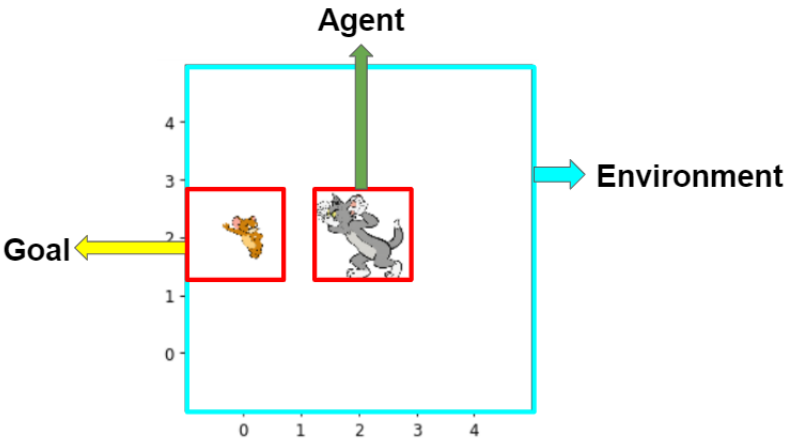
CSE474/574: Introduction to Machine Learning (Fall 2018)

Instructor: Sargur N. Srihari, Alina Vereshchaka, Nathan Margaglio

Project 4: Supporting material

Appendix 1 - Environment Description

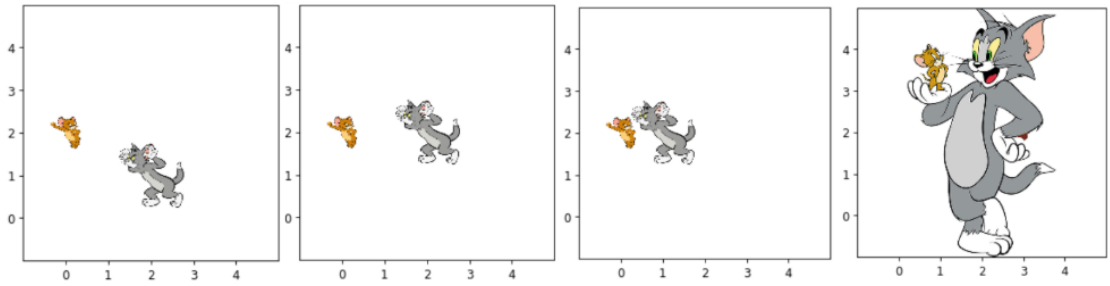
The environment is designed as a grid-world 5x5:



- States - 25 possible states (0, 0), (0, 1), (0, 2), ... ,(4, 3), (4, 4)
- Actions - *left, right, up, down*
- The goal (yellow square) and the agent (green square) are dynamically changing the initial position on every reset.

Goal

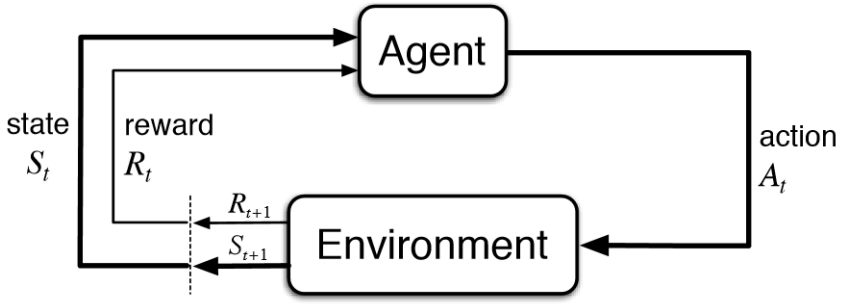
Our main goal is to let our agent learn the shortest path to the goal. In the environment the agent controls a green square, and the goal is to navigate to the yellow square (reward +1), using the shortest path. At the start of each episode all squares are randomly placed within a 5x5 grid-world. The agent has 100 steps to achieve as large a reward as possible. They have the same position over each reset, thus the agent needs to learn a fixed optimal path.



Appendix 2 - Reinfocement Learning Overview

Reinforcement learning is a direction in Machine Learning where an agent learn how to behave in a environment by performing actions and seeing the results. In reinforecement learning, an agent learns from trial-and-error feedback rewards from its environment, and results in a policy that maps states to actions to maximize the long-term total reward as a delayed supervision signal. Reinforcement learning combining with the neural networks has made great progress recently, including playing Atari games and beating world champions at the game of Go. It is also widely used in robotics.

Markov Decision Process



Basic reinforcement is modeled as a Markov decision process a 5-tuple (S, A, P_a, R_a, γ), where

- S is a finite set of states
- A is a finite set of actions
- $P_a(s, s') = \Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$ is the probability that action a in state s at time t will lead to state s' at time $t + 1$
- $R_a(s, s')$ is the immediate reward (or expected immediate reward) received after transitioning from state s to state s' due to action a
- $\gamma \in [0, 1)$ is the discount factor, which represents the difference in importance between future rewards and present rewards

In application, we typically have an *environment*, which handles state and reward, and an *agent*, which decides which action to take given a particular state. An environment starts with some initial state s_0, which is passed to the agent. The agent passes an action a_0, based on the state s_0, back to the environment. The environment reacts to the action, then passes the next state, s_1, along with the resulting reward for taking the action, r_0, back to the agent. This process continues until we reach a *terminal* state, indicating the end of an *episode*, at which point the process may start over again.

