20 - Deep Reinforcement Learning

Ludwig Krippahl

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Summary

- Introduction to Deep Reinforcement Learning
- Exploration and Exploitation
- Learning policies with Deep Neural Networks
- Example: cartpole problem
- Assignment 2

Introduction to DRL

Introduction

Previously:

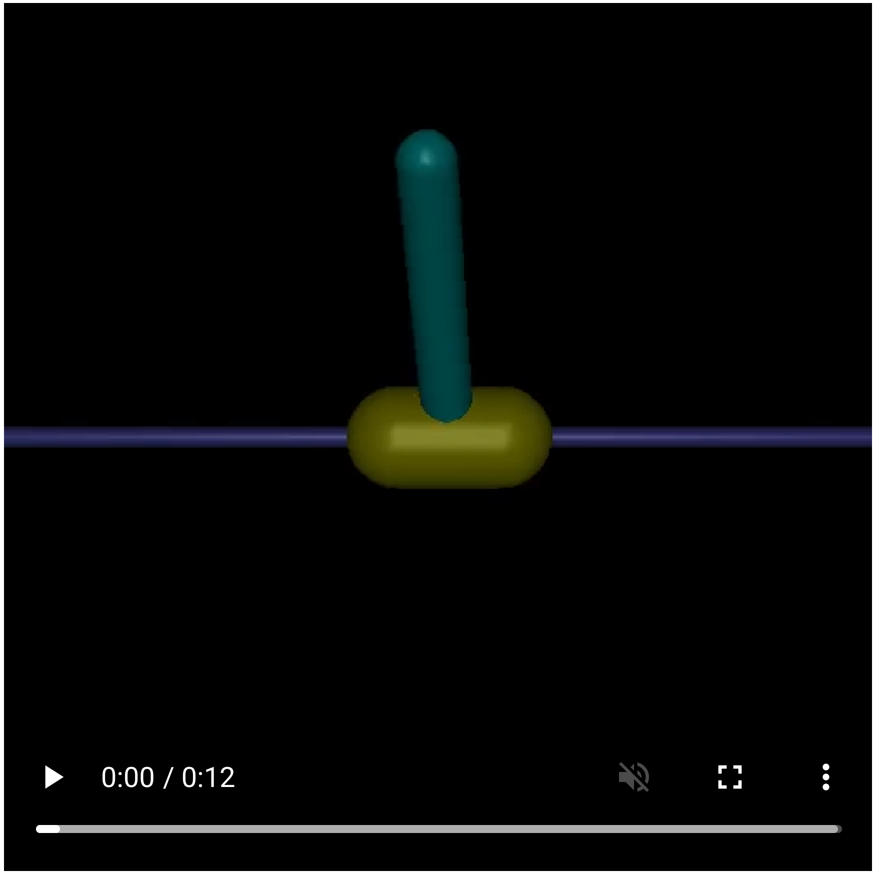
- Knowing the Markov Decision Process (MDP) we can:
  - Compute the V-function for a policy
  - Compute the Q-function
  - Improve the policy choosing best action for each state

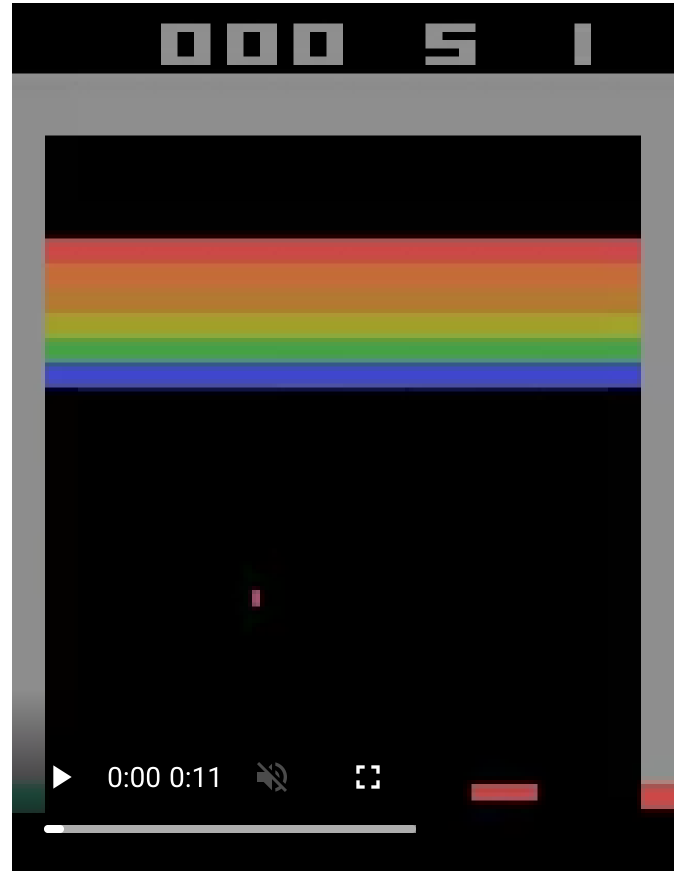
Unfortunately...

