A (very gentle) introduction to deep reinforcement learning for solving complex CO problems

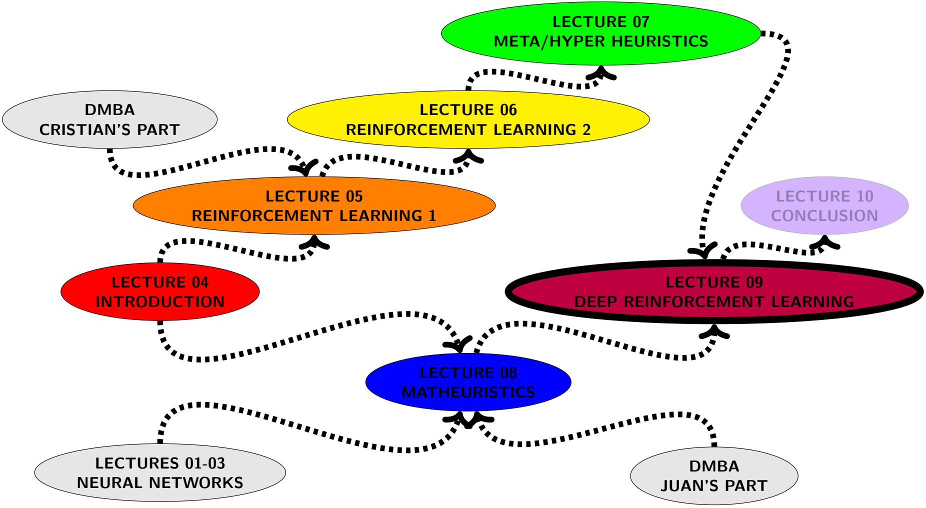
Maxence Delorme

Department of Econometrics and Operations Research
Tilburg University

m.delorme@tilburguniversity.edu

OR & ML Lecture 9

Our journey



LECTURE 07
META/HYPER HEURISTICS
DMBA
CRISTIAN'S PART
LECTURE 06
REINFORCEMENT LEARNING 2
LECTURE 05
REINFORCEMENT LEARNING 1
LECTURE 10
CONCLUSION
LECTURE 04
INTRODUCTION
LECTURE 09
DEEP REINFORCEMENT LEARNING
LECTURE 08
MATHEURISTICS
LECTURES 01-03
NEURAL NETWORKS
DMBA
JUAN'S PART

Outline

Plan

A hyper-heuristic for the BPP

Solve the following instance with the 2 heuristics

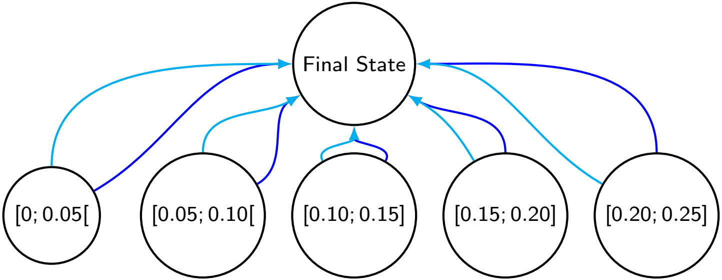
- $c = 100, w = [60, 50, 18, 31, 19, 22]$

One instance of our data set

- $m = 250$ items and capacity $c = 100$
- An instance is composed of m_1 large items and m_2 small items
- m_1 is uniformly distributed in the range $[0; \frac{m}{4}]$ and $m_2 = m - m_1$
- The item weights have the following distributions
  - Large items are uniformly distributed in the range $]0.5c; 0.9c]$
  - Small items are uniformly distributed in the range $[0.1c; 0.5c]$
- In addition, a large item has 85% chance to be odd while a small item has 85% chance to be even

HH choosing a BPP heuristic with Q-learning

One possible model for the problem (teal for FFD and blue for SS)



Learnt policy: use the Subset-Sum heuristic if there are less than 10% **of large items and First-Fit Decreasing otherwise**

But is it really the optimal policy?

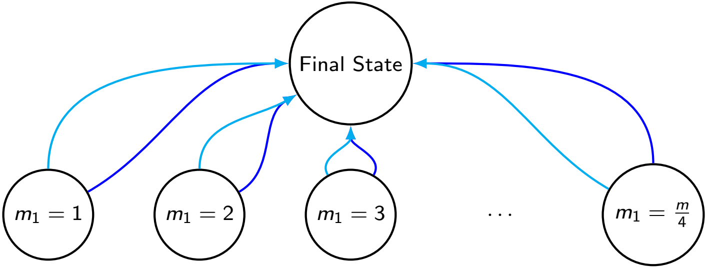
Activity

Discuss 5 minutes with your neighbour about

- Whether or not the 10% threshold is really the best to switch to First-Fit Decreasing
- How could we change the state definition to have a more accurate threshold if the number of items m is fixed ?
- How could we change the state definition to have a more accurate threshold if the number of items m is not fixed anymore?
- What issues could arise with such a state definition?

