CS6482 Deep RL

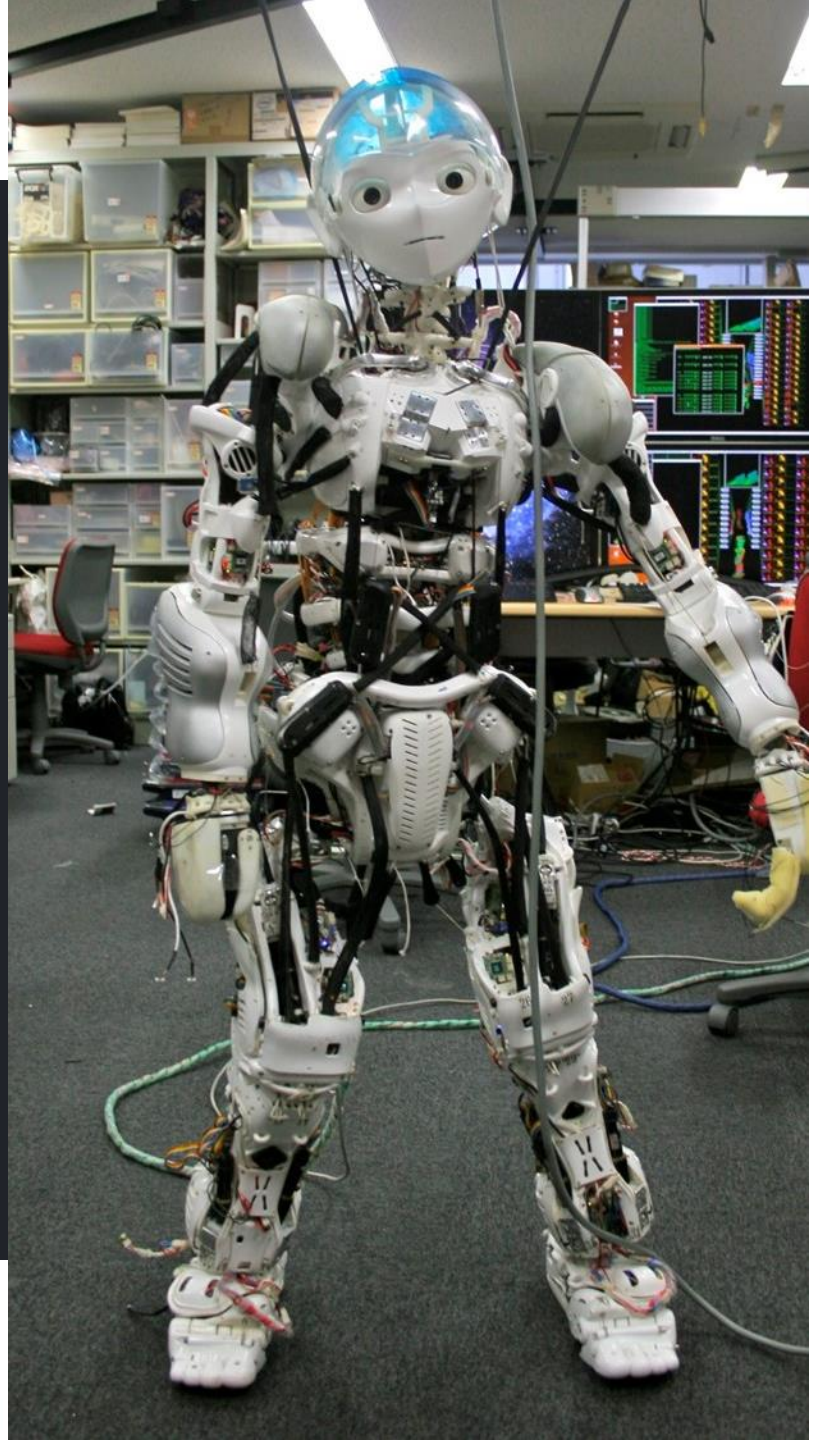
H: DQN for Atari

J.J. Collins

Dept. of CSIS

University of Limerick

UNIVERSITY OF LIMERICK
OLLSCOIL LUIMNIGH



Objectives

- Describe the architecture and loss function used in DQN Atari
- Understand the DQN Atari algorithm that uses Double DQNs
- Evaluate the performance of DQN Atari
- Work through DQN coding fragments

CS4287

Outline

Based on excerpts from:

1. Mnih, V., Kavukcuoglu, K., Silver, D. *et al.* Human-level control through deep reinforcement learning. *Nature* 518, 529–533 (2015)
   See demo in Supplementary Material at end of page:
   https://www.nature.com/articles/nature14236#citeas
   https://www.youtube.com/watch?v=fevMOp5TDQs