- This requires full knowledge of the MDP
- And complete information about the states
  - We must enumerate all combinations of states and actions

Introduction

- Realistic cases are too large (or continuous)





- But we can combine the fundamental ideas with deep learning

Features of deep learning

- Delayed feedback:
  - Without the full MDP we must rely on sequences of events and feedback can be delayed
- Evaluative feedback:
  - In supervised learning we know the ground truth but in reinforcement learning we only have relative values (e.g. this way better than that way)
- Sampled feedback
  - We do not know the complete decision problem and only have samples of actions, states and rewards which depend on how we explore the state and action space.
- Feature extraction
  - Usually in deep reinforcement learning the observation does provide the best features. E.g. to play Starcraft or drive a car

Exploration and Exploitation

Exploration and Exploitation

Gathering data

- Agent must interact with environment and observe
- Exploit agent's knowledge to guide exploration?
  - Can go far but always following same recipe
- Risk different actions?
  - May be a bad idea but may reveal better alternatives.
- Tradeoff between exploration and exploitation

Exploration and Exploitation

- ϵ-greedy exploration:
  - Mostly follow best estimated action but risk random action with small ϵ
- Decaying ϵ-greedy exploration:
  - Start with large ϵ and reduce gradually during training
- Optimistic initialization:
  - Initialize Q-function with high values in order to favour novel actions
- Softmax exploration:
  - Pass Q-function through softmax and use as probability of choosing each action
- Upper Confidence Bound (UCB):
  - Favours less visited combinations considering uncertainty

$$a_t = \operatorname*{argmax}_a \left(Q_t(s,a) + c\sqrt{\frac{2\ln t}{N_t(a)}} \right)$$

Learning Policies

Learning Policies

- We cannot use the same Iterative Policy Evaluation algorithm
- But we can use temporal-difference learning.

Improving the state-value function:

- Recall:

$$v_\pi(s_t) = \mathbb{E}_\pi \left(R_{t+1} + \gamma v_\pi(s_{t+1})\right)$$

- If we want to improve V-function:

$$v_{t+1}(s_t) = v_t(s_t) + \alpha_t \left(R_{t+1} + \gamma v_t(s_{t+1}) - v_t(s_t)\right)$$

Improving the action-value (Q-funcion)

- State-action-reward-state-action algorithm (SARSA)

$$q_{t+1}(s_t, a_t) = q_t(s_t, a_t) + \alpha_t \left(R_{t+1} + \gamma q_t(s_{t+1}, a_{t+1}) - q_t(s_t, a_t)\right)$$

- Estimate for the future action values to be given by the action the policy will choose

- Q-learning algorithm:

$$q_{t+1}(s_t, a_t) = q_t(s_t, a_t) + \alpha_t \left(R_{t+1} + \gamma \underset{a}{\operatorname{argmax}} \left(q_t(s_{t+1}, a)\right) - q_t(s_t, a_t)\right)$$

- Uses the best possible action at next step to estimate discounted future return

- On-policy and off-policy

- SARSA is an on-policy algorithm, because it follows the policy
- Q-learning is off-policy since it ignores the policy for the future return

Deep Reinforcement Learning

- These learning algorithms assume a V table and a Q table to update
- There is no such thing in deep reinforcement learning
- We must approximate the Q-function with a deep neural network:
  - The input is the observation of the state
  - The output, one for each action, is $q(s, a)$
  - Linear activation on the output (regression problem)
  - Use MSE or equivalent
- What data do we use to train the network?