State definition for a fixed m

One possible model for the problem (teal for FFD and blue for SS)



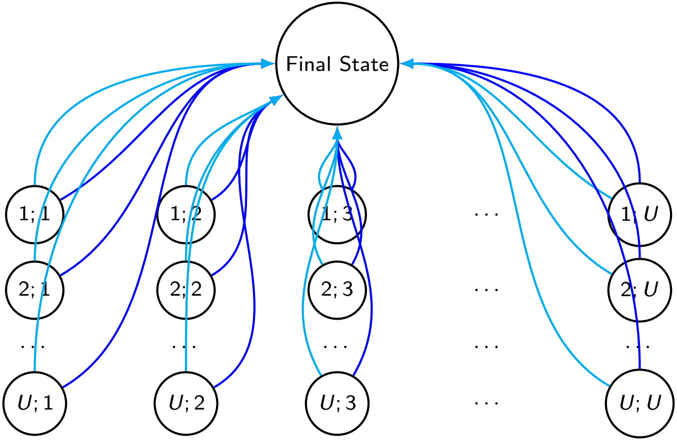
Results (decreasing α, $\gamma = 1$, $\epsilon = 0.1$, $q_0 = 0$)

30	-0.24	0.0 (0)	0.2 (17)
31	-1.07	-1.0 (1)	0.1 (14)
32	0.4	0.3 (3)	-0.1 (14)
33	0.09	0.1 (11)	0.0 (5)
34	-0.5	-1.0 (1)	-0.5 (14)
35	1.5	0.5 (14)	-1.0 (1)

Use the Subset-Sum heuristic if $m_1 \leq 25$, use First-Fit decreasing if $m_1 \geq 35$, the policy is unclear in-between . . . what can we do?

State definition for a non-fixed m

One possible model for the problem (teal for FFD and blue for SS)

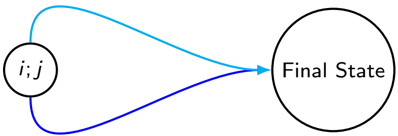


Being in state $(i; j)$ means that the instance has i large items and j small items and U is an upper bound on the number of items

What are the drawbacks of such a state definition?

State definition for a non-fixed m

An ideal model for the problem (teal for FFD and blue for SS)



Being in state $(i; j)$ means that the instance has i large items and j small items

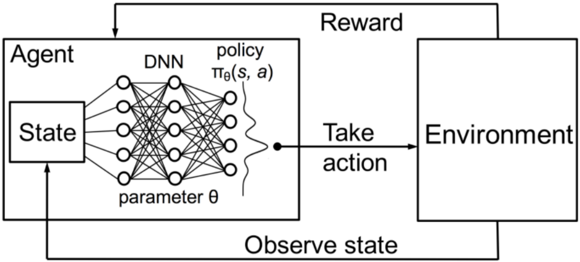
Such a state definition seems much simpler . . . but it is not suitable for q-learning as we need one cell of the q-table for each state.

What if we could replace the q-table by a function whose input is a state and whose output is a q-value?

Plan

Introduction to DRL

Schematic representation of DRL (from Mao et al.)



More on DRL, see Mnih et al.

- Deep Q-learning (DQL) is a **value-based** method, which means that it uses a function approximator (a deep Q-network) to estimate action function $q(s, a, \theta) \approx q^*(s, a)$
- One can derive the optimal policy from the (well-approximated) action values
- There are also **policy-based** methods which directly try to approximate the optimal policy

DQL algorithm

From Q-learning to DQL

Initialise $q(s, a)\ \forall s \in S, \forall a \in A(s)$

for each episode $e \in E$ **do**

  Initialise $t = 0$ and s_t

  while s_t is not a terminal state **do**

    Select action $a_t \in A(s_t)$ based on ϵ-greedy learning policy

    Observe reward r_{t+1} and the new state s_{t+1}

    Update the action value $q(s_t, a_t)$ with

$$q(s_t, a_t) = (1 - \alpha) \cdot q(s_t, a_t) + \alpha \cdot \left(r_{t+1} + \gamma \cdot \max_a \left\{q(s_{t+1}, a)\right\}\right)$$

    $t = t + 1$

  end

end

DQL algorithm

From Q-learning to DQL

Initialise θ

for each episode $e \in E$ **do**

  Initialise $t = 0$ and s_t

  while s_t is not a terminal state **do**

    Select action $a_t \in A(s_t)$ based on ϵ-greedy learning policy

    Observe reward r_{t+1} and the new state s_{t+1}

    Update the action value $q(s_t, a_t)$ with

$$q(s_t, a_t) = (1 - \alpha) \cdot q(s_t, a_t) + \alpha \cdot \left(r_{t+1} + \gamma \cdot \max_a \left\{q(s_{t+1}, a)\right\}\right)$$