2. Chapter 11 in Gulli, Kapoor, and Pal. Deep Learning with TensorFlow 2 and Keras, 2nd Ed. Packt Birmingham. 2020.

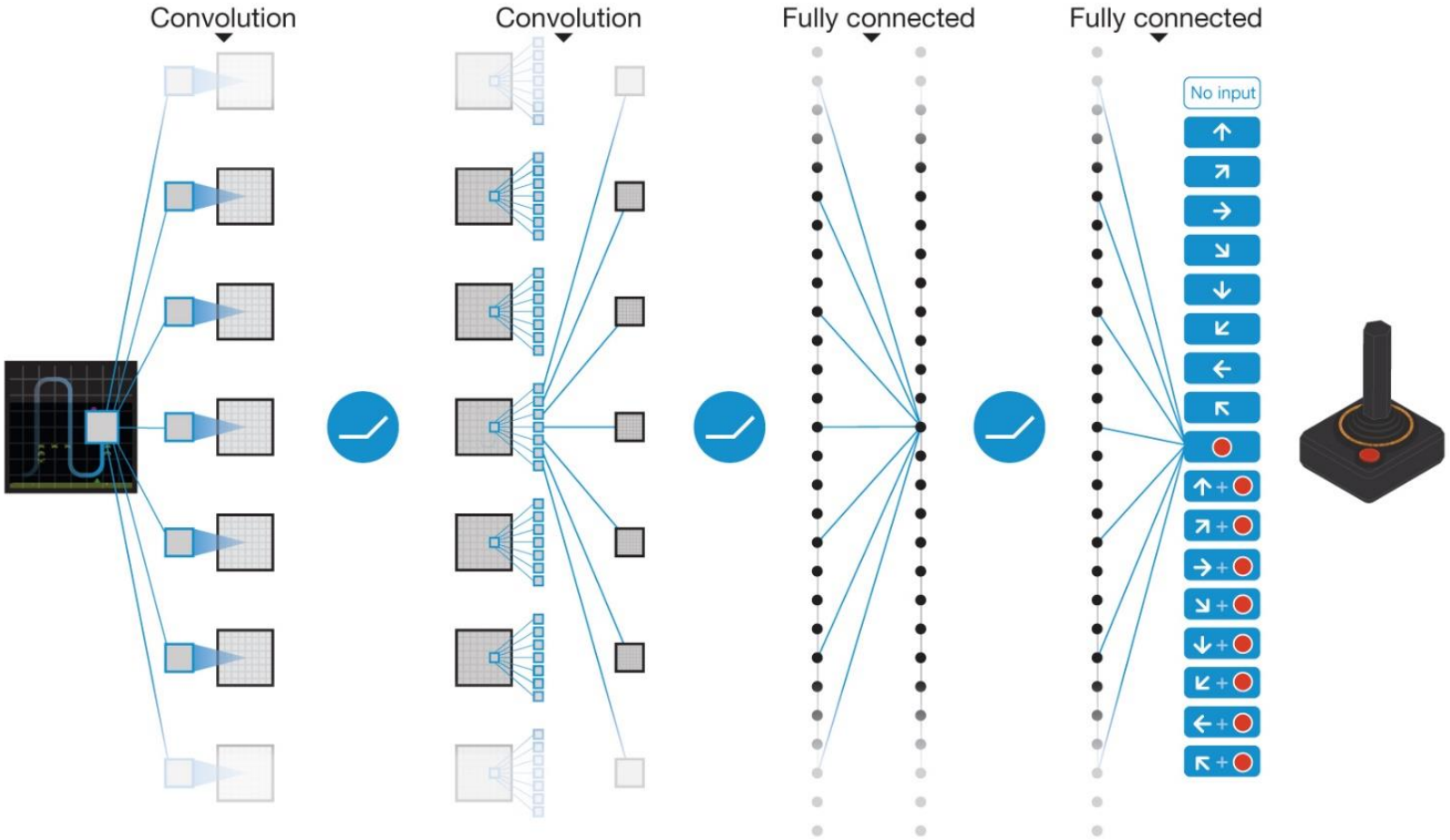
Q update for windy grid world

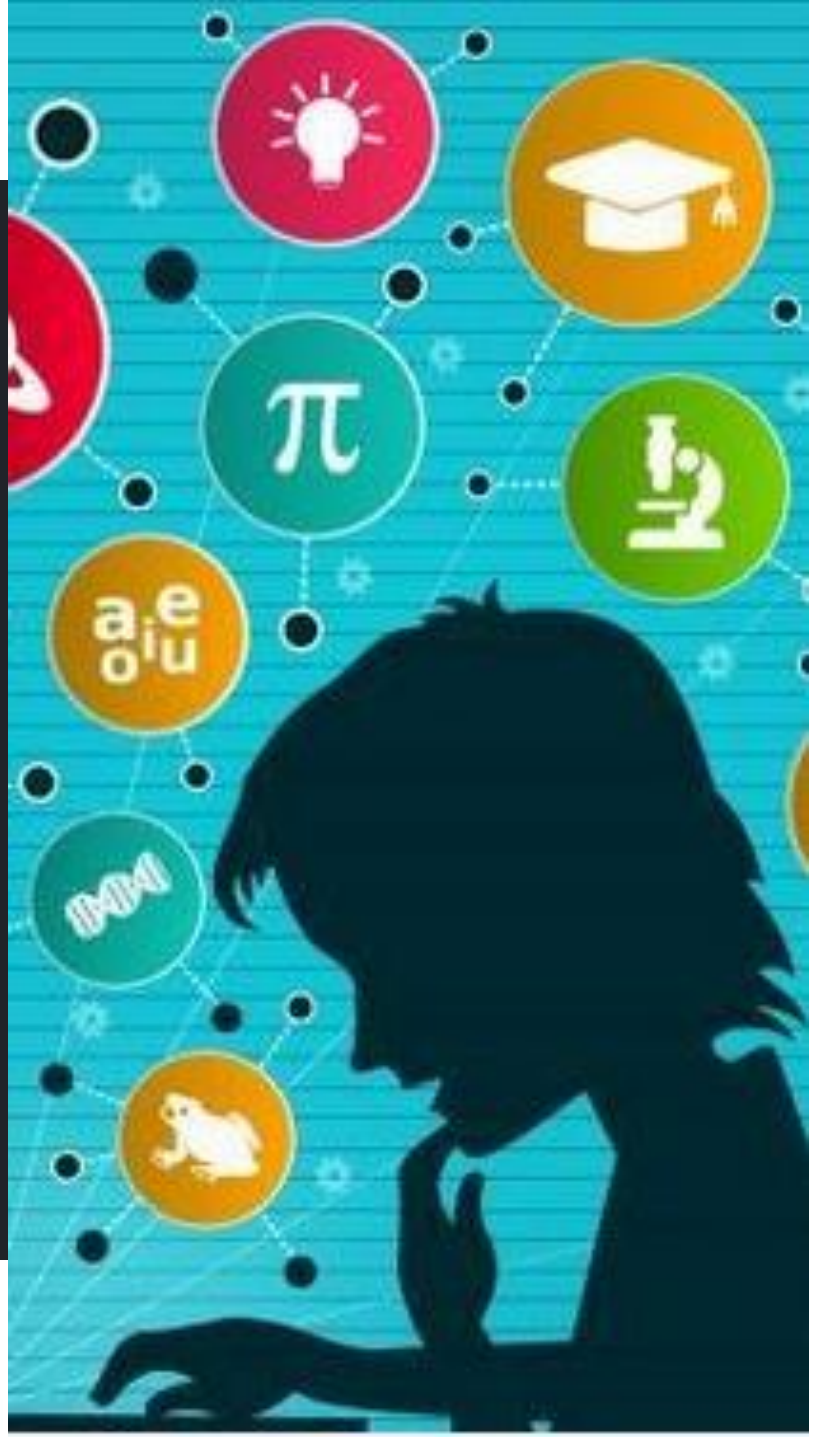
State	Q(Cell1,North)	Q(Cell1,North)	Q(Cell1,East)	Q(Cell1,West)
Cell 1	25	25	40	10
Cell 2	20	20	25	35

