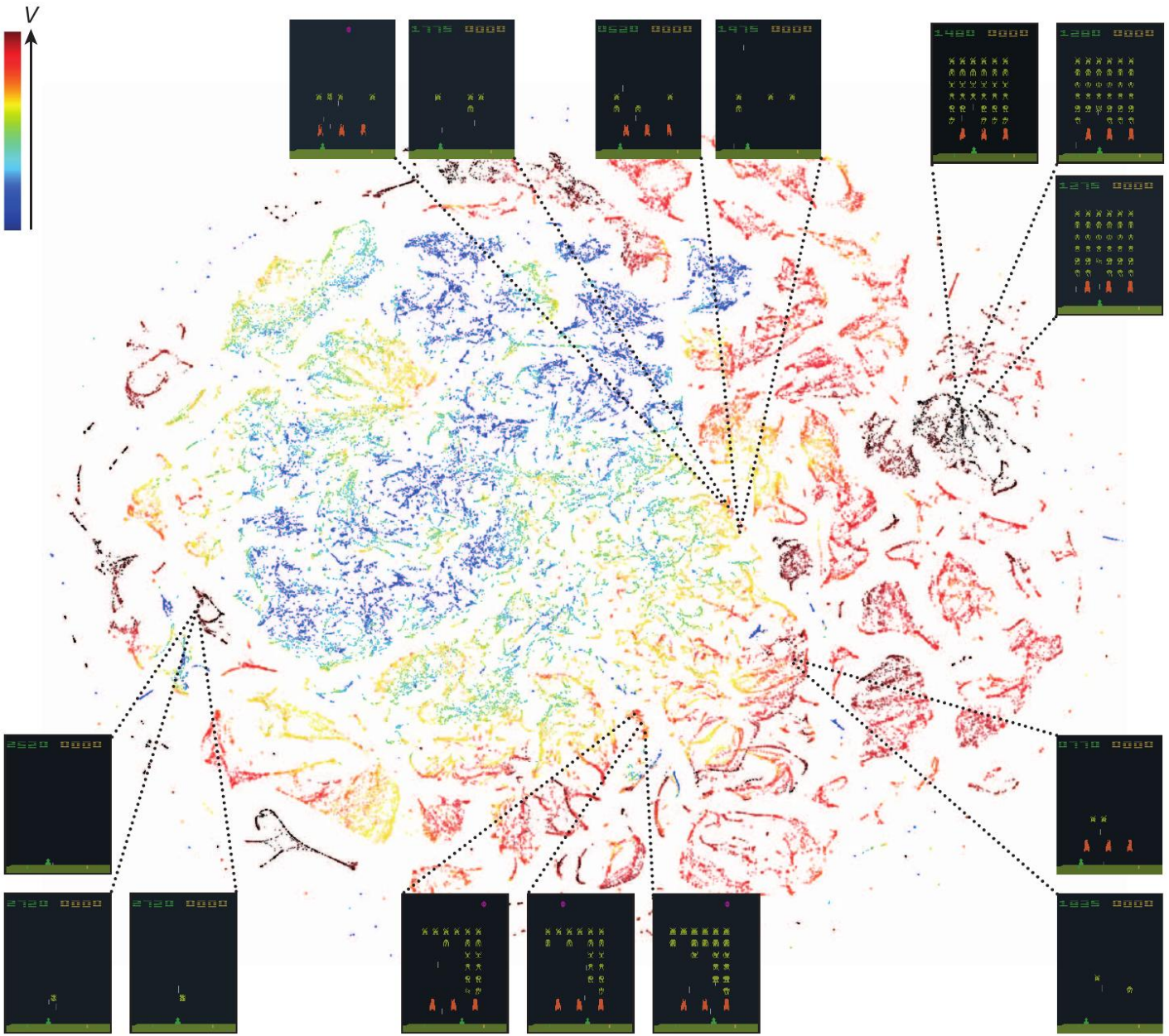
Table 3 | The effects of replay and separating the target Q-network

Game	With replay, with target Q	With replay, without target Q	Without replay, with target Q
Breakout	316.8	240.7	10.2
Enduro	1006.3	831.4	141.9
River Raid	7446.6	4102.8	2867.7
Seaquest	2894.4	822.6	1003.0
Space Invaders	1088.9	826.3	373.2

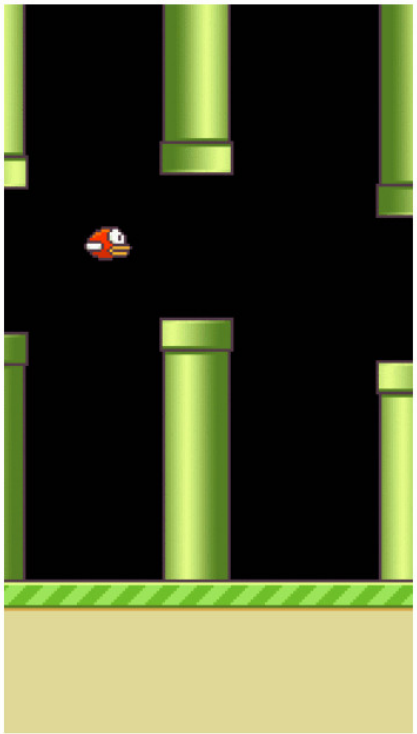
Results


1
052 5 1
2
056 5 1
3
065 5 1
4
072 5 1
Value (V)
25
24
23
22
21
20
19
18
17
16
15
0
5
10
15
20
25
30
35
40
45
50
55
60
65
70
75
80
85
90
95
100
105
110
115
120
Frame #