    $t = t + 1$

  end

end

DQL algorithm

From Q-learning to DQL

Initialise θ

for each episode $e \in E$ **do**

Initialise $t = 0$ and s_t

while s_t is not a terminal state **do**

Select a random action $a_t \in A(s_t)$ with probability ϵ and $\max_a q(s_t, a, \theta)$ otherwise

Observe reward r_{t+1} and the new state s_{t+1}

Update the action value $q(s_t, a_t)$ with

$$q(s_t, a_t) = (1 - \alpha) \cdot q(s_t, a_t) + \alpha \cdot \left(r_{t+1} + \gamma \cdot \max_a \left\{q(s_{t+1}, a)\right\}\right)$$

$t = t + 1$

end

end

DQL algorithm

From Q-learning to DQL

Initialise θ

for each episode $e \in E$ **do**

  Initialise $t = 0$ and s_t

  while s_t is not a terminal state **do**

    Select a random action $a_t \in A(s_t)$ with probability ϵ and $\max_a q(s_t, a, \theta)$ otherwise

    Observe reward r_{t+1} and the new state s_{t+1}

    Update θ by performing an **online gradient step** to minimize the squared loss

$$\left(y_t - q(s_t, a_t, \theta)\right)^2$$

$$\text{where} \quad y_t = r_{t+1} + \gamma \max_a q(s_{t+1}, a, \theta)$$

    $t = t + 1$

  end

end

Target network in DQL

DQL algorithm

Initialise θ

for each episode $e \in E$ **do**

  Initialise $t = 0$ and s_t

  while s_t is not a terminal state **do**

    Select a random action $a_t \in A(s_t)$ with probability ϵ and $\max_a q(s_t, a, \theta)$ otherwise

    Observe reward r_{t+1} and the new state s_{t+1}

    Update θ by performing an **online gradient step** to minimize the squared loss

$$\left(r_{t+1} + \gamma \max_a q(s_{t+1}, a, \theta) - q(s_t, a_t, \theta)\right)^2$$

    $t = t + 1$

  end

end

Target network in DQL

DQL algorithm

Initialise θ, $i = 0$, and θ'

for each episode $e \in E$ **do**

Initialise $t = 0$ and s_t

while s_t is not a terminal state **do**

Select a random action $a_t \in A(s_t)$ with probability ϵ and $\max_a q(s_t, a, \theta)$ otherwise

Observe reward r_{t+1} and the new state s_{t+1}

Update θ by performing an **online gradient step** to minimize the squared loss

$$\left(r_{t+1} + \gamma \max_a q(s_{t+1}, a, \theta') - q(s_t, a_t, \theta)\right)^2$$

$t = t + 1$, $i = i + 1$

If $i \equiv K = 0$ **then** $\theta' = \theta$

end

end

Experience replay in DQL

DQL algorithm

Initialise θ,$i = 0$, and θ'

for each episode $e \in E$ **do**

  Initialise $t = 0$ and s_t

  while s_t is not a terminal state **do**

    Select a random action $a_t \in A(s_t)$ with probability ϵ and $\max_a q(s_t, a, \theta)$ otherwise

    Observe reward r_{t+1} and the new state s_{t+1}

    Update θ by performing an **online gradient step** to minimize the squared loss

$$\left(r_{t+1} + \gamma \max_a q(s_{t+1}, a, \theta') - q(s_t, a_t, \theta)\right)^2$$

    $t = t + 1$, $i = i + 1$

    If $i \equiv K = 0$ **then** $\theta' = \theta$

  end

end

Experience replay in DQL

DQL algorithm

Initialise θ,$i = 0$, θ', and $\mathcal{M} = \emptyset$

for each episode $e \in E$ **do**

  Initialise $t = 0$ and s_t

  while s_t is not a terminal state **do**

    Select a random action $a_t \in A(s_t)$ following ϵ-greedy policy

    Observe r_{t+1} and s_{t+1} and update $\mathcal{M} = \mathcal{M} \cup (s_t, a_t, r_{t+1}, s_{t+1})$

    Update θ by performing a **minibatch stochastic gradient descent step** using a set of moves $\mathcal{B}$ **randomly** selected from $\mathcal{M}$ to minimize the squared loss

$$\sum_{(s,a,r,s') \in \mathcal{B}} \left(r + \gamma \max_{a'} q(s', a', \theta') - q(s, a, \theta) \right)^2$$

    $t = t + 1$, $i = i + 1$

    If $i \equiv K = 0$ **then** $\theta' = \theta$

  end

end

DQL algorithm

DQL pseudo-code (see Dai et al.)