Learning Policies

Training the network to approximate Q

- To train the network we need (X, Y) pairs:
  - X: input, corresponding to the observation of the state
  - Y: output, corresponding to the target Q-values for the actions
- We start from experiences:
  - Tuples state, action, next state, reward obtained and flag for terminal state:

$$(s_t, a_t, s_{t+1}, r_{t_1}, terminal)$$

- We use the network to estimate future returns.
- Adapting SARSA:

$$y_i^{SARSA}(a_t) = R_{t+1} + \gamma q_t(s_{t+1}, a_{t+1}; \theta_i)$$

- Adapting Q-learning:

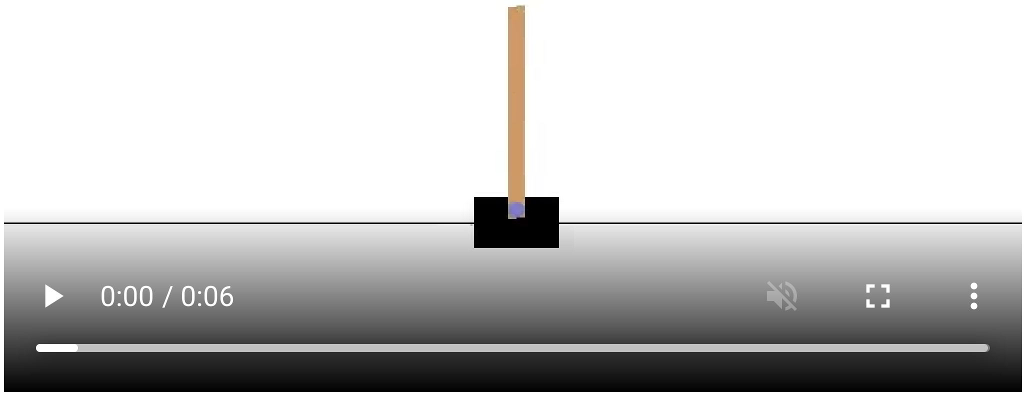
$$y_i^{Q-learn}(a_t) = R_{t+1} + \gamma \underset{a}{\text{argmax}} \left(q_t(s_{t+1}, a; \theta_i)\right)$$

Demo: cartpole problem

Cartpole

Based on a tutorial by Mike Wang, towardsdatascience.com

- Open-AI CartPole-V1
  - Applying a force of +1 or -1 to a cart where pole is balanced.
  - The pole must be kept within 15º of vertical



Cartpole

- Importing and setup. Note: you need the Open-AI gym

```
import gym
import tensorflow as tf
import numpy as np
from tensorflow import keras
from collections import deque
import random

RANDOM_SEED = 5
tf.random.set_seed(RANDOM_SEED)

env = gym.make('CartPole-v1')
env.seed(RANDOM_SEED)
np.random.seed(RANDOM_SEED)

train_episodes = 300
```

Cartpole

- Model for learning Q-function

```
def agent(state_shape, action_shape):
    learning_rate = 0.001
    init = tf.keras.initializers.HeUniform()
    model = keras.Sequential()
    model.add(keras.layers.Dense(24, input_shape=state_shape,
                                 activation='relu',
                                 kernel_initializer=init))
    model.add(keras.layers.Dense(12, activation='relu',
                                 kernel_initializer=init))
    model.add(keras.layers.Dense(action_shape,
                                 activation='linear',
                                 kernel_initializer=init))
    model.compile(loss=tf.keras.losses.Huber(),
                  optimizer=tf.keras.optimizers.Adam(lr=learning_rate),
                  metrics=['accuracy'])
    return model
```

- Two optimizations

  - He uniform initializer, uniform $6/\sqrt{fan_{in}}$
  - Huber loss function, similar to MSE but becomes linear for larger error