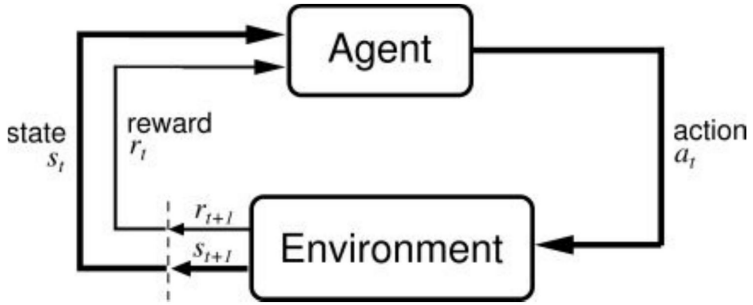
Results: t-SNE Visualization



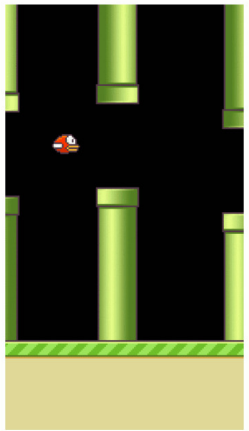
Example: DQN for Flappy Bird



Part of this material is from Ben Lau:
https://yanpanlau.github.io/2016/07/10/FlappyBird-Keras.html



State =



$$\text{Reward} = \begin{cases} 0.1, & \text{if bird still alive.} \\ +1, & \text{if bird pass a pipe.} \\ -1, & \text{if bird die.} \end{cases}$$

$$\text{Action} = \begin{cases} [1,0], & \text{Flap.} \\ [0,1], & \text{Do nothing.} \end{cases}$$

Deep Q-Networks (DQN),
Mnih et al., NIPS-2014; Mnih et al., Nature-2015.

Secret Ingredients:
Q-Learning + Deep Learning + Experience Replay Memory.

CNN Model Definition

- Input: 84x84x4 images
- Layer-1 Convolutional: 32 filter, 8x8 with stride 4 + Relu.
- Layer-2 Convolutional: 64 filter, 4x4 filters with stride 2 + Relu.
- Layer-3 Convolutional: 64 filter, 3x3 filters with stride 1 + Relu.
- Layer-4 FC: 512 units + Relu
- Output FC: linear layer with a single output for each valid action.

```
model = Sequential()
model.add(Convolution2D(32,8,8, subsample=(4,4),
          border_mode="same",input_shape=(84,84,4)))
model.add(Activation("relu"))
model.add(Convolution2D(64,4,4,subsample=(2,2),border_mode="same"))
model.add(Activation("relu"))
model.add(Convolution2D(64,3,3, subsample=(1,1),
          border_mode="same"))
model.add(Activation("relu"))
model.add(Flatten())
model.add(Dense(512))
model.add(Activation("relu"))
model.add(Dense(2))

adam = Adam(lr=LEARNING_RATE)
model.compile(loss='mse',optimizer=adam)
```

Deep Q-Networks (DQN),
Mnih et al., NIPS-2014; Mnih et al., Nature-2015.

Secret Ingredients:
Q-Learning + Deep Learning + Experience Replay Memory.

Fill Replay Memory

```
from collections import deque
D = deque()
#select and apply initial action
a_t = [1,0]
x_t, r_0, terminal = game_state.frame_step(a_t)
x_t = preProcessImage(x_t) #convert to gray scale and scale
s_t = np.stack((x_t, x_t, x_t, x_t), axis=2)
s_t = s_t.reshape(1, s_t.shape[0], s_t.shape[1], s_t.shape[2])

for _ range(FILL_MEM):
    a_t[random.randrange(NUM_ACTIONS)] = 1  #select action

    #run the selected action and observed next state and reward
    x_t1_colored, r_t, terminal = game_state.frame_step(a_t)

    #Process image
    x_t1 = x_t = preProcessImage(x_t1)
    x_t1 = x_t1.reshape(1, x_t1.shape[0], x_t1.shape[1], 1) #1x84x84x1
    s_t1 = np.append(x_t1, s_t[:, :, :, :3], axis=3)

    # store the last state in replay memory D
    D.append((s_t, action_index, r_t, s_t1, terminal))
    s_t = s_t1, t = t + 1
```

Deep Q-Networks (DQN),
Mnih et al., NIPS-2014; Mnih et al., Nature-2015.

Secret Ingredients:
Q-Learning + Deep Learning + Experience Replay Memory.

Learning

```
batch = random.sample(D, BATCH)
inputs = np.zeros((BATCH, s_t.shape[1], s_t.shape[2], s_t.shape[3]))
targets = np.zeros((inputs.shape[0], ACTIONS))

#Now we do the experience replay
for i in range(0, len(batch)):
    state_t=batch[i][0],action_t=batch[i][1],reward_t=batch[i][2]
    state_t1=batch[i][3],terminal=batch[i][4]
    inputs[i:i + 1] = state_t

    #Get current NW estimate for Q-function.
    targets[i] = model.predict(state_t)

    #Get target: y_i= r(s,a) + gamma*argmax{a_t1}Q(s_t1,a_t1)
    Q_sa = model.predict(state_t1)
    if terminal: #if eposide ends
        targets[i, action_t] = reward_t
    else:
        targets[i, action_t] = reward_t + GAMMA * np.max(Q_sa)
    #Train network using batch
loss += model.train_on_batch(inputs, targets)
#Get ready for next iteration
s_t = s_t1
t = t + 1
```