Algorithm 1 **Q-learning for the Greedy Algorithm**

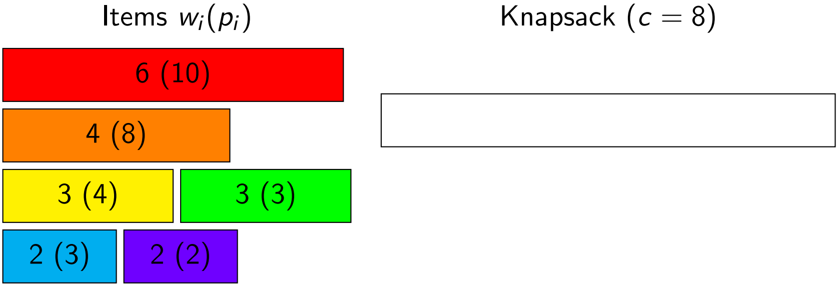
Initialize experience replay memory $\mathcal{M}$ to capacity N
for episode $e = 1$ **to** L **do**
  Draw graph G from distribution $\mathbb{D}$
  Initialize the state to empty $S_1 = ()$
  for step $t = 1$ **to** T **do**
    $v_t = \begin{cases} \text{random node } v \in \overline{S}_t, & \text{w.p. } \epsilon \\ \text{argmax}_{v \in \overline{S}_t} \widehat{Q}(h(S_t), v; \Theta), & \text{otherwise} \end{cases}$
    Add v_t to partial solution: $S_{t+1} := (S_t, v_t)$
    if $t \geq n$ **then**
      Add tuple $(S_{t-n}, v_{t-n}, R_{t-n,t}, S_t)$ to $\mathcal{M}$
      Sample random batch from $B \overset{iid.}{\sim} \mathcal{M}$
      Update Θ by SGD over $\boxed{6}$ for B
    end if
  end for
end for
return Θ

$$(y - \widehat{Q}(h(S_t), v_t; \Theta))^2, \quad \text{where } y = \gamma \max_{v'} \widehat{Q}(h(S_{t+1}), v'; \Theta) + r(S_t, v_t) \qquad (6)$$

A DQL for the 0-1 Knapsack Problem

The 0-1 knapsack problem

Given a knapsack with capacity c and a set of n items with profit p_i and weight $w_i (i = 1, \ldots, n)$, the 0-1 Knapsack Problem (0-1 KP) consists in finding a set of items with maximum profit that fits into the knapsack. Without loss of generality, we consider that $p_i > 0$ and that $0 < w_i \leq c$ and integer.



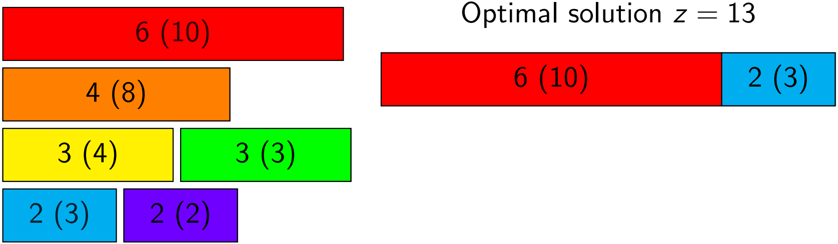
A DQL for the 0-1 Knapsack Problem

The 0-1 knapsack problem

Given a knapsack with capacity c and a set of n items with profit p_i and weight $w_i(i = 1, \dots, n)$, the 0-1 Knapsack Problem (0-1 KP) consists in finding a set of items with maximum profit that fits into the knapsack. Without loss of generality, we consider that $p_i > 0$ and that $0 < w_i \leq c$ and integer.

Items $w_i(p_i)$

Knapsack ($c = 8$)
Optimal solution $z = 13$

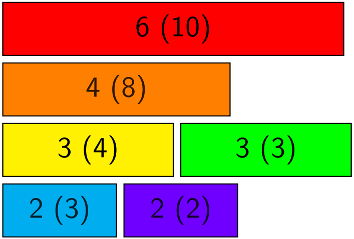


A DQL for the 0-1 Knapsack Problem

The 0-1 knapsack problem

Given a knapsack with capacity c and a set of n items with profit p_i and weight $w_i (i = 1, \dots, n)$, the 0-1 Knapsack Problem (0-1 KP) consists in finding a set of items with maximum profit that fits into the knapsack. Without loss of generality, we consider that $p_i > 0$ and that $0 < w_i \leq c$ and integer.

