DQN_project_MVA

April 19, 2018

You may need to install OpenCV and scikit-video.

```
In [55]: import keras
         import random
         import numpy as np
         import io
         import base64
         from IPython.display import HTML
         import skvideo.io
         import cv2
         import json
         import collections

         from keras.models import Sequential,model_from_json
         from keras.layers.core import Dense
         from keras.optimizers import sgd
         from keras.models import Model
         from keras.layers import Input, Dense, Conv2D, MaxPooling2D, Activation, AveragePooli
```

1 MiniProject #3: Deep Reinforcement Learning

Notations: E_p is the expectation under probability p. Please justify each of your answer and widely comment your code.

2 Context

In a reinforcement learning algorithm, we modelize each step t as an action a_t obtained from a state s_t, i.e. $\{(a_t, s_t)_{t \leq T}\}$ having the Markov property. We consider a discount factor $\gamma \in [0, 1]$ that ensures convergence. The goal is to find among all the policies π, one that maximizes the expected reward:

$$R(\pi) = \sum_{t \leq T} E_{p^\pi}[\gamma^t r(s_t, a_t)]\,,$$

where:

$$p^\pi(a_0, a_1, s_1, ..., a_T, s_T) = p(a_0) \prod_{t=1}^{T} \pi(a_t|s_t) p(s_{t+1}|s_t, a_t)\,.$$

We note the Q-function:

$$Q^{\pi}(s,a) = E_{p^{\pi}}[\sum_{t \leq T} \gamma^t r(s_t, a_t) | s_0 = s, a_0 = a] \,.$$

Thus, the optimal Q function is:

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a) \,.$$

In this project, we will apply the deep reinforcement learning techniques to a simple game: an agent will have to learn from scratch a policy that will permit it maximizing a reward.

2.1 The environment, the agent and the game

2.1.1 The environment

`Environment` is an abstract class that represents the states, rewards, and actions to obtain the new state.

```
In [56]: class Environment(object):
             def __init__(self):
                 pass

             def act(self, act):
                 """
                 One can act on the environment and obtain its reaction:
                 - the new state
                 - the reward of the new state
                 - should we continue the game?

                 self.state =
                 self.reward =
                 self.game_over =

                 return: state, reward, game_over
                 """
                 pass


             def reset(self):
                 """
                 Reinitialize the environment to a random state and returns
                 the original state

                 return: state
                 """
                 pass

             def draw(self):
                 """
```

```
        Visualize in the console or graphically the current state
        """
        pass
```

The method `act` allows to act on the environment at a given state s_t (stored internally), via action a_t. The method will return the new state s_{t+1}, the reward $r(s_t, a_t)$ and determines if $t \leq T$ (*game_over*).

The method `reset` simply reinitializes the environment to a random state s_0.

The method `draw` displays the current state s_t (this is useful to check the behavior of the Agent).

We modelize s_t as a tensor, while a_t is an integer.

2.1.2 The Agent

The goal of the `Agent` is to interact with the `Environment` by proposing actions a_t obtained from a given state s_t to attempt to maximize its **reward** $r(s_t, a_t)$. We propose the following abstract class:

```
In [57]: class Agent(object):
             def __init__(self, epsilon=0.1, n_action=4, epsilon_min=0.01, epsilon_decay = 0.99
                 self.epsilon = epsilon
                 self.n_action = n_action
                 self.epsilon_min = epsilon_min
                 self.epsilon_decay = epsilon_decay

             def set_epsilon(self):

                 if self.epsilon > self.epsilon_min:
                     self.epsilon *= self.epsilon_decay

                 return self.epsilon

             def act(self,s,train=True):
                 """ This function should return the next action to do:
                 an integer between 0 and 4 (not included) with a random exploration of epsilo
                 if train:
                     if np.random.rand() <= self.epsilon:
                         a = np.random.randint(0, self.n_action, size=1)[0]
                     else:
                         a = self.learned_act(s)
                 else: # in some cases, this can improve the performance.. remove it if poor p
                     a = self.learned_act(s)

                 return a

             def learned_act(self,s):
                 """ Act via the policy of the agent, from a given state s
                 it proposes an action a"""
                 pass
```

```
    def reinforce(self, s, n_s, a, r, game_over_):
        """ This function is the core of the learning algorithm.
        It takes as an input the current state s_, the next state n_s_
        the action a_ used to move from s_ to n_s_ and the reward r_.

        Its goal is to learn a policy.
        """
        pass

    def save(self):
        """ This function returns basic stats if applicable: the
        loss and/or the model"""
        pass

    def load(self):
        """ This function allows to restore a model"""
        pass
```

Question 1: Explain the function act. Why is `epsilon` essential?

act() returns the next action to do.

act() will choose an action randomly with probability epsilon. This enables the agent to explore the board and not to explore too much too soon. Otherwise, with a greedy approach, the agent would have no knowledge of the board to make better decisions in the future.

act() will choose to act according to what it has learned with probability 1-epsilon (greedy approach).

Epsilon is essential because it determines how much exploring vs exploiting the agent will do. There is a balance to find. The agent cannot just be greedy and just exploit because that would be detrimental in the long run.

