Lab 7: Deep Reinforcement Learning

By: Erik Gärtner

Introduction

In this lab you are going to implement a *deep* reinforcement learning agent. Unlike in supervised learning the algorithm does not learn from examples but rather from interacting with the problem, i.e. trial and error.

Environments

Cart Pole

Your agent is going to solve the `CartPole-v1` environment in OpenAI's Gym. The goal of the environment is to balance a pole on top of a cart that your agent controls. The environment has four floats as the state: position, velocity, angle, and angular velocity. Your agent can either push the cart right or left.

From the OpenAI's webpage (https://gym.openai.com/envs/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.

Lunar Lander

The second environment is the `LunarLander-v2`. Read more about it here https://gym.openai.com/envs/LunarLander-v2/.

Agent

You are going to implement the **Vanilla Policy Gradient** (also known as REINFORCE or Monte-Carlo Policy Gradient) algorithm for the agent. It's one of the most simple algorithms in deep reinforcement learning. Gradient Policy agents are at the heart of many of the latest breakthroughs such as OpenAI's Dota 2 agents.

The basic idea of the algorithm is to model the policy (the mapping from a state to an action) as a neural network. That is, the neural network takes the state as input and outputs a *probability distribution* over the possible actions. In this assignment we are going to use the *SoftMax* activation function to model the

probability for each action. We then randomly sample an action based on the probability given by the SoftMax.

Vanilla Policy Gradient is an *on-policy* algorithm meaning we always follow the policy. This opposed to for example *Q-learning* where the agent would take the best action at each time step except with some small probability when it would instead take a random action to explore. In our algorithm, during training, the agent explores all the time since it samples from the policy distribution and does *not only* take the best (i.e. most probable action).

The skeleton code provided is intended to be as simple as possible. Therefore some performance improvements that complicates the code has been omitted.

You will need to construct and optimize on the loss directly, with the loss given as:

$$L = -\sum_i A_i \log(\pi_\theta(a_i|s_i))$$

The loss can be explained as: the logarithm of the probability of the selected action, $\pi(a_i|s_i)$, where a_i is the action taken when in state s_i during interaction times the (discounted) reward A_i, and where i is the index in the training batch. Note that $\pi(a_i|s_i)$ is the policy, i.e. the neural network!

For more on Vanilla Policy Gradient you can read:

- Andrew Karpathy's Blog Post that explains the theory
- Chapter 16 in Aurélien Géron: Hands-On Machine Learning with Scikit-Learn and TensorFlow, Concepts, Tools, and Techniques to Build Intelligent Systems. O'Reilly Media, 2017, ISBN: 9781491962299.

Tools

- **Python 3.6**
- **OpenAI's Gym** is a framework for learning environments. You will not need to design your own environment. We will use the built-in `CartPole-v1` and `LunarLander-v2` environments.
- **Tensorflow** is useful for the neural network in the reinforcement learning agent. Using more abstract frameworks such as Keras is possible but *more* complicated.
- **Numpy** and **Matplotlib** will be useful as well for smaller computations and plotting.

These tools will already be installed and available on the lab computers. See `requirements.txt` for all packages required by the lab if you want to install it on your computer.

Goals

- Implement a Vanilla Policy Gradient agent that **solves** `CartPole-v1`.
- Try varying the learning rate, gamma and network design. What happens?
- **Try** the agent on the harder problem of `LunarLander-v2`. How well does it solve this problem?
- Try varying the learning rate, gamma and network design? What happens?

Instructions

1. Read module `rlagent/main.py` and understand the provided training loop. The module is complete and fully functioning. You may however need to modify it to the agent's hyperparameters, or to change the environment. To run the training execute:

   ```
   python -m rlagent.main
   ```

2. Have a look at the agent module `rlagent/agent.py`. This module is only a skeleton that randomly selects an action. It is up to you implement the Vanilla Policy Gradient algorithm in here. You will not need to modify the method signatures to solve the task.

   1. Implement `__init__()` that initializes the neural network for the policy.
   2. Implement `take_action()` that samples the action from the policy network. Run the training and see that it works and acts *randomly*.
   3. Implement `record_action()` that just saves a taken action and its reward.
   4. Implement `discount_rewards_and_normalize()` that discounts and normalizes (rewards should have zero mean and unit variance) the rewards for an episode.
   5. Implement `train_agent()` that takes the experiences (states, actions, and rewards) collected in `record_action()` and sends them through the network to compute and optimize the loss.

3. When you have a working implementation your agent should solve `CartPole-v1` in ~500 episodes.

4. Experiment with applying the agent to the `LunarLander-v2` environment as described in the Goal section.