Items $w_i(p_i)$



Knapsack $(c = 8)$

Optimal solution $z = 13$

Activity

Discuss 5 minutes with your neighbour about

- Easy deterministic strategies to find good quality solutions for the 0-1 KP
- Solving the 0-1 KP with DQL
  - what would be the initial, intermediary, and end states?
  - what would be the actions and their rewards?

Get your inspiration for the states from the paper of Afshar et al.

A DQL for the 0-1 Knapsack Problem

One possible model for the problem

- Use a state with $2N + 4$ integers where N is an upper bound on the maximum number of items
- The first number n is the number of items left
- The second number is the capacity left in the knapsack c
- The third number S_w is the sum of the weights of the items $\sum_{i=1}^{n} w_i$
- The fourth number S_p is the sum of the profits of the items $\sum_{i=1}^{n} p_i$
- The next $2n$ numbers are the weights w_i and p_i for every item $i = 1, \ldots, n$
- The last $2N - 2n$ numbers are zeroes

A DQL for the 0-1 Knapsack Problem

One possible model for the problem

- There are N possible actions in every state which are taking one out of the N items
- Taking an item that fits into the knapsack brings a reward of p_i and updates the state accordingly (reduces n by 1, c by w_i, S_w by w_i, S_p by p_i, set w_i and p_i to 0 for the selected item i, and rearrange the vector)
- Taking an item that does not fit into the knapsack (i.e., selecting an item i such that $w_i > c$) brings a reward of 0 and leads to a final state
- Taking an item that does not exist (i.e., selecting item i such that $i > n$) brings a reward of 0 and leads to a final state

A DQL for the 0-1 Knapsack Problem

Parameters chosen for the Neural Network

- Adam optimizer ($\alpha = 0.001$)
- 3 dense hidden layers with 32, 16, 8 ReLU neurons each
- Loss function is mean squared error (MSE)
- Training happens after every action with $|\mathcal{B}| = 64$

Parameters chosen for Q-learning

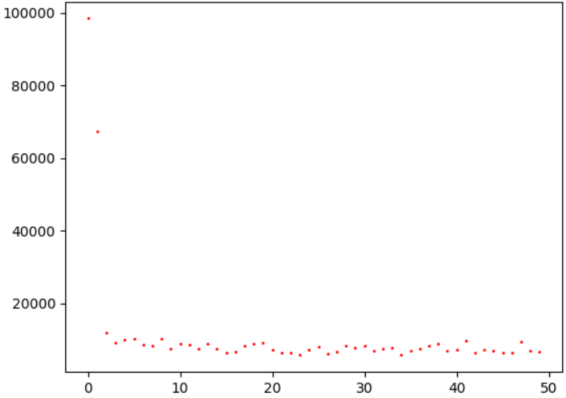
- Decreasing $\epsilon_e = \max\{0.993^e, 0.001\}$
- $\gamma = 0.999$

Parameters chosen for the 0-1 KP

- Maximum number of items $N = 50$, n randomly distributed in $[\frac{N}{2}; N]$
- Maximum number of episodes $E = 5000$
- w_i integer and uniformly distributed in the range $[1; 100]$
- $c = r \cdot \sum_{i=1}^{n} w_i$ where r is randomly distributed in $[0.1; 0.9]$
- $p_i = w_i + 10$ (hard instances)

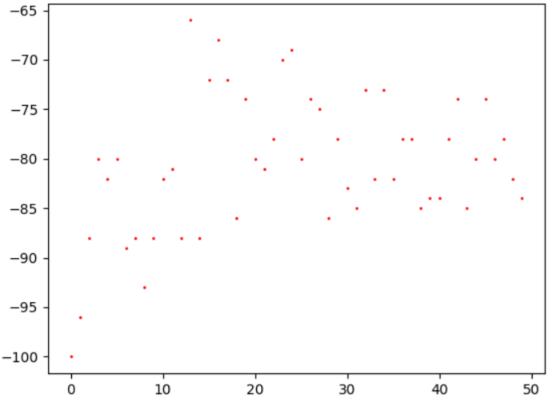
A DQL for the 0-1 Knapsack Problem

Evolution of the profit difference between the greedy solution and the solution obtained by the trained agent (groups of 100 instances)



The agent learnt how to obtain reasonably good solutions

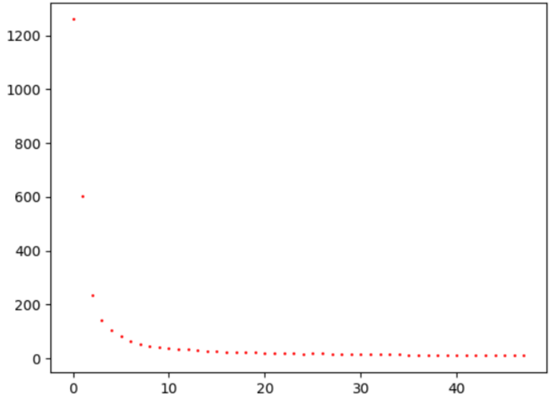
Evolution of the competitive score obtained by the trained agent: +1,0, or -1 if its solution is better, the same, or worse than the greedy solution, respectively (groups of 100 instances)