2.1.3 The Game

The `Agent` and the `Environment` work in an interlaced way as in the following (take some time to understand this code as it is the core of the project)

```
epoch = 300
env = Environment()
agent = Agent()


# Number of won games
score = 0
loss = 0
```

```
for e in range(epoch):
    # At each epoch, we restart to a fresh game and get the initial state
    state = env.reset()
    # This assumes that the games will end
    game_over = False

    win = 0
    lose = 0

    while not game_over:
        # The agent performs an action
        action = agent.act(state)

        # Apply an action to the environment, get the next state, the reward
        # and if the games end
        prev_state = state
        state, reward, game_over = env.act(action)

        # Update the counters
        if reward > 0:
            win = win + reward
        if reward < 0:
            lose = lose -reward

        # Apply the reinforcement strategy
        loss = agent.reinforce(prev_state, state,  action, reward, game_over)

    # Save as a mp4
    if e % 10 == 0:
        env.draw(e)

    # Update stats
    score += win-lose

    print("Epoch {:03d}/{:03d} | Loss {:.4f} | Win/lose count {}/{} ({})"
          .format(e, epoch, loss, win, lose, win-lose))
    agent.save()
```

3 The game, *eat cheese*

A rat runs on an island and tries to eat as much as possible. The island is subdivided into $N \times N$ cells, in which there are cheese (+0.5) and poisonous cells (-1). The rat has a visibility of 2 cells (thus it can see 5^2 cells). The rat is given a time T to accumulate as much food as possible. It can perform 4 actions: going up, down, left, right.

The goal is to code an agent to solve this task that will learn by trial and error. We propose the following environment:

```
In [58]: class Environment(object):
```

```
def __init__(self, grid_size=10, max_time=500, temperature=0.1):
    grid_size = grid_size+4
    self.grid_size = grid_size
    self.max_time = max_time
    self.temperature = temperature

    #board on which one plays
    self.board = np.zeros((grid_size,grid_size))
    self.position = np.zeros((grid_size,grid_size))

    # coordinate of the cat
    self.x = 0
    self.y = 1

    # self time
    self.t = 0

    self.scale=16

    self.to_draw = np.zeros((max_time+2, grid_size*self.scale, grid_size*self.scal


def draw(self,e):
    skvideo.io.vwrite(str(e) + '.mp4', self.to_draw)

def get_frame(self,t):
    b = np.zeros((self.grid_size,self.grid_size,3))+128
    b[self.board>0,0] = 256
    b[self.board < 0, 2] = 256
    b[self.x,self.y,:]=256
    b[-2:,:,:]=0
    b[:,-2:,:]=0
    b[:2,:,:]=0
    b[:,:2,:]=0

    b =  cv2.resize(b, None, fx=self.scale, fy=self.scale, interpolation=cv2.INTE

    self.to_draw[t,:,:,:]=b


def act(self, action):
    """This function returns the new state, reward and decides if the
    game ends."""

    self.get_frame(int(self.t))

    self.position = np.zeros((self.grid_size, self.grid_size))
```

```
        self.position[0:2,:]= -1
        self.position[:,0:2] = -1
        self.position[-2:, :] = -1
        self.position[-2:, :] = -1

        self.position[self.x, self.y] = 1
        if action == 0:
            if self.x == self.grid_size-3:
                self.x = self.x-1
            else:
                self.x = self.x + 1
        elif action == 1:
            if self.x == 2:
                self.x = self.x+1
            else:
                self.x = self.x-1
        elif action == 2:
            if self.y == self.grid_size - 3:
                self.y = self.y - 1
            else:
                self.y = self.y + 1
        elif action == 3:
            if self.y == 2:
                self.y = self.y + 1
            else:
                self.y = self.y - 1
        else:
            RuntimeError('Error: action not recognized')

        self.t = self.t + 1
        reward = self.board[self.x, self.y]
        self.board[self.x, self.y] = 0
        game_over = self.t > self.max_time
        state = np.concatenate((self.board.reshape(self.grid_size, self.grid_size,1),
                        self.position.reshape(self.grid_size, self.grid_size,1)),axis=
        state = state[self.x-2:self.x+3,self.y-2:self.y+3,:]

        return state, reward, game_over

    def reset(self):
        """This function resets the game and returns the initial state"""

        self.x = np.random.randint(3, self.grid_size-3, size=1)[0]
        self.y = np.random.randint(3, self.grid_size-3, size=1)[0]


        bonus = 0.5*np.random.binomial(1,self.temperature,size=self.grid_size**2)
        bonus = bonus.reshape(self.grid_size,self.grid_size)
```

```
        malus = -1.0*np.random.binomial(1,self.temperature,size=self.grid_size**2)
        malus = malus.reshape(self.grid_size, self.grid_size)

        self.to_draw = np.zeros((self.max_time+2, self.grid_size*self.scale, self.gri

        malus[bonus>0]=0

        self.board = bonus + malus

        self.position = np.zeros((self.grid_size, self.grid_size))
        self.position[0:2,:]= -1
        self.position[:,0:2] = -1
        self.position[-2:, :] = -1
        self.position[-2:, :] = -1
        self.board[self.x,self.y] = 0
        self.t = 0

        state = np.concatenate((
                                self.board.reshape(self.grid_size, self.grid_size,1),
                        self.position.reshape(self.grid_size, self.grid_size,1)),axis=

        state = state[self.x - 2:self.x + 3, self.y - 2:self.y + 3, :]
        return state
```

The following elements are important because they correspond to the hyper parameters for this project:

```
In [59]: # parameters
         size = 13
         T=200
         temperature=0.3
         epochs_train=1 # set small when debugging
         epochs_test=1 # set small when debugging

         # display videos
         def display_videos(name):
             video = io.open(name, 'r+b').read()
             encoded = base64.b64encode(video)
             return '''<video alt="test" controls>
                         <source src="data:video/mp4;base64,{0}" type="video/mp4" />
                       </video>'''.format(encoded.decode('ascii'))
```

Question 2 Explain the use of the arrays `position` and `board`.

The array `position` represents the coordinates of the rat on the board. If the rat is on the cell of coordinates (4,5), the matrix of dimension (board_size, board_size) will take value 1 at position (4,5). The rest will be zeros. The array has values -1 on the 2-cell edges of the board.

The array `board` represents every possible position (coordinates x and y) on the virtual board, that is, every possible place the rat could be. The cells have different values corresponding to colors which each represent poison (blue), cheese (red) or an empty cell (grey). Every time the agent goes to a new position, the cell becomes (or stays) grey to indicate the rat has eaten the poison or the cheese (or nothing).