- $Q(s,a) = Q(s,a) + \alpha\left[r + \gamma max_{a'}\left(Q(s',a')\right) - Q(s,a)\right]$
- Agent is in cell 1
- Selects maximising action move east results in a transition to cell 2, and receives intermediate reward 0
- $Q(Cell1, East) = 40 + \alpha[0 + \gamma 35 - 40]$
- Let discount = 1 and step size = 0.5
- $Q(Cell1, East) = 40 + 0.5[35 - 40]$
- Q(Cell1, East) = 37.5

DQN Atari: Architecture

- V Mnih *et al. Nature* **518**, 529-533 (2015)





DQN Architecture

Input → 84 × 84 × 4 image extracted from raw input

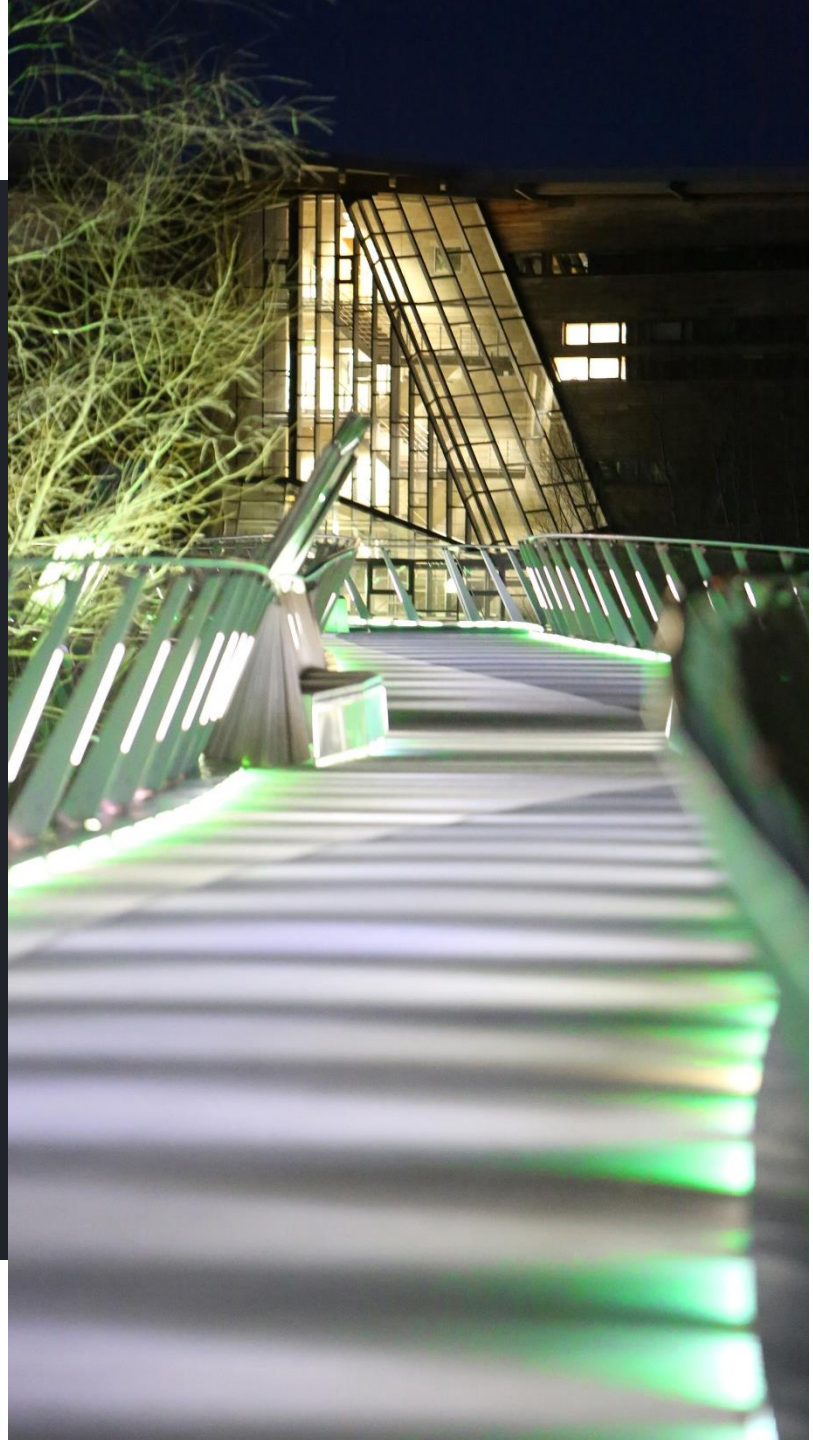
The first hidden layer convolves 32 filters of 8 × 8 with stride 4 and applies a rectifier nonlinearity.

The 2nd hidden layer convolves 64 filters of 4 × 4 with stride 2, followed by a rectifier nonlinearity.

The 3rd convolutional layer that convolves 64 filters of 3 × 3 with stride 1 followed by a rectifier.

The final hidden layer is fully-connected and consists of 512 rectifier units.

The output layer is a fully-connected linear layer with a single output for each valid action



Recap

Machine Learning Update to Model:

- **NewEstimate = OldEstimate + StepSize [Target – OldEstimate]**
- Where
- **Error = [Target – OldEstimate]**

- Sample Averaging in intro to RL
  - $Q_{n+1} = Q_n + \frac{1}{n}[R_n - Q_n]$

- TD Update
  - $V(s_{t+1}) \leftarrow V(s_t) + \alpha[\boldsymbol{R_{t+1}} + \boldsymbol{\gamma V(s_{t+1})} - V(s_t)]$
  - Where the target is $R_{t+1} + \gamma V(s_{t+1})$
  - The error is $\delta = [R_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$
  - Likewise for Q

The Loss Function

Parameters in the network at iteration i are θ_i

Optimal target → $r + \gamma \, max_{a'} Q^*(s', a')$

Substituted by approximate target → $r + \gamma \, max_{a'} Q(s', a'; \theta_i^-)$

where θ_i^- are the parameters from a previous iteration used to compute the target at iteration i

The Loss function

$$L_i(\theta_i) = \mathrm{E}_{(s,a,r,s')}\left[\left(r + \gamma \, max_{a'} Q(s', a'; \theta_i^-) - Q(s, a, ; \theta_i)\right)^2\right]$$

where $\theta_i^- = \theta_{i-1}$

The Q update can be achieved by updating weights after every iteration using single samples and optimising L using stochastic gradient descent

DQN Atari: the Algorithm using Experience Replay

V Mnih *et al.*

***Nature* 518, 529-533 (2015)**

Experience $e_t = (s_t, a_t, r_t, s_{t+1})$

$D_t = \{e_1, e_2, \ldots e_t\}$

Learning performed on samples of experience (minibatches).