Cartpole

- Training function uses two models to help stabilize training
  - `target_model` lags behind `model`

```
def train(env, replay_memory, model, target_model, done):
    discount_factor = 0.618
    batch_size = 64 * 2
    mini_batch = random.sample(replay_memory, batch_size)
    current_states = np.array([transition[0] for transition in mini_batch])
    current_qs_list = model.predict(current_states)
    new_current_states = np.array([transition[3] for transition in mini_batch])
    future_qs_list = target_model.predict(new_current_states)
    X = []
    Y = []
    for index, (observation, action, reward, new_observation, done) in enumerat
        if not done:
            max_future_q = reward + discount_factor * np.max(future_qs_list[inc
        else:
            max_future_q = reward
        current_qs = current_qs_list[index]
        current_qs[action] = max_future_q
        X.append(observation)
        Y.append(current_qs)
    model.fit(np.array(X), np.array(Y), batch_size=batch_size, verbose=0, shuff
```

FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE NOVA DE LISBOA

Cartpole

Main function, setup

- ϵ decais over time.
- `MIN_REPLAY_SIZE` is smallest size for pool of experiences
- `target_model` is a copy of `model`

```
def main():
    epsilon = 1
    max_epsilon = 1
    min_epsilon = 0.01
    decay = 0.01
    MIN_REPLAY_SIZE = 1000

    model = agent(env.observation_space.shape, env.action_space.n)
    target_model = agent(env.observation_space.shape, env.action_space.n)
    target_model.set_weights(model.get_weights())

    replay_memory = deque(maxlen=50000)
    steps_to_update_target_model = 0
```

Cartpole

- Adding experiences to the `replay_memory`
  - It is a `deque`, so it discards oldest elements if capacity is reached
- Uses ϵ to choose exploration or exploitation

```
for episode in range(train_episodes):
    total_training_rewards = 0
    observation = env.reset()
    done = False
    while not done:
        steps_to_update_target_model += 1
        if True:
            env.render()
        random_number = np.random.rand()
        if random_number <= epsilon:
            action = env.action_space.sample()
        else:
            reshaped = observation.reshape([1, observation.shape[0]])
            predicted = model.predict(reshaped).flatten()
            action = np.argmax(predicted)
        new_observation, reward, done, info = env.step(action)
        replay_memory.append([observation, action, reward, new_observation, dor
```

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Cartpole

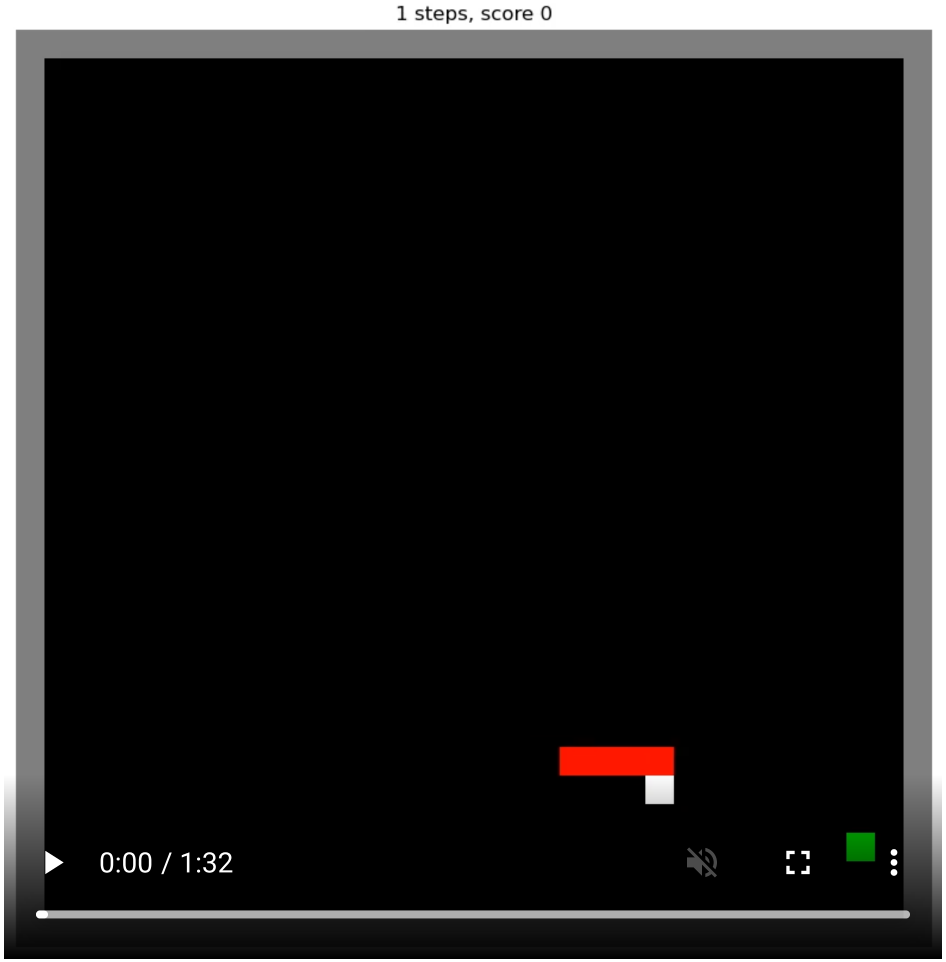
- Train model if there are enough experiences in memory pool
- Update `target_model` by copying weights, but less frequently

```
        if len(replay_memory) >= MIN_REPLAY_SIZE and \
            (steps_to_update_target_model % 4 == 0 or done):
            train(env, replay_memory, model, target_model, done)
        observation = new_observation
        total_training_rewards += reward
        if done:
            print('Rewards: {} after n steps = {} with final reward = {}'.form
                      total_training_rewards, episode, reward))
            total_training_rewards += 1

            if steps_to_update_target_model >= 100:
                print('Copying main network weights to the target network weig
                target_model.set_weights(model.get_weights())
                steps_to_update_target_model = 0
            break
    epsilon = min_epsilon + (max_epsilon - min_epsilon) * np.exp(-decay * epis
env.close()
```