3.1 Random Agent

Question 3 Implement a random Agent (only `learned_act` needs to be implemented):

```
In [60]: class RandomAgent(Agent):
             def __init__(self):
                 super(RandomAgent, self).__init__()
                 pass

             def learned_act(self, s):
                 return np.random.randint(0, self.n_action)
```

Question 4 Visualize the game moves. You need to fill in the following function for the evaluation:

```
In [73]: def test(agent,env,epochs,prefix=''):

             # Number of won games
             score = 0
             loss = 0

             for e in range(epochs):

                 ##### FILL IN HERE
                 # At each epoch, we restart to a fresh game and get the initial state
                 state = env.reset()
                 # This assumes that the games will end
                 game_over = False

                 win = 0
                 lose = 0

                 while not game_over:
                     # The agent performs an action
                     action = agent.act(state)

                     # Apply an action to the environment, get the next state, the reward
```

```
                    # and if the games end
                    prev_state = state
                    state, reward, game_over = env.act(action)

                    # Update the counters
                    if reward > 0:
                        win = win + reward
                    if reward < 0:
                        lose = lose -reward

                    # Apply the reinforcement strategy
                    #loss = agent.reinforce(prev_state, state,  action, reward, game_over)

                # Update stats
                #score += win-lose

                # Save as a mp4
                env.draw(prefix+str(e))

                # Update stats
                score = score + win-lose

                #return e, epochs, loss, win, lose, win-lose #, env.to_draw
                print("Win/lose count {:.2f}/{:.2f}. Average score ({:.2f})"
                      .format(win, lose, score/(1+e)))
            print('Final score: '+str(round(score/epochs,2)))
```

```
In [62]: # Initialize the game
         import skvideo.io
         import skvideo.datasets
         import skvideo.utils
         env = Environment(grid_size=size, max_time=T,temperature=temperature)

         # Initialize the agent!
         agent = RandomAgent()

         test(agent,env,epochs_test,prefix='random')
         #HTML(display_videos('random0.mp4'))
         #print(test(agent,env,epochs_test,prefix='random'))
```

```
Win/lose count 11.50/16.00. Average score (-4.50)
Final score: -4.5
```

The score is completely random.

3.2 DQN

Let us assume here that $T = \infty$.

Question 5 Let π be a policy, show that:

$$Q^{\pi}(s,a) = E_{(s',a')\sim p(.|s,a)}[r(s,a) + \gamma Q^{\pi}(s',a')]$$

Then, show that for the optimal policy π^* (we assume its existence), the following holds:

$$Q^*(s,a) = E_{s'\sim \pi^*(.|s,a)}[r(s,a) + \gamma \max_{a'} Q^*(s',a')].$$

Finally, deduce that a plausible objective is:

$$\mathcal{L}(\theta) = E_{s'\sim \pi^*(.|s,a)}\|r + \gamma \max \max_{a'} Q(s',a',\theta) - Q(s,a,\theta)\|^2.$$

Answer
(please see appendix for full proof)
We note the Q-function:

$$Q^{\pi}(s,a) = E_{p^{\pi}}[\sum_{t\leq T} \gamma^t r(s_t,a_t)|s_0 = s, a_0 = a]\,.$$

We can write Q recursively to get a Bellman equation.

$$Q^{\pi}(s,a) = E_{(s',a')\sim p(.|s,a)}[r(s,a) + \gamma \sum_{t\leq T} \gamma^t r(s_{t+1},a_{t+1})|s_0 = s, a_0 = a]$$

$$Q^{\pi}(s,a) = E_{(s',a')\sim p(.|s,a)}[r(s,a) + \gamma Q^{\pi}(s',a')]\,.$$

Where s' and a' are the state and the action in the period following that of s and a.
Given that the optimal Q function is:

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a)\,.$$

By plugging this into the Bellman equation, and by linearity of the maximum, we can write:

$$Q^*(s,a) = E_{s'\sim \pi^*(.|s,a)}[r(s,a) + \gamma \max_{a'} Q^*(s',a')].$$

Therefore, taking a mean squared error loss, a plausible objective function is:

$$\mathcal{L}(\theta) = E_{s'\sim \pi^*(.|s,a)}\|r + \gamma \max \max_{a'} Q(s',a',\theta) - Q(s,a,\theta)\|^2.$$

We try to choose Q such that it is as close as possible to the optimal strategy.

The DQN-learning algorithm relies on these derivations to train the parameters θ of a Deep Neural Network:

1. At the state s_t, select the action a_t with best reward using Q_t and store the results;

2. Obtain the new state s_{t+1} from the environment p;
3. Store (s_t, a_t, s_{t+1});
4. Obtain Q_{t+1} by minimizing $\mathcal{L}$ from a recovered batch from the previously stored results.

Question 6 Implement the class `Memory` that stores moves (in a replay buffer) via `remember` and provides a `random_access` to these. Specify a maximum memory size to avoid side effects. You can for example use a `list()` and set by default `max_memory=100`.

```
In [63]: class Memory(object):
             def __init__(self, max_memory=100):
                 self.max_memory = max_memory
                 self.memory = list()

             def remember(self, m):
                 self.memory.append(m)
                 self.memory = collections.deque(self.memory, maxlen=self.max_memory)
                 # m is a list of state(t), state(t+1), action, reward, game_over_

             def random_access(self):
                 return random.choice(self.memory)
```