$\phi_t = \phi(s_t)$ is the preprocessing applied to the raw image

An example of Double DQN

One for action selection

One to learn the target

Initialize replay memory D to capacity N
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 $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    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 θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

DQN – Stacked Frames

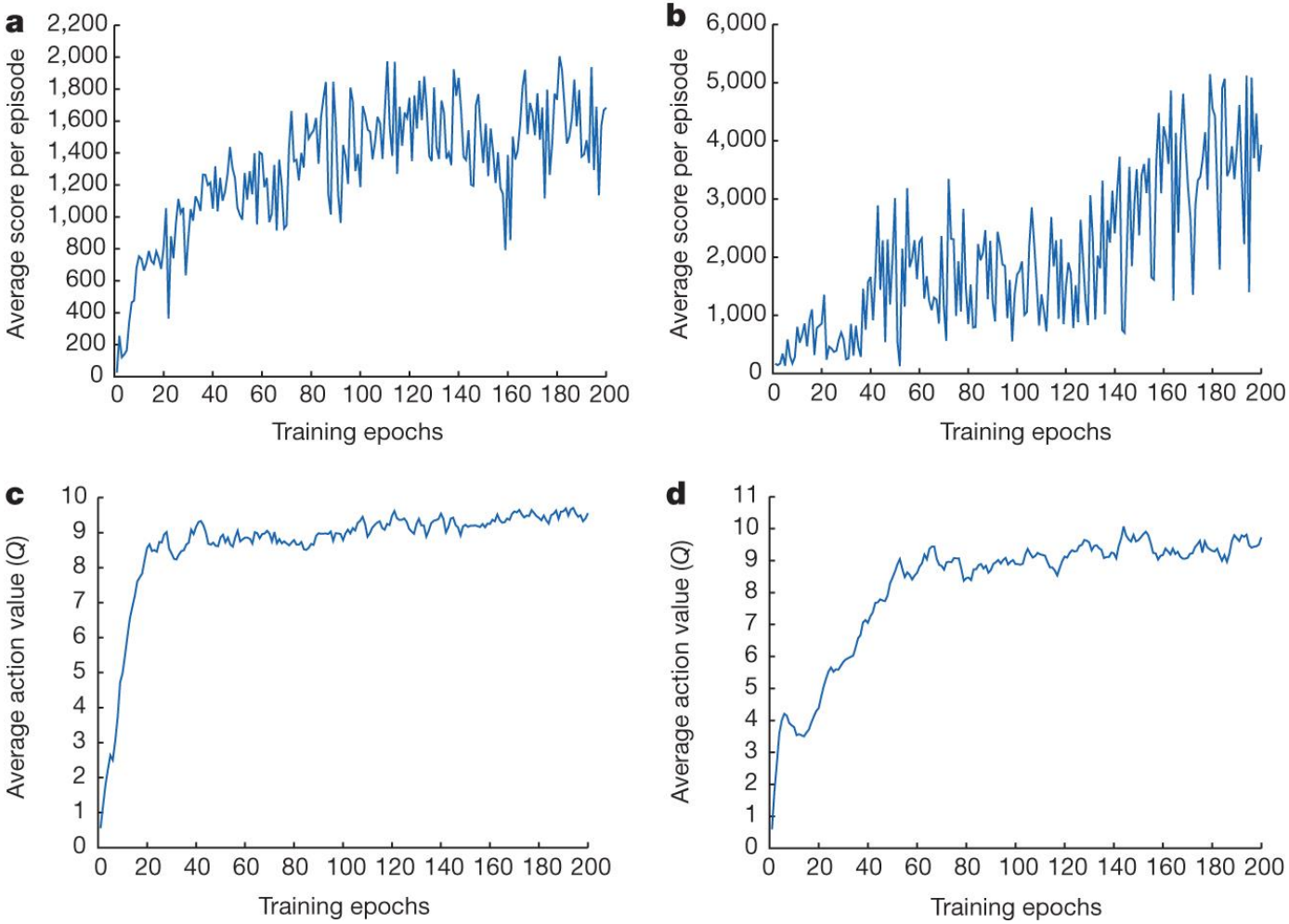
In these experiments, we used the RMSProp algorithm with minibatches of size 32. The behavior policy during training was ϵ-greedy with ϵ annealed linearly from 1 to 0.1 over the first million frames, and fixed at 0.1 thereafter. We trained for a total of 10 million frames and used a replay memory of one million most recent frames.

Following previous approaches to playing Atari games, we also use a simple frame-skipping technique [3]. More precisely, the agent sees and selects actions on every k^{th} frame instead of every frame, and its last action is repeated on skipped frames. 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 significantly increasing the runtime. We use $k = 4$ for all games except Space Invaders where we noticed that using $k = 4$ makes the lasers invisible because of the period at which they blink. We used $k = 3$ to make the lasers visible and this change was the only difference in hyperparameter values between any of the games.

- Mnih, Volodymyr & Kavukcuoglu, Koray & Silver, David & Graves, Alex & Antonoglou, Ioannis & Wierstra, Daan & Riedmiller, Martin. (2013). Playing Atari with Deep Reinforcement Learning.