Assignment 2

Assignment 2

- Motivation: train a snake player



Assignment 2

- Motivation: train a snake player

Goals:

- Explore different options
  - Networks, algorithms (SARSA, Q-learning, etc.)
  - Scheduling, experiences, exploration
  - ...
- Understand the problems
  - Reward is rare, only when agent finds food
- Basically, think and learn
- Grading will depend greatly on explanations

Assignment 2

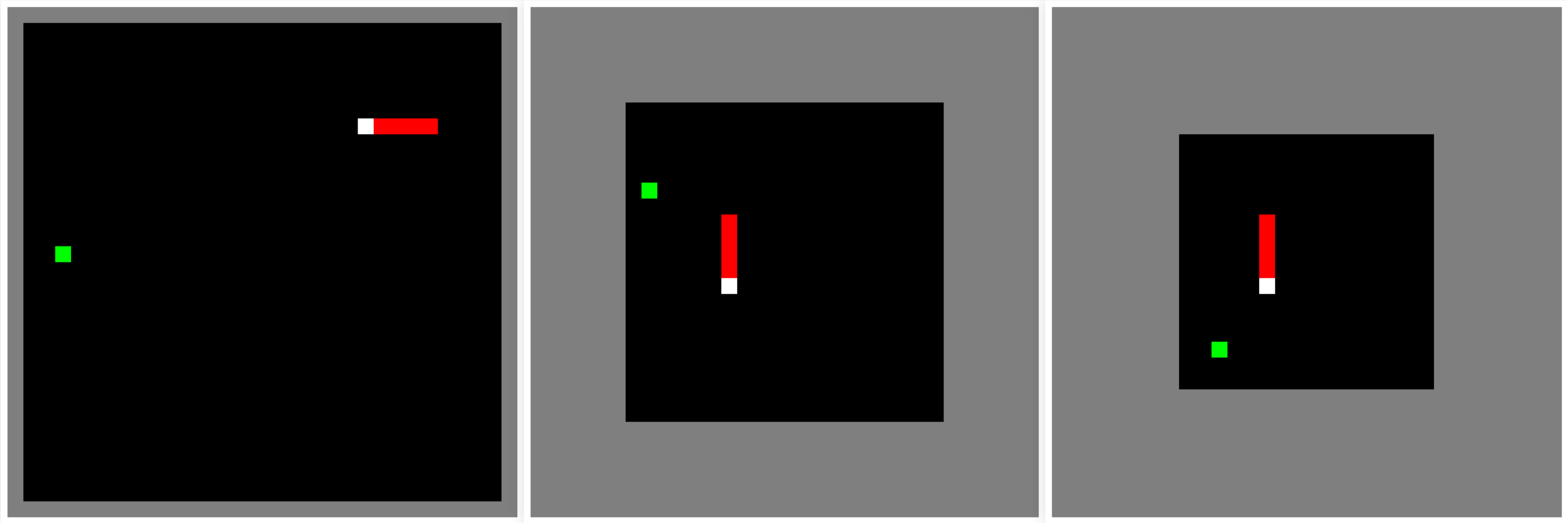
- The snake game

```
class SnakeGame:
    " Implements the snake game core"

    def __init__(self, width, height, food_amount=1,
                 border = 0, grass_growth = 0,
                 max_grass = 0):
        ...

    def step(self, action):
        ...
        return self.board_state(),reward,self.done, {'score':self.score}

    def reset(self):
        ...
        return self.board_state(),0,self.done, {'score':self.score}
```