The pipeline we will use for training is given below:

```
In [64]: def train(agent,env,epoch,prefix=''):
             # Number of won games
             score = 0
             loss = 0

             for e in range(epoch):
                 # At each epoch, we restart to a fresh game and get the initial state
                 state = env.reset()

                 # This assumes that the games will terminate
                 game_over = False

                 win = 0
                 lose = 0

                 while not game_over:
                     # The agent performs an action
                     action = agent.act(state)

                     # Apply an action to the environment, get the next state, the reward
```

```
            # and if the games end
            prev_state = state
            state, reward, game_over = env.act(action)

            # Update the counters
            if reward > 0:
                win = win + reward
            if reward < 0:
                lose = lose -reward

            # Apply the reinforcement strategy
            loss = agent.reinforce(prev_state, state,  action, reward, game_over)

        # Save as a mp4
        if e % 10 == 0:
            env.draw(prefix+str(e))

        # Update stats
        score += win-lose

        print("Epoch {:03d}/{:03d} | Loss {:.4f} | Win/lose count {:.2f}/{:.2f} ({:.2
              .format(e+1, epoch, loss, win, lose, win-lose))
        agent.save(name_weights=prefix+'model.h5',name_model=prefix+'model.json')
```

Question 7 Implement the DQN training algorithm using a cascade of fully connected layers. You can use different learning rate, batch size or memory size parameters. In particular, the loss might oscillate while the player will start to win the games. You have to find a good criterium.

```
In [65]: class DQN(Agent):
             def __init__(self, grid_size, epsilon = 0.1, memory_size=100, batch_size = 64, n_s
                 super(DQN, self).__init__(epsilon = epsilon, epsilon_min=epsilon_min, epsilon_

                 #, epsilon_min=epsilon_min, epsilon_decay=epsilon_decay

                 # Discount for Q learning
                 self.discount = 0.99

                 self.grid_size = grid_size

                 # number of state
                 self.n_state = n_state

                 # Memory
                 self.memory = Memory(memory_size)

                 # Batch size when learning
```

```
        self.batch_size = batch_size

    def learned_act(self, s):
        # returns act learned by neural network model

        act_pred = self.model.predict(np.reshape(s, (1, 5, 5, self.n_state)))

        return np.argmax(act_pred[0])
        #pass

    def reinforce(self, s_, n_s_, a_, r_, game_over_):
        # Two steps: first memorize the states, second learn from the pool

        self.memory.remember([s_, n_s_, a_, r_, game_over_])

        # initialise input states and targets
        input_states = np.zeros((self.batch_size, 5, 5, self.n_state))
        target_q = np.zeros((self.batch_size, 4))

        for i in range(self.batch_size):
            ######## FILL IN

            batch=self.memory.random_access()
            input_states[i]=batch[0] # state observed in t; batch[1] is the state in
            game_over_=batch[4] # the fifth element of the batch says if game over or

            if game_over_:
                ######## FILL IN
                target_q[i, batch[2]] = batch[3] # batch[2] is the action, bacth[3] i
            else:
                ######## FILL IN
                x = np.reshape(batch[1], (1, 5, 5, self.n_state)) # get observered st
                target_q[i, batch[2]] = batch[3] + self.discount * np.amax(self.model

        ######## FILL IN
        # HINT: Clip the target to avoid exploiding gradients.. -- clipping is a bit

        target_q = np.clip(target_q, -3, 3)
        l = self.model.train_on_batch(input_states, target_q)

        return l

    def save(self,name_weights='model.h5',name_model='model.json'):
        self.model.save_weights(name_weights, overwrite=True)
        with open(name_model, "w") as outfile:
            json.dump(self.model.to_json(), outfile)

    def load(self,name_weights='model.h5',name_model='model.json'):
```

```
            with open(name_model, "r") as jfile:
                model = model_from_json(json.load(jfile))
            model.load_weights(name_weights)
            model.compile("sgd", "mse")
            self.model = model


    class DQN_FC(DQN):
        def __init__(self, *args, lr=0.1,**kwargs):
            super(DQN_FC, self).__init__( *args,**kwargs)

            # NN Model

            ####### FILL IN

            model = Sequential()
            model.add(Reshape((5*5*self.n_state,), input_shape=(5,5,self.n_state)))
            model.add(Dense(24, input_dim=50, activation='relu'))
            model.add(Dense(24, activation='relu'))
            model.add(Dense(4, activation='linear')) # keep four or set var = 4?
            model.compile(sgd(lr=lr, decay=1e-4, momentum=0.0), "mse")
            self.model = model
```

In [66]:
```
env = Environment(grid_size=size, max_time=T, temperature=0.3)
agent = DQN_FC(size, lr=.5, epsilon = 0.5, memory_size=10000, batch_size = 64) # epsi
train(agent, env, 20, prefix='fc_train') #epochs_train
#HTML(display_videos('fc_train' + str(e) + '.mp4'))
```

```
Epoch 001/020 | Loss 0.0071 | Win/lose count 5.00/9.00 (-4.00)
Epoch 002/020 | Loss 0.0123 | Win/lose count 8.50/15.00 (-6.50)
Epoch 003/020 | Loss 0.0065 | Win/lose count 5.50/5.00 (0.50)
Epoch 004/020 | Loss 0.0201 | Win/lose count 10.00/11.00 (-1.00)
Epoch 005/020 | Loss 0.0181 | Win/lose count 10.50/13.00 (-2.50)
Epoch 006/020 | Loss 0.0124 | Win/lose count 10.50/5.00 (5.50)
Epoch 007/020 | Loss 0.0136 | Win/lose count 7.50/11.00 (-3.50)
Epoch 008/020 | Loss 0.0104 | Win/lose count 5.00/4.00 (1.00)
Epoch 009/020 | Loss 0.0178 | Win/lose count 7.50/7.00 (0.50)
Epoch 010/020 | Loss 0.0167 | Win/lose count 12.50/14.00 (-1.50)
Epoch 011/020 | Loss 0.0155 | Win/lose count 11.00/10.00 (1.00)
Epoch 012/020 | Loss 0.0206 | Win/lose count 11.00/12.00 (-1.00)
Epoch 013/020 | Loss 0.0146 | Win/lose count 13.50/10.00 (3.50)
Epoch 014/020 | Loss 0.0235 | Win/lose count 11.50/13.00 (-1.50)
Epoch 015/020 | Loss 0.0162 | Win/lose count 15.50/11.00 (4.50)
Epoch 016/020 | Loss 0.0108 | Win/lose count 4.00/10.00 (-6.00)
Epoch 017/020 | Loss 0.0183 | Win/lose count 13.00/7.00 (6.00)
Epoch 018/020 | Loss 0.0219 | Win/lose count 7.00/8.00 (-1.00)
Epoch 019/020 | Loss 0.0212 | Win/lose count 12.00/14.00 (-2.00)
Epoch 020/020 | Loss 0.0188 | Win/lose count 7.50/10.00 (-2.50)
```

There is a lot of volatility and we do not see a clear improvement with the nubmer of epochs.