DQN Atari: Average Score and Q

- V Mnih *et al. Nature* **518**, 529-533 (2015)
- Space Invaders: a and c
- Seaquest: b and d



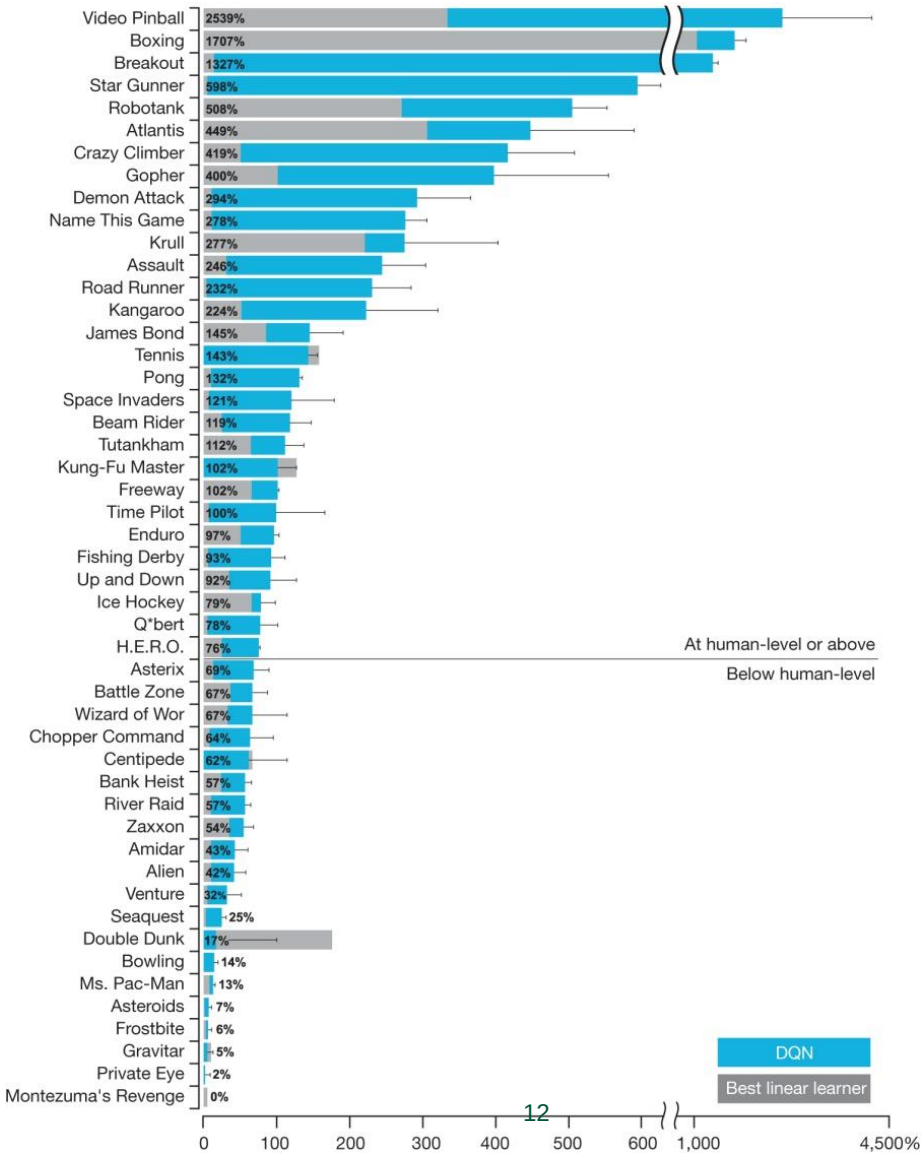
UNIVERSITY OF LIMERICK
OLLSCOIL LUIMNIGH

DQN Atari: Ranking

V Mnih *et al. Nature* **518**, 529-533 (2015)

Comparison with best published alternative RL algorithms

The performance of DQN is normalized with respect to a professional human games tester (100% level) and random play (0% level).



DQN Atari: Coding Fragments

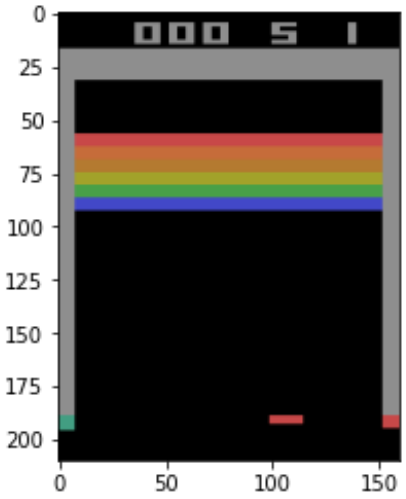
- From Gulli et al.
- Similar to sample code for CartPole-v0 except for model and experience (data)

```
MONITOR = TRUE
class DQN():

        def __init__(self, env_string,batch_size=64, IM_SIZE = 84, m = 4):

        if MONITOR:self.env = gym.wrappers.Monitor(self.env, '../data/'+env_string, force=True)
```

Preprocessing

- Not all of the image is relevant
- Top showing score
- Image is coloured
- Crop, convert to grayscale, and reshape as a square of size 84 x 84

```
Box(210, 160, 3)
Discrete(4)
<bound method Wrapper.close c
```




```python
def preprocess_state(self, img):
    img_temp = img[31:195]  # Choose the important area of the image
    img_temp = tf.image.rgb_to_grayscale(img_temp)
    img_temp = tf.image.resize(img_temp, [self.IM_SIZE, self.IM_SIZE],
                               method=tf.image.ResizeMethod.NEAREST_NEIGHBOR)
    img_temp = tf.cast(img_temp, tf.float32)
    return img_temp[:,:,0]
```

Capturing Experience

- How can the agent determine if projectile/ball/other is moving up or down?
- Concatenating the state space for the last 4 timesteps as one input to the CNN
- Network is presented with four frames of the environment per timestep

```
def combine_images(self, img1, img2):
    if len(img1.shape) == 3 and img1.shape[0] == self.m:
        im = np.append(img1[1:,:, :],np.expand_dims(img2,0), axis=2)
        return tf.expand_dims(im, 0)
    else:
        im = np.stack([img1]*self.m, axis = 2)
        return tf.expand_dims(im, 0)
    #return np.reshape(state, [1, 4])
```

Replay from Experience

- Create minibatches from saved experience

```
def replay(self, batch_size):
    x_batch, y_batch = [], []
    minibatch = random.sample(self.memory, min(len(self.memory), batch_size))
    for state, action, reward, next_state, done in minibatch:
        y_target = self.model_target.predict(state)
        y_target[0][action] = reward if done else reward + \
        self.gamma * np.max(self.model.predict(next_state)[0])
        x_batch.append(state[0])
        y_batch.append(y_target[0])

    self.model.fit(np.array(x_batch), np.array(y_batch), batch_size=len(x_batch), verbose=0)
    #epsilon = max(epsilon_min, epsilon_decay*epsilon) # decrease epsilon
```