Once trained, the agent can even be occasionally better than the greedy heuristic

A DQL for the 0-1 Knapsack Problem

Evolution of the loss function of the neural network used to approximate the q-values in the DQN (groups of 100 instances)



After training, the neural network learnt how to approximate the q-values

To sum-up

Important aspects of DQL

- It is at the intersection of neural networks and reinforcement learning
- It approximates q-values with a neural network called **deep Q-network**
- The weights of the neural network θ are updated in a minibatch SGD fashion to minimize the squared loss

$$\left(r + \gamma \max_{a'} q(s', a', \theta) - q(s, a, \theta)\right)^2$$

- **Experience replay** and **target network** are important
- It is a **value-based** method (it aims at approximating the q-values); there are also **policy-based** methods who focus on the policy

Important aspects of DQL for solving CO problems

- In most cases, the DQL builds a solution step-by-step (greedy heuristic)
- Attention should be paid to how the states are defined: if one aims at solving every 0-1 KP, one should use a general way to define the states
- There is (in general) a limit on the size of the memory for experience replay

Are there tools to help us with DQL environments?

Plan

Important features of the Gym OpenAI library

- The gym library is a collection of test problems (environments)
- Those test problems can be used to train and test (deep) reinforcement learning algorithms
- These environments have a shared interface which means that a DRL algorithm can be used on several test problems by changing only one line of code (the one defining the environment)
- Available at `https://www.gymlibrary.dev/`
- Contains many "Atari" and "classic control" games
- Allows you to create your own environment

Watch the training of "MountainCar-v0 ", "CartPole-v1", and "LunarLander-v2"

Gym OpenAI library

CartPole-v1 description

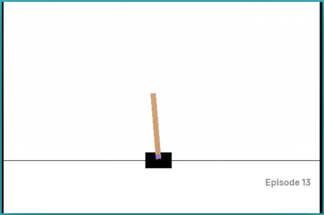
CartPole-v1

A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every timestep that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center.

This environment corresponds to the version of the cart-pole problem described by Barto, Sutton, and Anderson [Barto83].

[Barto83] AG Barto, RS Sutton and CW Anderson, "Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problem", IEEE Transactions on Systems, Man, and Cybernetics, 1983.

VIEW SOURCE ON GITHUB



RandomAgent on CartPole-v1

CartPole-v1 environment

```
Observation:
    Type: Box(4)
    Num     Observation               Min                     Max
    0       Cart Position             -4.8                    4.8
    1       Cart Velocity             -Inf                    Inf
    2       Pole Angle                -0.418 rad (-24 deg)    0.418 rad (24 deg)
    3       Pole Angular Velocity     -Inf                    Inf

Actions:
    Type: Discrete(2)
    Num   Action
    0     Push cart to the left
    1     Push cart to the right

    Note: The amount the velocity that is reduced or increased is not
    fixed; it depends on the angle the pole is pointing. This is because
    the center of gravity of the pole increases the amount of energy needed
    to move the cart underneath it

Reward:
    Reward is 1 for every step taken, including the termination step

Starting State:
    All observations are assigned a uniform random value in [-0.05..0.05]

Episode Termination:
    Pole Angle is more than 12 degrees.
    Cart Position is more than 2.4 (center of the cart reaches the edge of
    the display).
    Episode length is greater than 200.
    Solved Requirements:
    Considered solved when the average return is greater than or equal to
    195.0 over 100 consecutive trials.
"""
```

MountainCar-v0 description

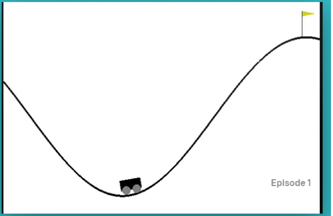
MountainCar-v0

A car is on a one-dimensional track, positioned between two "mountains". The goal is to drive up the mountain on the right; however, the car's engine is not strong enough to scale the mountain in a single pass. Therefore, the only way to succeed is to drive back and forth to build up momentum.

This problem was first described by Andrew Moore in his PhD thesis [Moore90].