- The board state is a numpy array (height,width,3)
  - Snake: tail in red, head in white
  - Food: green
  - Walls (border): gray

Assignment 2

- You can control border thickness:
  - `SnakeGame(w,h, border = n)`
- Total width and height will be `w+2*n,  h+2*n`



- Don't go too small, minimum 14 (?)
- Target: 30x30, border 1, for images of 32x32

Assignment 2

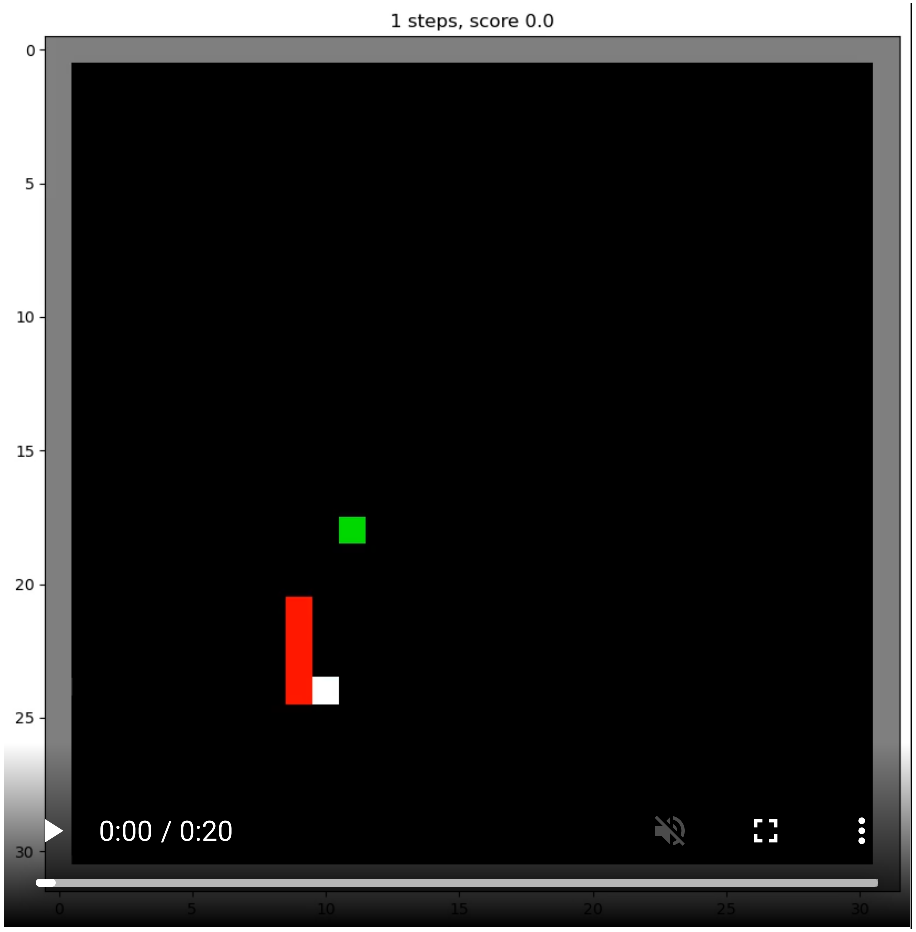
- Problem of rare rewards

- `SnakeGame(30,30,border=1,max_grass=0.05,grass_growth=0.001)`

Assignment 2

- Problem of rare rewards

- Use heuristic to populate initial pool of examples



Assignment 2

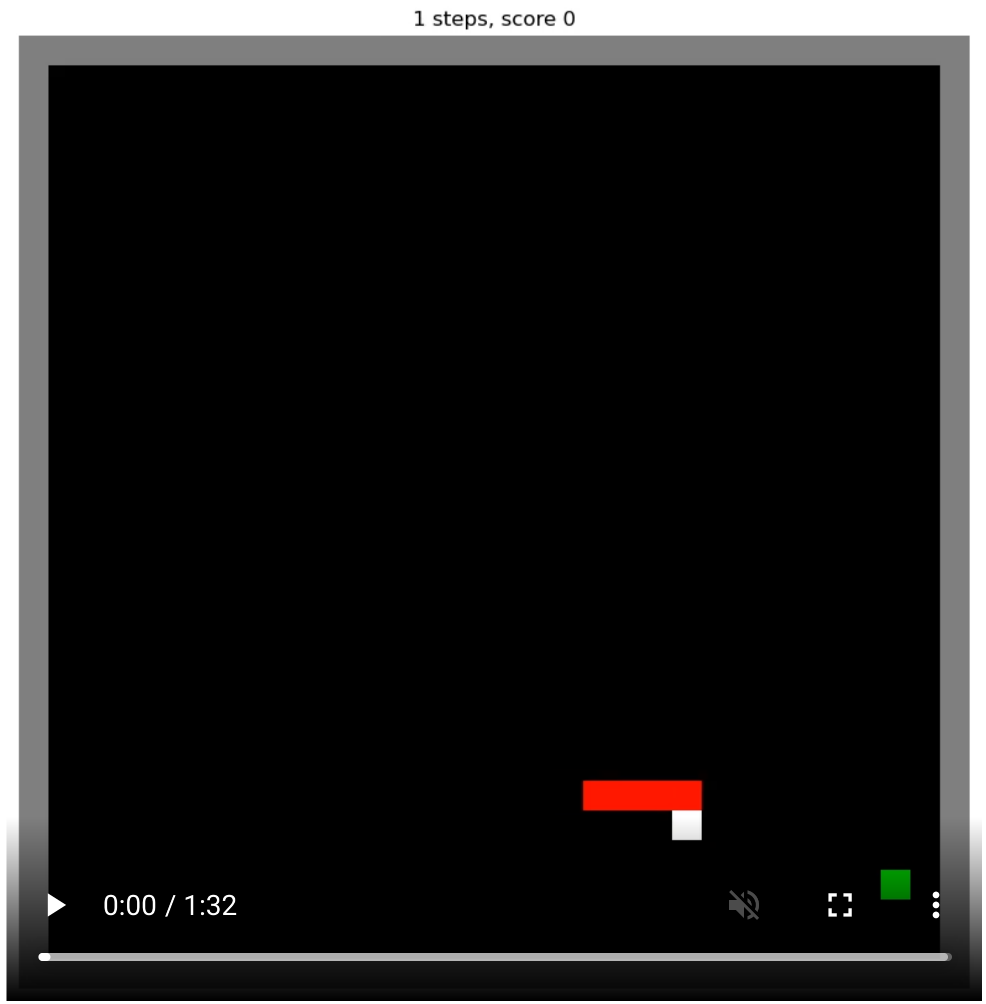
- Problem of rare rewards
  - Use heuristic to populate initial pool of examples
- You can cheat with this method:

```
class SnakeGame:
    ...
    def get_state(self):
        "easily get current state (score, apple, snake head and tail)"
        score = self.score
        apple = self.apples
        head = self.snake[0]
        tail = self.snake[1:]
        return score,apple,head,tail,self.direction
```

- This is **NOT** for training
  - The agent must use the image of the board
- But you can use it to generate examples

Assignment 2

- Use heuristic to populate initial pool of examples, result



Assignment 2

- Can take a while to train
  - Especially because of playing the game and predicting actions
- Experiment with smaller boards

Instructions

- The code is available now (you just need `SnakeGame`)
- I will post the instructions and questions files this week
- Deadline is June 10 (plus 48 hours)

Summary

Summary

- Deep Reinforcement Learning
- Exploration and Exploitatio
- Improving policy
  - Approximate Q-function with DNN
  - SARSA and Q-learning adapted to DNN
- Polecart example
- Assignment 2

Further reading (Optional)

- Morales, Grokking Deep Reinforcement Learning, 2020, Chp. 4-6, 8