Question 8 Implement the DQN training algorithm using a CNN (for example, 2 convolutional layers and one final fully connected layer).

```
In [67]: class DQN_CNN(DQN):
             def __init__(self, *args,lr=0.1,**kwargs):
                 super(DQN_CNN, self).__init__(*args,**kwargs)

                 ###### FILL IN"

                 model = Sequential()
                 model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(5, 5, self.n_sta
                 model.add(Conv2D(32, (3, 3), activation='relu'))
                 model.add(MaxPooling2D(pool_size=(1, 1)))
                 model.add(Flatten())
                 model.add(Dense(4, activation='softmax'))

                 model.compile(sgd(lr=lr, decay=1e-4, momentum=0.0), "mse")
                 self.model = model


In [68]: env = Environment(grid_size=size, max_time=T, temperature=0.3)
         agent = DQN_CNN(size, lr=.5, epsilon = 0.5, memory_size=10000, batch_size = 64) #0.3
         train(agent,env,20,prefix='cnn_train') #epochs_train
         #HTML(display_videos('cnn_train10.mp4'))

Epoch 001/020 | Loss 0.0578 | Win/lose count 6.50/5.00 (1.50)
Epoch 002/020 | Loss 0.0595 | Win/lose count 3.50/2.00 (1.50)
Epoch 003/020 | Loss 0.0431 | Win/lose count 6.50/3.00 (3.50)
Epoch 004/020 | Loss 0.0521 | Win/lose count 5.50/4.00 (1.50)
Epoch 005/020 | Loss 0.0548 | Win/lose count 4.50/8.00 (-3.50)
Epoch 006/020 | Loss 0.0431 | Win/lose count 4.50/9.00 (-4.50)
Epoch 007/020 | Loss 0.0510 | Win/lose count 3.50/7.00 (-3.50)
Epoch 008/020 | Loss 0.0540 | Win/lose count 6.00/7.00 (-1.00)
Epoch 009/020 | Loss 0.0454 | Win/lose count 5.00/3.00 (2.00)
Epoch 010/020 | Loss 0.0450 | Win/lose count 3.00/6.00 (-3.00)
Epoch 011/020 | Loss 0.0570 | Win/lose count 5.00/5.00 (0.00)
Epoch 012/020 | Loss 0.0516 | Win/lose count 6.00/4.00 (2.00)
Epoch 013/020 | Loss 0.0498 | Win/lose count 4.50/3.00 (1.50)
Epoch 014/020 | Loss 0.0566 | Win/lose count 6.00/6.00 (0.00)
Epoch 015/020 | Loss 0.0483 | Win/lose count 3.50/9.00 (-5.50)
Epoch 016/020 | Loss 0.0560 | Win/lose count 7.50/9.00 (-1.50)
Epoch 017/020 | Loss 0.0560 | Win/lose count 5.00/5.00 (0.00)
Epoch 018/020 | Loss 0.0525 | Win/lose count 1.50/5.00 (-3.50)
```

```
Epoch 019/020 | Loss 0.0481 | Win/lose count 6.00/5.00 (1.00)
Epoch 020/020 | Loss 0.0537 | Win/lose count 6.00/6.00 (0.00)
```

As for the FC agent, there is a lot of volatility. The higher epsilon, the higher the volatility.

Question 9 Test both algorithms and compare their performances. Which issue(s) do you observe? Observe also different behaviors by changing the temperature.

```
In [74]: env = Environment(grid_size=size, max_time=T,temperature=0.5)
         agent_cnn = DQN_CNN(size, lr=.1, epsilon = 0.1, memory_size=2000, batch_size = 32)
         agent_cnn.load(name_weights='cnn_trainmodel.h5',name_model='cnn_trainmodel.json')

         agent_fc = DQN_FC(size, lr=.1, epsilon = 0.1, memory_size=2000, batch_size = 32)
         agent_cnn.load(name_weights='fc_trainmodel.h5',name_model='fc_trainmodel.json')
         print('Test of the CNN')
         test(agent_cnn,env,5,prefix='cnn_test') #epochs_test
         print('Test of the FC')
         test(agent_fc,env,5,prefix='fc_test') #epochs_test
```

```
Test of the CNN
Win/lose count 24.00/9.00. Average score (15.00)
Win/lose count 19.00/8.00. Average score (13.00)
Win/lose count 8.50/0.00. Average score (11.50)
Win/lose count 12.00/5.00. Average score (10.38)
Win/lose count 5.50/5.00. Average score (8.40)
Final score: 8.4
Test of the FC
Win/lose count 9.00/5.00. Average score (4.00)
Win/lose count 3.00/6.00. Average score (0.50)
Win/lose count 5.50/5.00. Average score (0.50)
Win/lose count 3.50/1.00. Average score (1.00)
Win/lose count 2.00/4.00. Average score (0.40)
Final score: 0.4
```

```
In [70]: #HTML(display_videos('cnn_test10.mp4'))
```

```
In [71]: #HTML(display_videos('fc_test10.mp4'))
```

On average, the CNN agent seems better than the FC agent. One issue is that there is volatility in the test results. Also, epsilon doesn't change which means as the agent learns he doesn't reduce the proportion of randomness in his game.