In reinforcement learning, we attempt to choose actions which yields the best results given some predefined criteria. This involves learning a mapping from state to action which attempts to maximize discounted accumulative reward. In deep reinforcement learning, a neural network is used approximate this mapping. Currently, there are two frequently used approaches to doing this: off-policy learning and on-policy learning.

Discounting factor (γ)

The discounting factor ($\gamma \in [0, 1]$) penalize the rewards in the future. Reward at time k worth only γ^{k-1} Motivation:

- The future rewards may have higher uncertainty (stock market)

- The future rewards do not provide immediate benefits (as human beings, we might prefer to have fun today rather than 5 years later)
- Discounting provides mathematical convenience (we donâĂŹt need to track future steps infinitely to compute return)
- It is sometimes possible to use undiscounted Markov reward processes (e.g. $\gamma = 1$), if all sequences terminate.

Main Definitions

Deterministic policy (π) is a function that maps states to actions:

$$\pi(s) = a \tag{1}$$

Return (G_t):

$$G_t = R_{t+1} + \gamma R_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2}$$

Value function ($V_\pi(s)$):

$$V_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s] \tag{3}$$

$$= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s\right], \text{for all } s \in S \tag{4}$$

Action-value function ($Q_\pi(s, a)$) - how good to take an action at a particular state:

$$Q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \tag{5}$$

$$= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a\right] \tag{6}$$

Objective is to find such a policy, that returns max Q-value.

$$\pi^*(s) = argmax_a Q(s, a) \tag{7}$$

Appendix 3 - Deep Q-Learning

Deep Q-Learning Algorithm (modified from the original)

Experience Replay

Experience replay will help us to handle three main things:

- Avoid forgetting previous experiences
- Reduce correlations between experiences
- Increases learning speed with mini-batches

The main idea behind the experience replay is that by storing an agentâĂŹs experiences, and then randomly drawing batches of them to train the network, we can more robustly learn to perform well in the task. By keeping the experiences we draw random, we prevent the network from only learning about what it is immediately doing in the environment, and allow it to learn from a more varied array of past experiences. Each of these experiences are stored as a tuple of *<state,action,reward,next state>*. The Experience Replay buffer stores a fixed number of recent memories (memory capacity), and as new ones come in, old ones are removed. When the time comes to train, we simply draw a uniform batch of random memories from the buffer, and train our network with them.

Initialize replay memory to capacity N

Initialize the environment (reset)

For episode $= 1, M$ **do** (Begin a loop of interactions between the agent and environment)

Initialize the first state $s_0 = s$

For $t = 1, T$ **do**

With probability ϵ select a random action a_t, otherwise select $a_t = argmax_a Q(s, a; \Theta)$

Execute action a_t and observe reward r_t and next state s_{t+1}

A tuple $< s_t, a_t, r_t, s_{t+1} >$ has to be stored in memory

Sample random minibatch of observations (s_t, a_t, r_t, s_{t+1}) from memory

Calculate Q-value

$$Q_t = \begin{cases} r_t, & \text{if episode terminates at step } t+1 \\ r_t + \gamma max_a Q(s_t, a_t; \Theta), & \text{otherwise} \end{cases}$$

Train a neural network on a sampled batched from the memory

End **For**

End **For**

Appendix 4 - Exploration vs Exploitation

Our agent will randomly select its action at first by a certain percentage, called "exploration rate" or "epsilon". This is because at first, it is better for the agent to try all kinds of things before it starts to see the patterns. When it is not deciding the action randomly, the agent will predict the reward value based on the current state and pick the action that will give the highest reward.

Exponential-decay formula for epsilon:

$$\epsilon = \epsilon_{min} + (\epsilon_{max} - \epsilon_{min}) * e^{-\lambda |S|}, \tag{8}$$

where

$\epsilon_{min}, \epsilon_{max} \in [0, 1]$

λ - hyperparameter for epsilon

$|S|$ - total number of steps

Appendix 5 - Neural Network

Neural network structure for Task 1

- The model's structure is: LINEAR → RELU → LINEAR → RELU → LINEAR.
- Activation function for the first and second hidden layers is 'relu'
- Activation function for the output layer is 'linear' (that will return real values)
- Input dimensions for the first hidden layer equals to the size of your observation space ($state\_size$)
- Number of hidden nodes is 128 for both hidden layers
- Number of the output should be the same as the size of the action space ($action\_size$)

Appendix 6 - Useful links

1. Jupyter Notebook:
    - Installation
    - Quick guide (Medium article)
    - Google Cobal
2. Sutton, R.S. and Barto, A.G.. Reinforcement learning: An introduction. MIT press, 2018
3. NIPS template (can be used for reports)