DQN Atari: Coding Fragments

- The Model

```
# Init model
self.model = Sequential()
self.model.add( Conv2D(32, 8, (4,4), activation='relu',padding='valid', input_shape=(IM_SIZE, IM_SIZE, m)))
#self.model.add(MaxPooling2D())
self.model.add( Conv2D(64, 4, (2,2), activation='relu',padding='valid'))
self.model.add(MaxPooling2D())
self.model.add( Conv2D(64, 3, (1,1), activation='relu',padding='valid'))
self.model.add(MaxPooling2D())
self.model.add(Flatten())
self.model.add(Dense(256, activation='elu'))
self.model.add(Dense(action_size, activation='linear'))
self.model.compile(loss='mse', optimizer=Adam(lr=alpha, decay=alpha_decay))
self.model_target = tf.keras.models.clone_model(self.model)
```

DQN Atari: Coding Fragments

- Training

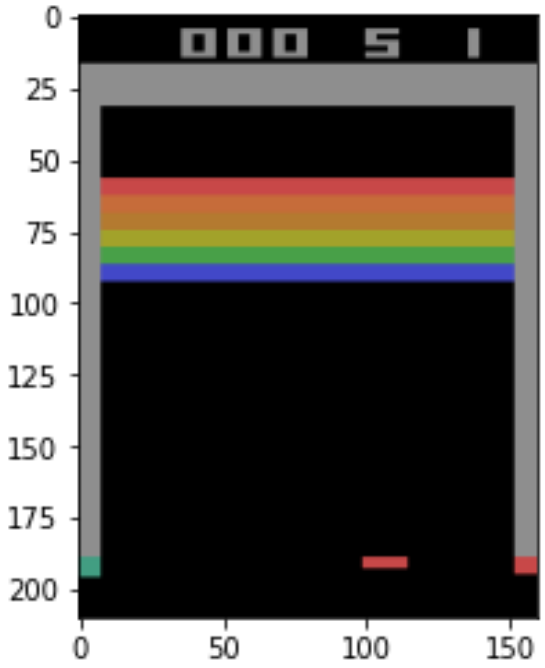
```
def train(self):
    scores = deque(maxlen=100)
    avg_scores = []
    for e in range(EPOCHS):
        state = self.env.reset()
        state = self.preprocess_state(state)
        state = self.combine_images(state, state)
        done = False
        i = 0
        while not done:
            action = self.choose_action(state,self.epsilon)
            next_state, reward, done, _ = self.env.step(action)
            next_state = self.preprocess_state(next_state)
            next_state = self.combine_images(next_state, state)
            self.remember(state, action, reward, next_state, done)
            state = next_state
            #.........................
            i += reward
        #.....................
        self.replay(self.batch_size)
```

Don't forget to set `env_string = 'BreakoutDeterministic-v5'`

Atari: Visualising the OpenAI Gym Game in Colab

```
import gym
import matplotlib.pyplot as plt

from gym import envs
all_envs = envs.registry.all()
env_ids = [env_spec.id for env_spec in all_envs]
#print(*env_ids, sep='\n')

env_name = 'BreakoutDeterministic-v4'
env = gym.make(env_name)

obs = env.reset()
plt.imshow(env.render(mode='rgb_array'))

print(env.observation_space)
print(env.action_space)

env.close
```

```
Box(210, 160, 3)
Discrete(4)
<bound method Wrapper.close o
```



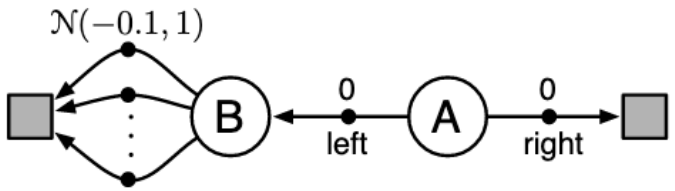
See most recent code for setup of Gym environment

Combatting Catastrophic Forgetting and Maximisation Bias and Instability

- More on Double DQNs
- Dueling DQNs
- Prioritised Replays

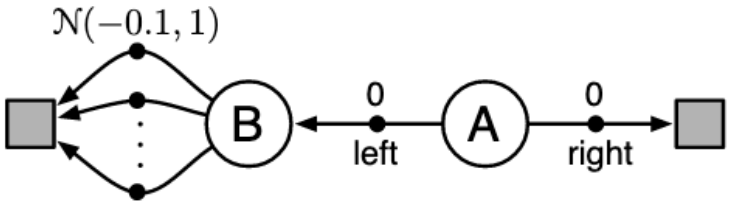
Maximization bias in Q-learning

- Consider an MDP with two non-terminal states A and B
- Episodes start in A with a choice between two actions, left and right
- The right action transitions immediately to the terminal state with a reward and return of zero
- The left action transitions to B, also with a reward of zero
- From B there are many possible actions all of which cause immediate termination with a reward drawn from a normal distribution with mean −0.1 and variance 1.0
- Thus, the expected return for any trajectory starting with left is −0.1, and thus taking left in state A is a mistake



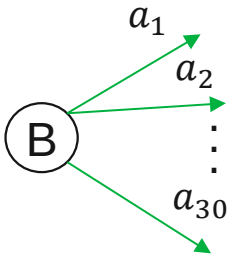
Maximization bias in Q-learning

- Assume we observed 4 episodes:
  - $\{A, \rightarrow, 0\}$
  - $\{A, \leftarrow, 0, B, a_1, -1.1\}$
  - $\{A, \leftarrow, 0, B, a_2, 0.9\}$
  - $\{A, \leftarrow, 0, B, a_3, -0.1\}$
- $\max_a [Q(B, a)] = 0.9$ yet $q^*(B, \cdot) = -0.1$
- Maximization bias is common when action outcomes are noisy
- Some actions will be evaluated following a lower-than-expected sampled sampled reward
- The max operator biases towards higher-than-expected sampled reward