The temperature corresponds to the parmater of the binomial distribution which determines the density of poison and cheese on the board. Malus cells are set to zero if a bonus was drawn at the same coordinates ("malus[bonus>0]=0"). So the higher the temperature, the higher the proportion of bonus and so the easier the game. Changing the temperature to 0.9 we see that the results are way better.

The algorithm tends to not explore the map which can be an issue. We propose two ideas in order to encourage exploration: 1. Incorporating a decreasing ϵ-greedy exploration. You can use the method `set_epsilon` 2. Append via the environment a new state that describes if a cell has been visited or not

---

Question 10 Design a new `train_explore` function and environment class `EnvironmentExploring` to tackle the issue of exploration.

```
In [75]: def train_explore(agent,env,epoch,prefix=''):

             # Number of won games
             score = 0
             loss = 0

             for e in range(epoch):
                 # At each epoch, we restart to a fresh game and get the initial state
                 state = env.reset()
                 # This assumes that the games will terminate
                 game_over = False

                 win = 0
                 lose = 0
                 malus_tot = 0

                 while not game_over:
                     # The agent performs an action
                     action = agent.act(state)

                     # Apply an action to the environment, get the next state, the reward
                     # and if the games end
                     prev_state = state
                     state, reward, game_over, malus = env.act(action)

                     # Update the counters
                     if reward > 0:
                         win = win + reward
                     if reward < 0:
                         lose = lose - reward

                     # total malus in game
                     malus_tot += malus

                     # Apply the reinforcement strategy
                     loss = agent.reinforce(prev_state, state,  action, reward, game_over)
```

```
            # Epsilon decay method
            #agent.set_epsilon()

        agent.set_epsilon()

        # Save as a mp4
        if e % 10 == 0:
            env.draw(prefix+str(e))

        # Update stats
        score += win - lose + malus_tot

        print("Epoch {:03d}/{:03d} | Malus {:.2f} | Loss {:.4f} | Win/lose count {:.2
              .format(e+1, epoch, malus_tot, loss, win, lose, win-lose+malus_tot))
        agent.save(name_weights=prefix+'model.h5',name_model=prefix+'model.json')


def test_explore(agent,env,epochs,prefix=''):

    # Number of won games
    score = 0
    loss = 0

    for e in range(epochs):

        ##### FILL IN HERE
        # At each epoch, we restart to a fresh game and get the initial state
        state = env.reset()
        # This assumes that the games will end
        game_over = False

        win = 0
        lose = 0
        malus_tot = 0

        while not game_over:
            # The agent performs an action
            action = agent.act(state)

            # Apply an action to the environment, get the next state, the reward
            # and if the games end
            prev_state = state
            state, reward, game_over, malus_score = env.act(action)

            # Update the counters
            if reward > 0:
                win = win + reward
            if reward < 0:
```

```
                lose = lose - reward

            #total malus in game
            malus_tot += malus_score

            # Apply the reinforcement strategy
            #loss = agent.reinforce(prev_state, state,  action, reward, game_over)

        # Update stats
        score += win - lose + malus_tot

        # Save as a mp4
        env.draw(prefix+str(e))

        #return e, epochs, loss, win, lose, win-lose #, env.to_draw
        print("Win/lose count {:.2f}/{:.2f}. Average real score {:.2f}. Malus {:.2f}."
              .format(win, lose, score/(1+e), malus_tot))
    print('Final score: '+str(round(score/epochs,2)))


class EnvironmentExploring(Environment):
    def __init__(self, grid_size=10, max_time=500, temperature=0.1, penalty=0.1):
        super(EnvironmentExploring, self).__init__(grid_size=grid_size, max_time=max_t

        self.malus_position = np.zeros((self.grid_size,self.grid_size))
        self.penalty = penalty

    def act(self, action, train=True):
        """This function returns the new state, reward and decides if the
        game ends."""

        self.get_frame(int(self.t))

        self.position = np.zeros((self.grid_size, self.grid_size))

        self.position[0:2,:]= -1
        self.position[:,0:2] = -1
        self.position[-2:, :] = -1
        self.position[-2:, :] = -1

        self.position[self.x, self.y] = 1
        if action == 0:
            if self.x == self.grid_size-3:
                self.x = self.x-1
            else:
                self.x = self.x + 1
        elif action == 1:
            if self.x == 2:
```

```
            self.x = self.x+1
        else:
            self.x = self.x-1
    elif action == 2:
        if self.y == self.grid_size - 3:
            self.y = self.y - 1
        else:
            self.y = self.y + 1
    elif action == 3:
        if self.y == 2:
            self.y = self.y + 1
        else:
            self.y = self.y - 1
    else:
        RuntimeError('Error: action not recognized')

    self.t = self.t + 1

    #
    malus_score = self.malus_position[self.x, self.y]

    reward = 0
    if train:
        reward = -self.malus_position[self.x, self.y]
    self.malus_position[self.x, self.y] = self.penalty
    # if cell (x, y) is explored, its value is "penalty" so that if the agent goe
    # he will be punished by substracting the penalty value from reward

    reward = reward + self.board[self.x, self.y]

    self.board[self.x, self.y] = 0
    game_over = self.t > self.max_time
    # 3 "feature" states instead of 2
    state = np.concatenate((self.malus_position.reshape(self.grid_size, self.grid
                                    self.board.reshape(self.grid_size, self.grid_s
                            self.position.reshape(self.grid_size, self.grid_size,
    state = state[self.x-2:self.x+3,self.y-2:self.y+3,:]
    return state, reward, game_over, malus_score


def reset(self):
    """This function resets the game and returns the initial state"""

    self.x = np.random.randint(3, self.grid_size-3, size=1)[0]
    self.y = np.random.randint(3, self.grid_size-3, size=1)[0]


    bonus = 0.5*np.random.binomial(1,self.temperature,size=self.grid_size**2)
```

```
        bonus = bonus.reshape(self.grid_size,self.grid_size)

        malus = -1.0*np.random.binomial(1,self.temperature,size=self.grid_size**2)
        malus = malus.reshape(self.grid_size, self.grid_size)

        self.to_draw = np.zeros((self.max_time+2, self.grid_size*self.scale, self.gri


        malus[bonus>0]=0

        self.board = bonus + malus

        self.malus_position = np.zeros((self.grid_size, self.grid_size))
        self.position = np.zeros((self.grid_size, self.grid_size))
        self.position[0:2,:]= -1
        self.position[:,0:2] = -1
        self.position[-2:, :] = -1
        self.position[-2:, :] = -1
        self.board[self.x,self.y] = 0
        self.t = 0

        state = np.concatenate((self.malus_position.reshape(self.grid_size, self.grid_
                                self.board.reshape(self.grid_size, self.grid_size,1),
                            self.position.reshape(self.grid_size, self.grid_size,1)),axis=

        state = state[self.x - 2:self.x + 3, self.y - 2:self.y + 3, :]
        return state
```

