Project 2: Reinforcement Learning

T-504: Introduction to Machine Learning

Fall, 2022

1 Introduction

The goal of this project is to apply your theoretic knowledge about reinforcement learning algorithms to learn a policy for an agent playing the video game Flappy Bird. (A clone that is playable online can be found at `http://flappybird.io/`).

This project can be done in groups of up to 3 students. (The groups do not have to be the same as for the labs.) The intended workload for this project is about 15-20 hours per student (not counting computation time, that is, assuming you set up the program, let it run and do something else while you wait for the results). Start working on this project in time. There will be some waiting for results of your program and probably several iterations of tests until you are happy with the results.

2 Problem Description

Your task is to use Reinforcement Learning to find a good policy for Flappy Bird. The goal of the game is to survive for as long as possible (pass through as many gaps in the pipes as possible) before dying.

Your goal is two-fold:

1. Learn a policy that is as good as possible, and
2. learn the policy as quickly as possible (with few simulations of the game).

Obviously, there is some trade-off between these two goals.

The quality of a policy is the expected number of points achieved, with one point rewarded for passing through each pipe. For example, this could be measured by playing Flappy Bird with a policy n times and taking the average of the achieved points. The time for learning a policy is measured in the number of frames of the game that have been simulated.

During training, it may make sense to set up a different reward structure to incentivize the agent to avoid or seek out certain situations.

To solve the problem, you need to consider the following points:

The Nature of the Environment Is it discrete/finite or continuous? Is is stochastic or deterministic? Is it episodic or sequential? What constitutes a state of the environment? Is it a Markov process? (Or in other words: What information should be in the state such that it can be considered Markovian?) What knowledge do you have about the dynamics of the environment (state transition probabilities)?

The Algorithm(s) to Use Depending on the nature of the environment and your knowledge about it, you need to decide which algorithms can potentially be used to learn a policy. Candidates are Dynamic Programming, Monte-Carlo Methods or the various temporal difference learning algorithms (e.g., Q-Learning or Sarsa).

Also, the encoding of the policy or value function that you learn depends on the nature of the environment. Can you use a tabular approach? (Maybe you need to discretize the environment for that.) Is it better to use function approximation? If so, what kind?

The Parameters of the Algorithms Depending on which algorithm(s) you decide to use, you need to decide on the parameters. E.g., when to stop in dynamic programming, what learning rate to use for Q-Learning, how to balance exploration and exploitation, etc.

How to Measure Progress / Success If you are trying out different algorithms or parameters you need an objective way to measure how good your policy is.

3 Environment

For the project, we will use a clone of the game implemented in Python in the PyGame Learning Environment (PLE), which offers a convenient way of simulating the game fast than in real time and gives access to some high level features (such as position of the bird and obstacles as opposed to the pixels on the screen). The PyGame Learning Environment (PLE) requires the PyGame library, as well as numpy and scipy. You should be able to install PyGame with Anaconda or using `pip install pygame` on Linux. A copy of PLE comes with the code for the project.

Use the attached python code (`flappy_agent.py`) to get started.

Your task is to implement different agents all with the same interface as lined out in the FlappyAgent class.

An agent has two policies, one for training and one for evaluation (after training). The policies are implemented in the training_policy and policy function, which get a game state as input and need to return either 0 (flap the wings) or 1 (do nothing).

The game state, is a python dictionary containing the locations of the bird and the next two gaps in the pipes as well as current velocity of the bird (speed of falling down). The screen is 288 pixels wide and 512 pixels high and all coordinates and distances are measured in pixels. Velocity is between -8 and 10 (at most rising 8 pixels or falling 10 pixels per frame). All values are integers.

Alternatively to these high-level features of the game state you could work with the raw screen pixels by using env.getScreenRGB() instead of env.game.getGameState() in the run_game function. However, this is a much more difficult task and would probably greatly increase the training time.

4 Tasks

For each of the questions asking you to implement a specific algorithm, make a new subclass of FlappyAgent, so we can see and run the code that you used. The handin should consist of a report (pdf) and your code. In the report, make sure to note which part of the code belongs to the respective question.

1. **Characterize the environment**. Based on the characteristics of the environment, discuss which algorithms could potentially be used to solve it and what challenges you would expect in the process.

2. Implement and run **Q-Learning** to learn a policy for the task. Use a tabular approach for $Q(s, a)$, where the state consists of

   - the current y-position of the bird (`player_y` component of the game state)
   - the top y position of the next gap (`next_pipe_top_y`)
   - the horizontal distance between bird and next pipe (`next_pipe_dist_to_player`)
   - the current vertical velocity of the bird (`player_vel`)

   . To decrease the size of the table, split the space of these attributes (except the velocity) evenly into 15 intervals each, resulting in at most 64125 different states to consider (not all of those are reachable).