Double Q-learning

- Assume visiting state B a total of 60 times (2 times per action)
- $a_i \sim \mathcal{N}(-0.1, 1)$
- $\mathbb{E}\left(\max_a [Q(B,a)]\right) = \sim^{*} 1.3 \neq -0.1$
- How can we fix this bias?
  - Store and update 2 independent Q tables: Q_1, Q_2
  - For each observed transition update one Q table (and not the other!)
  - Use one for choosing maximizing action and the other for retrieving the value
- $\mathbb{E}\left(Q_1(B, \operatorname{argmax}_a [Q_2(B,a)]\right) = -0.1$



$*$ evaluated through sim

Double Q learning is unbiased

- We are actually interested in the action that maximizes the expected reward
  - $\max_a \left[\mathbb{E}[Q(s,a)]\right]$
- In Q learning $\mathbb{E}\left(\max_a [Q(s,a)]\right) \neq \max_a \left[\mathbb{E}[Q(s,a)]\right]$
  - Due to the maximization bias
- Double Q learning $\mathbb{E}\left(Q_1(s, \arg\max_a [Q_2(s,a)]\right) = \max_a \left[\mathbb{E}[Q(s,a)]\right]$
  - Because the returned value (from Q_1) is independent of the maximizing action (from Q_2)

Double Q-learning

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

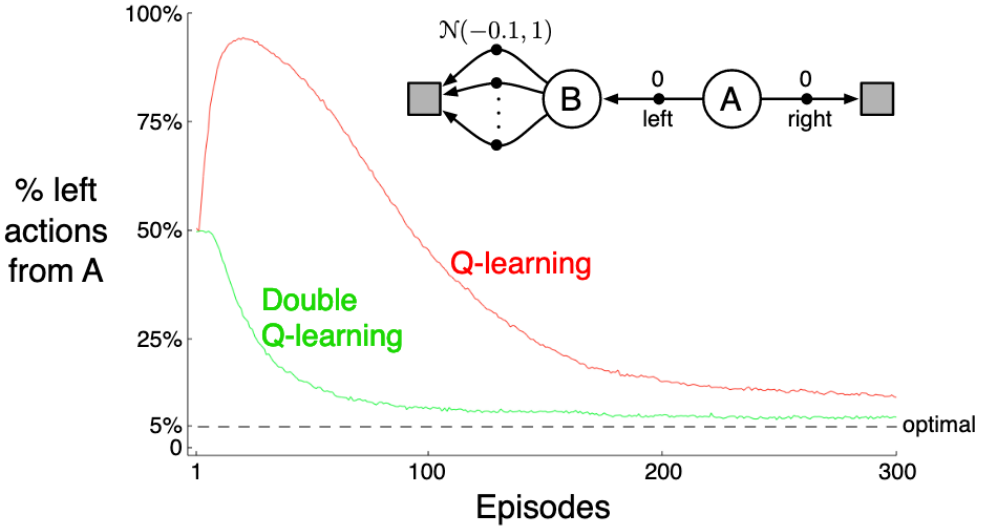
Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$
  until S is terminal

Double Q-learning

Initialize $Q_1(s,a)$ and $Q_2(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily
Initialize $Q_1(\textit{terminal-state}, \cdot) = Q_2(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q_1 and Q_2 (e.g., ε-greedy in $Q_1 + Q_2$)
    Take action A, observe R, S'
    With 0.5 probabilility:
      $Q_1(S,A) \leftarrow Q_1(S,A) + \alpha\Big(R + \gamma Q_2\big(S', \operatorname{argmax}_a Q_1(S',a)\big) - Q_1(S,A)\Big)$
    else:
      $Q_2(S,A) \leftarrow Q_2(S,A) + \alpha\Big(R + \gamma Q_1\big(S', \operatorname{argmax}_a Q_2(S',a)\big) - Q_2(S,A)\Big)$
    $S \leftarrow S'$
  until S is terminal

Double Q-learning

- From B there are many possible actions all of which cause immediate termination with a reward drawn from $\mathcal{N}(-0.1,1)$
- Taking left in state A is always a mistake



Dueling DQNs

- Target network tends

Priorotised Replays

Summary

Concludes the overview of Value Functions

WK5: basics

- Trial and Error learning
- Balancing Exploration versus Exploitation
- Action Selection - UCB