In [77]:
```
# Training
env = EnvironmentExploring(grid_size=size, max_time=T, temperature=0.3, penalty=0.5) #
agent = DQN_CNN(size, lr=.5, epsilon = 0.9, memory_size=10000, batch_size = 64, n_stat
train_explore(agent, env, 40, prefix='cnn_train_explore') #epochs_train
#HTML(display_videos('cnn_train_explore10.mp4'))
```

```
Epoch 001/040 | Malus 77.00 | Loss 0.0947 | Win/lose count 6.50/90.00 | Real score -6.50
Epoch 002/040 | Malus 54.50 | Loss 0.1064 | Win/lose count 13.00/71.50 | Real score -4.00
Epoch 003/040 | Malus 51.50 | Loss 0.0889 | Win/lose count 15.00/69.50 | Real score -3.00
Epoch 004/040 | Malus 53.00 | Loss 0.0960 | Win/lose count 16.50/68.00 | Real score 1.50
Epoch 005/040 | Malus 46.00 | Loss 0.0963 | Win/lose count 16.00/65.00 | Real score -3.00
Epoch 006/040 | Malus 57.00 | Loss 0.0915 | Win/lose count 16.00/67.00 | Real score 6.00
Epoch 007/040 | Malus 46.50 | Loss 0.0911 | Win/lose count 20.00/57.50 | Real score 9.00
Epoch 008/040 | Malus 62.00 | Loss 0.0892 | Win/lose count 14.00/72.00 | Real score 4.00
Epoch 009/040 | Malus 39.00 | Loss 0.0896 | Win/lose count 21.50/49.00 | Real score 11.50
Epoch 010/040 | Malus 42.50 | Loss 0.1017 | Win/lose count 21.00/51.50 | Real score 12.00
Epoch 011/040 | Malus 41.50 | Loss 0.0823 | Win/lose count 23.00/52.50 | Real score 12.00
Epoch 012/040 | Malus 58.00 | Loss 0.0750 | Win/lose count 18.00/65.00 | Real score 11.00
Epoch 013/040 | Malus 47.00 | Loss 0.0867 | Win/lose count 16.00/58.00 | Real score 5.00
Epoch 014/040 | Malus 56.50 | Loss 0.0897 | Win/lose count 16.00/67.50 | Real score 5.00
```

```
Epoch 015/040 | Malus 49.50 | Loss 0.0746 | Win/lose count 14.00/65.50 | Real score -2.00
Epoch 016/040 | Malus 62.00 | Loss 0.0817 | Win/lose count 15.00/71.00 | Real score 6.00
Epoch 017/040 | Malus 72.50 | Loss 0.0758 | Win/lose count 13.50/76.50 | Real score 9.50
Epoch 018/040 | Malus 44.00 | Loss 0.0974 | Win/lose count 15.00/53.00 | Real score 6.00
Epoch 019/040 | Malus 56.50 | Loss 0.0852 | Win/lose count 18.50/60.50 | Real score 14.50
Epoch 020/040 | Malus 35.00 | Loss 0.0757 | Win/lose count 27.50/43.00 | Real score 19.50
Epoch 021/040 | Malus 49.50 | Loss 0.0781 | Win/lose count 16.00/56.50 | Real score 9.00
Epoch 022/040 | Malus 55.50 | Loss 0.0950 | Win/lose count 16.50/61.50 | Real score 10.50
Epoch 023/040 | Malus 44.00 | Loss 0.0741 | Win/lose count 20.00/48.00 | Real score 16.00
Epoch 024/040 | Malus 40.00 | Loss 0.0796 | Win/lose count 22.50/49.00 | Real score 13.50
Epoch 025/040 | Malus 68.00 | Loss 0.0893 | Win/lose count 10.50/71.00 | Real score 7.50
Epoch 026/040 | Malus 42.50 | Loss 0.0800 | Win/lose count 22.50/47.50 | Real score 17.50
Epoch 027/040 | Malus 32.00 | Loss 0.0766 | Win/lose count 26.50/42.00 | Real score 16.50
Epoch 028/040 | Malus 65.00 | Loss 0.0776 | Win/lose count 12.50/71.00 | Real score 6.50
Epoch 029/040 | Malus 46.00 | Loss 0.0787 | Win/lose count 21.50/55.00 | Real score 12.50
Epoch 030/040 | Malus 38.50 | Loss 0.0723 | Win/lose count 24.00/49.50 | Real score 13.00
Epoch 031/040 | Malus 39.50 | Loss 0.0781 | Win/lose count 21.50/47.50 | Real score 13.50
Epoch 032/040 | Malus 65.00 | Loss 0.0774 | Win/lose count 16.00/72.00 | Real score 9.00
Epoch 033/040 | Malus 37.50 | Loss 0.0854 | Win/lose count 26.50/49.50 | Real score 14.50
Epoch 034/040 | Malus 41.00 | Loss 0.0840 | Win/lose count 18.50/46.00 | Real score 13.50
Epoch 035/040 | Malus 61.50 | Loss 0.0846 | Win/lose count 14.50/67.50 | Real score 8.50
Epoch 036/040 | Malus 55.50 | Loss 0.0773 | Win/lose count 16.50/58.50 | Real score 13.50
Epoch 037/040 | Malus 51.50 | Loss 0.0862 | Win/lose count 18.50/57.50 | Real score 12.50
Epoch 038/040 | Malus 46.50 | Loss 0.0795 | Win/lose count 23.00/57.50 | Real score 12.00
Epoch 039/040 | Malus 48.50 | Loss 0.0817 | Win/lose count 14.50/57.50 | Real score 5.50
Epoch 040/040 | Malus 45.50 | Loss 0.0748 | Win/lose count 18.00/58.50 | Real score 5.00
```

```
In [78]: # Evaluation
         test_explore(agent,env,5,prefix='cnn_test_explore') # epochs_test=5
         #HTML(display_videos('cnn_test_explore0.mp4'))
```

```
Win/lose count 19.00/63.00. Average real score 15.00. Malus 59.00.
Win/lose count 27.00/49.50. Average real score 18.50. Malus 44.50.
Win/lose count 12.50/73.50. Average real score 16.50. Malus 73.50.
Win/lose count 19.50/54.50. Average real score 16.50. Malus 51.50.
Win/lose count 7.00/86.50. Average real score 14.20. Malus 84.50.
Final score: 14.2
```