   Use an ϵ-greedy policy with $\epsilon = 0.1$ and a learning rate of 0.1.

   As reward structure for training, use the default values returned by FlappyAgent.reward_values() (1 point for passing a pipe and -5 points for crashing) without discounting of future rewards.

   Train a policy and evaluate it. Plot a curve showing the progress of training, i.e., the quality of the policy over the number of game steps (frames) trained on.

   Interpret the learning curve. What can be said about the performance of the algorithm with these parameters and this state representation on the given task?

3. Consider a function approximation approach for representing $Q(s,a)$. That is, use a parameterized function $Q(s,a,\vec{\theta}) \approx Q(s,a)$ to approximate $Q(s,a)$. Updating Q-values in the algorithms above now turns into updating $\vec{\theta}$ using gradient decent on the error. There are many possible choices for functions $Q(s,a,\vec{\theta})$, e.g., linear functions, polynomial functions or neural networks.

   As a starting point, setup a neural network (e.g., use sklearn's MLPRegressor or some other library) with:

   - the inputs being the normalized features of the state (normalized to the interval $[-1,1]$),
   - two fully-connected hidden layers with 100 neurons in the first layer and 10 in the second layer
   - two output neurons, one for each action, with the interpretation that the i-th output gives you the value $Q(s,i,\vec{\theta})$ when s is the input into the network and $\vec{\theta}$ are the weights of the network.
   - a sigmoid activation function for all hidden layers
   - an initial learning rate of 0.01

   Implement a version of Q-Learning or Sarsa using that function approximation approach. That means, instead of updating $Q(s,a)$ directly, you need to run `partial_fit` on the neural network to update the weights of the network.

   Learning from a single sample at a time is not very efficient. Therefore, you should keep a history (also called "replay buffer") of the last 1000 samples (observed state transitions) and in every step train on a batch of 100 randomly selected samples from the history. This technique is called experience replay. Both the size of the replay buffer and the batch-size are parameters of your algorithm that you might want to tune.

   An observation (s,a,r,s') is turned into a training example for your network by:

   - setting the input vector x to the normalized features of the state s
   - setting the output vector $y = (q_{flap}, q_{noflap})$ where $q_a = r + \gamma * \max_{a'} Q(s',a',\vec{\theta})$ and $q_b = Q(s,b,\vec{\theta})$. That is, the target Q-value of the action that was done in state s should computed as in Q-Learning from the estimated value of the greedy-best action in the successor state. The target value of the other action (that was not done in state s) is simply the old predicted value of that action.

   Run the algorithm and observe the results of training. Interpret the results and try to explain them. That is, it might work better or worse than the tabular approach or not at all. Why do think that is?

4. Try to **make the best agent**!

   This can be achieved in different ways, for example:

   - tuning the parameters of the learning algorithm (α, ϵ, ...)
   - tuning the reward structure used for training (rewards for different situations, discounting)
   - changing the features that represent the states or the discretization of the state space
   - using a different representation for $Q(s,a,\vec{\theta})$, if you use function approximation (linear model, different structure of a neural network, ...)
   - using different methods for solving the exploration/exploitation dilemma (e.g., UCB instead of epsilon-greedy)
   - using a different RL method (e.g., Monte-Carlo instead of Q-Learning or n-step TD-Learning instead of 1-step)

   Experiment with different options to try to find the method resulting in the best policy in a reasonable amount of time. Report on the results of your experiments varying at least 3 different parameters (parameters of the algorithm, state representation, ...) and interpret these results. What influence do the different parameters have on speed of learning, quality of the resulting policy, stability of the algorithm? Based on these experiments, what do you think is the best way to learn a policy for this environment? Remember that there are two goals: learning a policy that is as good as possible and learning a reasonable policy as quickly as possible. Different methods might work differently well for these two objectives.

5. Up to 10 **bonus points** (with a 100 points total for the assignment) will be given for (up to 5 points each):

   - the best policies learned from 1'000'000 frames or 1 hour training time, whichever is exceeded first
   - the fewest frames needed to train a policy that achieves 50 points consistently (achieving ≥ 50 points on at least 90% of the tested episodes)

   Note, that in either case learning must not be slower than real-time, i.e., generating sequences and learning from them must be possible to do with ≈ 30 frames per second on average on a single core (or alternatively, learning from n*30 frames should not take longer than n seconds on average).

   To get the bonus points, state in your report which agent class you want us to test for which of the two objectives and shortly describe how each of them learns (which algorithm, parameters, state representation, ...).

5 Grading

Your grade will consist of components for

- the correct implementation of the algorithms and answers to the questions
- the setup of the experiments and the report on the results
- how good your resulting agent is in learning a policy