WK6: Bellman's Equation, DP and MC

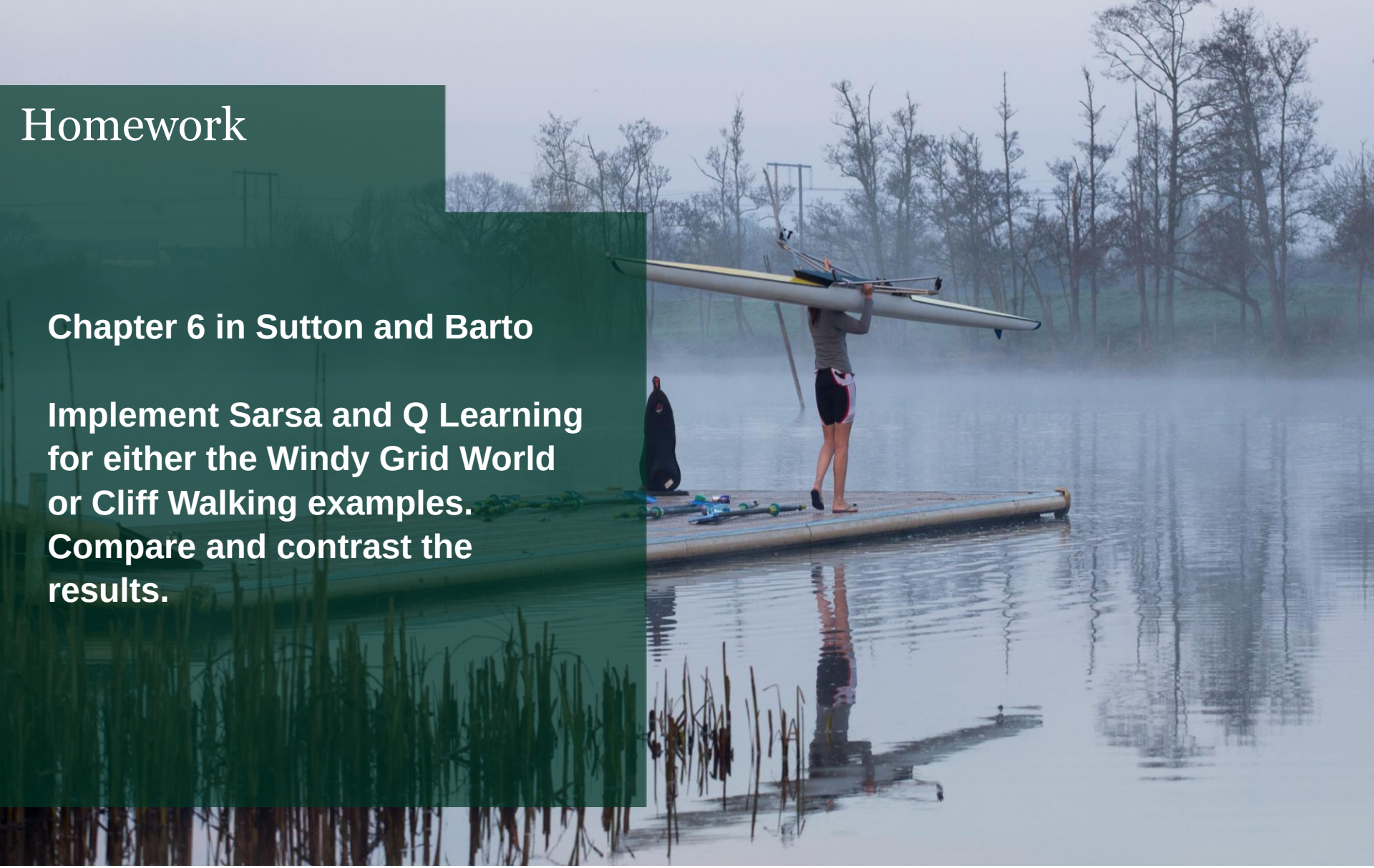
WK7: TD - Sarsa and Q

WK 8: TD Gammon and DQN CartPole

WK 9: DQN Atari

Next: Policy Gradient Approaches

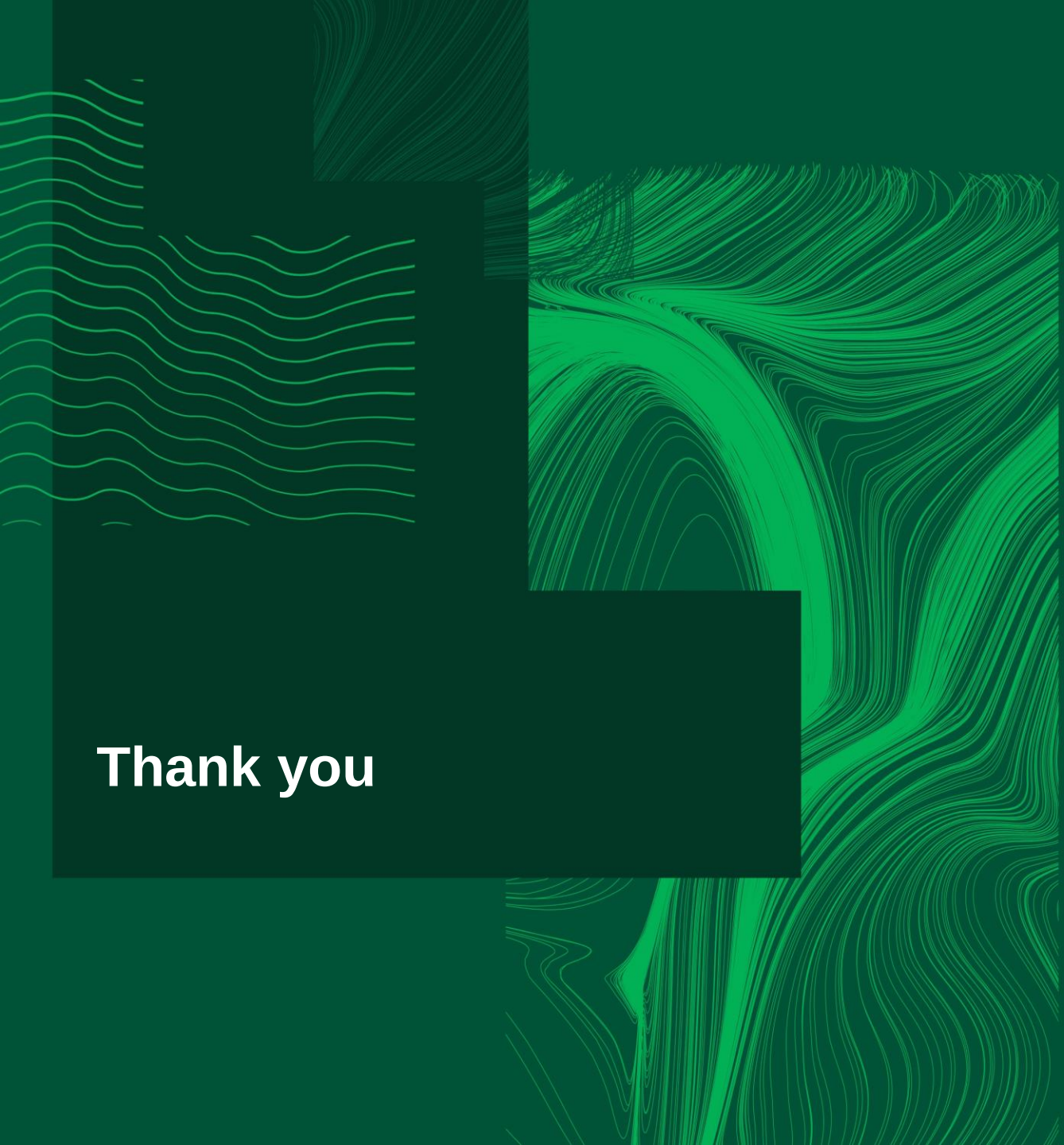
Homework



Chapter 6 in Sutton and Barto

Implement Sarsa and Q Learning for either the Windy Grid World or Cliff Walking examples. Compare and contrast the results.

Thank you



UNIVERSITY OF LIMERICK
OLLSCOIL LUIMNIGH

University of Limerick,
Limerick, V94 T9PX,
Ireland.

Ollscoil Luimnigh,
Luimneach,
V94 T9PX, Éire.

+353 (0) 61 202020

ul.ie