We see a clear progress compared to the previous models.

train_explore() is a modification of train() which incorporates the decay of epsilon through epsilon_decay. Epsilon cannot go below epsilon_min. The class EnvironmentExplore incorporates a state called malus_position which assigns a malus value to each cell explored. If the agent visits a cell already explored, the malus will be deduced from teh reward. The function test_explore() is more or less the same as test() but it returns an extra value which is the value of the malus points at each epoch. The "real" score is reward + malus because we do not want to count the malus in the final score since it is only used to train the agent and deter him from visitng cells already explored.

Epsilon is set at 0.9 so that the agent does a lot of exploring at the beginning but then, since epsilon_decay is 0.9, epsilon decreases 10% at each turn and so as the agent learns more he does less exploring since he has already done it at the beginning (with 40 epochs: 0.9ˆ41=0.013> epsilon_min).

BONUS question Use the expert DQN from the previous question to generate some winning games. Train a model that mimicks its behavior. Compare the performances.

Question 5 1.

Bellman : $Q^{\pi}(s,a) = E_{\pi}\left(\sum_{t=0}^{\infty} \gamma^t r_t(s_t, \pi(s_t), s_{t+1}) \mid s_1 = s, a_t = a \right)$

$$\Rightarrow Q^{\pi}(s_0,a_0) = E_{\pi}\left(r_0(s_0,a_0,s_1) + \sum_{t=1}^{\infty} \gamma^t r_t(s_t, \pi(s_t), s_{t+1}) \mid s_0, a_0 \right)$$

$$= E_{\pi}\left(r_0(s_0,a_0,s_1) + \gamma E_{\pi}\left[\sum_{t=1}^{\infty} \gamma^{t-1} r_t(s_t, \pi(s_t), s_{t+1}) \mid s_1, a_1 \right] \mid s_0, a_0 \right)$$

$$= E_{\pi}\left(r_0(s_0,a_0,s_1) + \gamma Q^{\pi}(s_1,a_1) \right)$$

2. $$V^{\pi}(s_0) = E\left(\sum_{t=0}^{\infty} \gamma^t r_t(s_t, \pi(s_t), s_{t+1}) \mid s_0 \right)$$

$$V^*(s_0) = \max E\left(\sum_{t=0}^{\infty} \gamma^t r_t(s_t, \pi(s_t), s_{t+1}) \mid s_0 \right)$$

$$\Rightarrow V^*(s_0) = \max_{\pi, a_0} E\left(r_0(s_0,a_0,s_1) + \sum_{t=1}^{\infty} \gamma^t r_t(s_t, \pi(s_t), s_{t+1}) \mid s_0 \right)$$

$$= \max_{a_0} E_{s_1}\left(r_0(s_0,a_0,s_1) + \gamma \max_{\pi} E_{s_2 s_3 s_4 \ldots}\left[\sum_{t=1}^{\infty} \gamma^{t-1} r_t(s_t, \pi(s_{t+1}), s_{t+1} \mid s_0 \right] \right)$$

$$= \max_{a_0} E_{s_1}\left(r_0(s_0,a_0,s_1) + \gamma V^*(s_1) \right)$$

$$= V^*(s) = \max E_{s'}\left[r(s,a,s') + \gamma V^*(s') \mid s \right]$$

We know, that : $V^*(s) = \max_{a'} Q^*(s,a')$

$$\Rightarrow V^*(s) = \max_{a} E_{s'}\left(r(s,a,s') + \gamma \max_{a'} Q^*(s',a') \mid s \right)$$

$$\Rightarrow Q^*(s,a) = E_{s'}\left[r(s,a,s') + \gamma \max_{a'} Q^*(s',a') \mid s,a \right]$$

$$Q^*(s,a) = E_{s' \sim \pi^*(\cdot \mid s,a)}\left[r(s,a) + \gamma \max_{a'} Q^*(s',a') \right]$$

3. MSE formula:

$$\mathcal{L}(\theta) = [\, Q^*(s,a,\theta) - Q(s,a\theta)\,]^2$$

plug in $Q^*(s,a,\theta)$:

$$\Rightarrow \quad \mathcal{L}(\theta) = E_{s,a}\left[\left(E_{s'}\left[r(s,a,s') + \gamma \max Q(s',a,\theta)\right) - Q(s,a,\theta)\right]^2\right]$$

target $\qquad$ $f(x)$