[Moore90] *A Moore, Efficient Memory-Based Learning for Robot Control, PhD thesis, University of Cambridge, 1990.*

VIEW SOURCE ON GITHUB



RandomAgent on MountainCar-v0

MountainCar-v0 environment

```
Observation:
    Type: Box(2)
    Num    Observation               Min            Max
    0      Car Position              -1.2           0.6
    1      Car Velocity              -0.07          0.07

Actions:
    Type: Discrete(3)
    Num    Action
    0      Accelerate to the Left
    1      Don't accelerate
    2      Accelerate to the Right

    Note: This does not affect the amount of velocity affected by the
    gravitational pull acting on the car.

Reward:
     Reward of 0 is awarded if the agent reached the flag (position = 0.5)
     on top of the mountain.
     Reward of -1 is awarded if the position of the agent is less than 0.5.

Starting State:
     The position of the car is assigned a uniform random value in
     [-0.6 , -0.4].
     The starting velocity of the car is always assigned to 0.

Episode Termination:
     The car position is more than 0.5
     Episode length is greater than 200
"""
```

Initialisation

```
class BalanceEnv(gym.Env):
    metadata = {'render.modes': ['human']}

    def __init__(self):
        self.seed()
        self.state = self.reset()
        self.action_space = spaces.Discrete(N)
        self.observation_space = spaces.Box(
            low=np.array([0.0] * (4 + 2*N)),
            high=np.array([N,N*R,N*R,N*(R+10)] + [R,R+10] * N),dtype=np.uint32)
```

Where N is the maximum number of items and R the maximum weight for an item

Defining the knapsack environment

Create knapsack instance

```
def reset(self):
    self.items = []
    self.C = 0
    self.nbItems = int(random.randint(25,50))
    for i in range(self.nbItems):
        self.items.append(random.randint(1,R))
        self.C += self.items[-1]
        self.items.append(self.items[-1] + 10)
    self.C = int(self.C * (0.1 + random.random() * 0.8))
    for i in range(self.nbItems,N):
        self.items.append(0)
        self.items.append(0)

    self.total_reward = 0
    self.total_weight = 0
    self.state = self._update_state()
    self.greedy = greedy(self.items,self.C)
    return self.state
```

Where "greedy" computes the solution obtained with the greedy approach (add items with highest profit per unit first)

Defining the knapsack environment

Perform an action

```
def step(self, action):
    reward = 0
    done = False
    if action >= N:
        raise ValueError('{} is an invalid action. Must be between {} and {}'.format(
            action, 0, N))
    else:
        self.total_weight += self.items[2*action]
        if self.items[2*action] == 0:
            reward = 0
            done = True
        elif self.total_weight > self.C:
            reward = 0
            done = True
        else:
            reward = self.items[2*action + 1]
            self.nbItems -= 1
            self.items[2*action] = 0
            self.items[2*action + 1] = 0
            self.items[2*action] , self.items[2*self.nbItems] =  self.items[2*self.nbItems] , self.items[2*action]
            self.items[2*action + 1], self.items[2*self.nbItems + 1] =self.items[2*self.nbItems + 1] , self.items[2*action + 1]

        self.total_reward += reward
        self.state = self._update_state()

    return self.state, reward, done, {}
```

Where “action” is the index of the item we add to the knapsack

Defining the knapsack environment

Define s' once the action is chosen and graphical representation

```python
def _update_state(self):
    tempS = []
    tempS.append(self.nbItems)
    sumW = sum(self.items[2*i] for i in range(N))
    sumP = sum(self.items[2*i+1] for i in range(N))
    tempS.append(self.C - self.total_weight)
    tempS.append(sumW)
    tempS.append(sumP)
    state = np.array(tempS + self.items)
    return state

def render(self, mode='human', close=False):
    # Render the environment to the screen
    print(round(self.total_reward,3),"(",round(self.total_reward - self.greedy,3),")", round(self.total_weight/self.C,3))
    """if self.total_reward - self.greedy > 0:
        print(self.initialItem)
        print(self.solution)
        print(self.greedySol)"""
```

Where "render" is a function called in the main program to check the current state of the episode (here the function only makes sense for the final state)

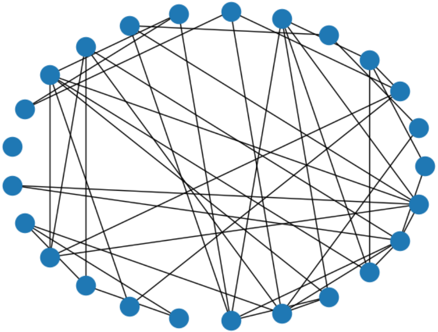
How can we build a suitable environment for a graph related problem?

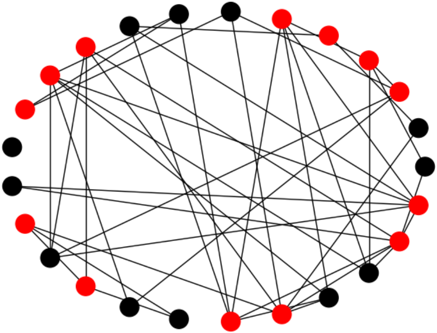
Plan

A DQL for the Minimum Vertex Cover

The Minimum Vertex Cover

Given a graph $G = (V, E)$, the Minimum Vertex Cover consists in finding the set of vertices $S \subseteq V$ of minimum cardinality such that every edge is covered, or in other words, that $\forall (i, j) \in E$, either $i \in S$ or $j \in S$





Activity

Discuss 5 minutes with your neighbour about

- Easy deterministic strategies to find good quality solutions for the minimum vertex cover
- Solving the MVC with DQL
  - what would be the initial, intermediary, and end states?
  - what would be the actions and their rewards?

A DQL for the Minimum Vertex Cover

How should we define a state?

- The adjacency matrix?
- But what if we change the node ordering?
- And what if we change the number of nodes?

One possible model for the problem

- Use a state with $V_{max} + 9$ integers where V_{max} is an upper bound on the maximum number of nodes where the first V_{max} numbers indicate the number of nodes with degree $0, \dots, V_{max} - 1$
- The 9 following numbers report some information about the instance such as the number of nodes currently in the solution, the average degree of a node, whether or not there is at least one node with degree $x (x = 1, \dots, 4)$, the maximum degree of a node, the number of nodes with degree at least one, and the the average degree of a node excluding those with degree 0

A DQL for the Minimum Vertex Cover

One possible model for the problem

- There are 4 possible actions in every state:
  - Add the node with the highest degree
  - Add the neighbor of a node with the lowest degree
  - Add the node with the highest support (sum of neighbor's degree)
  - Add the neighbor of the node with lowest support
- Every action has a -1 reward

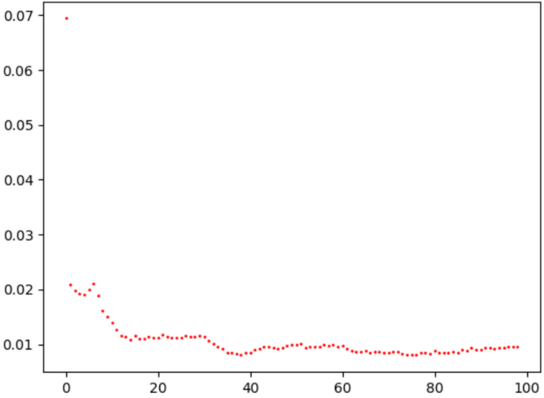
Parameters chosen for the DQN are unchanged

Parameters chosen for the MVC instance generation

- Max. number of nodes $V_{max} = 50$, $|V|$ rand. dist. in $[\frac{V_{max}}{2}; V_{max}]$
- Maximum number of episodes $E = 5,000$
- The probability for a pair of nodes to have an edge is p
- p is randomly distributed in $[0.15; 0.95]$

A DQL for the Minimum Vertex Cover

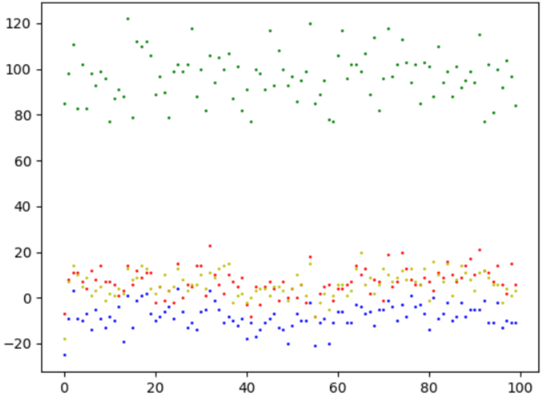
Evolution of the loss function of the neural network used to approximate the q-values in the DQN (groups of 50 instances)



After training, the neural network learnt how to approximate the q-values

A DQL for the Minimum Vertex Cover

Evolution of the profit difference between the solution obtained by the trained agent (groups of 50 instances) and each of the 4 greedy solutions



The agent learnt how to obtain reasonably good solutions, but could not surpass the best greedy heuristic

A DQL for the Minimum Vertex Cover

What happened?

- Maybe the network needed more training (some algorithms train for 2 weeks with 600,000 episodes)
- Maybe the network needed another architecture
- Maybe the way we defined the states did not provide enough information to learn relevant patterns

What other researchers do?

- Dai et al. mentioned that "Both the state of the graph and the context of a node v can be very complex, hard to describe in closed form, and may depend on complicated statistics such as global/local degree distribution, triangle counts, distance to tagged nodes, etc. In order to represent such complex phenomena over combinatorial structures, we will leverage a deep learning architecture over graphs, in particular the structure2vec of [9]"

Graph encoders compute a p-dimensional vector for every node
In the next lecture, we will talk about DeepWalk