Reinforcement Learning Lab

Lesson 9: REINFORCE

Davide Corsi and Alberto Castellini

University of Verona
email: davide.corsi@univr.it

Academic Year 2022-23

UNIVERSITÀ
di VERONA
Dipartimento
di INFORMATICA

Environment Setup

The first step for the setup of the laboratory environment is to update the repository and load the miniconda environment.

- Update the repository of the lab:

```
cd RL-Lab
git stash
git pull
git stash pop
```

- Activate the *miniconda* environment:

```
conda activate rl-lab
```

Reference

Today's lesson is based on the official documentation from the Open AI Spinning Up page:
https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html

Today Assignment

In today's lesson, we will implement the REINFORCE (*or naïve policy gradient*) algorithm to solve the CartPole problem. In particular, the file to complete is:

```
RL—Lab/lessons/lesson_9_code.py
```

Inside the file, two python functions are partially implemented. The objective of this lesson is to complete it.

- **def REINFORCE_naive()**
- **def REINFORCE_rw2go()**

Expected results can be found in:

```
RL—Lab/results/lesson_9_results.txt
```

General Structure and DRL Loop

1. In today's lesson, you should use some functions you already implemented in the past labs, in particular *training_loop* and *createDNN*.
2. Remember that REINFORCE is an on-policy algorithm; this means that you can train the policy only on data collected with the same policy! For this reason, you should always clear the memory buffer after each update of the network (that's why, in contrast to DQN, we don't need to set a limit to the number of elements in the buffer).
3. A crucial trick to improve the performance of this algorithm consists in collecting multiple trajectories and performing the network update by averaging among them.

```
# 1) Remember to always clean the buffer after a training step!
# 2) Perform the training only after 'frequency' episodes
if ep % frequency == 0:
    updateRule( neural_net, memory_buffer, optimizer )
    memory_buffer = []
```

REINFORCE

In REINFORCE, the update rule consists in the naïve implementation of the policy-gradient theorem. The objective function to implement is:

$$J_\pi = (\sum_{t=0}^{T} log\pi_\theta(a_t|s_t)) \cdot (\sum_{t=0}^{T} R_t) \tag{1}$$

Reward To Go is a simple while impactful improvement for REINFORCE. The intuition is to always provide information only on future actions and not on the past, with the idea of preserving the causality principle. The update rule should be changed as follows:

$$J_\pi = \sum_{t=0}^{T} (log\pi_\theta(a_t|s_t) \cdot \sum_{t}^{T} R_t) \tag{2}$$

Shaping

These operations require different reshaping: remember to always print the shape of the tensors and arrays before and after all the algebraic operations.

Code Snippets

Following is a python-like code snippet for the update rule:

```
# When working with tensor, remember to use the 'tf functions'
# (e.g., math.log, math.reduce_sum, math.reduce_mean) to avoid
# losing gradient informations
log_probs = tf.math.log(probabilities)
log_prob_sum = tf.math.reduce_sum(tf.math.log(probabilities))
objectives = log_prob_sum * sum(rewards)
```

By default, Tensorflow performs the gradient descent on the provided objective function, but a policy-gradient algorithm is a maximization problem! So you should always remember to optimize the negation of the objective:

```
# Always remember the negation of the objective
# to perform the maximization of the function!
objective = -tf.math.reduce_mean(objectives)
grads = tape.gradient(objective, neural_net.trainable_variables )
optimizer.apply_gradients( zip(grads, neural_net.trainable_variables) )
```

Expected Results

Multiple Trajectories

These results are obtained by exploiting the *multi-trajectories* trick. It is possible to obtain acceptable results also with a single-trajectory update, but the suggested solution exploits this improvement.

Memory Buffer

The algorithm is stochastic, but with respect to DQN it should be more stable. For this reason, setting a random seed is not necessary.

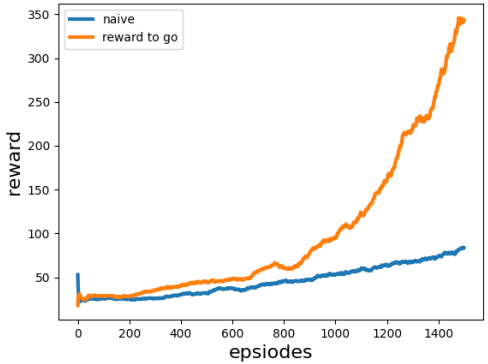

Figure: Note that obtaining this result requires time. You can stop the training after fewer iterations if you observe a growth in the reward.

Final Suggestions

1. Remember that in REINFORCE (and in general in all the gradient-based methods), the output of the neural network is a probability distribution. You should (i) use a *softmax* activation function and (ii) sample the action to perform from the output layer. Following is a code snippet:

```
# Add the output layer
model.add(Dense(nOutputs, activation="softmax"))
# Select the action to perform
distribution = neural_net( state.reshape(-1, 4) ).numpy()[0]
action = np.random.choice(2, p=distribution)
```

2. To average between multiple trajectories, you can create an array of the loss function and average them before the gradient-ascent procedure:

```
# Add the objective for the current trajectory
objectives.append( trajectory_probability * trajectory_reward )
# Compute the (negation) of the average objectives before the gradient
objective = -tf.math.reduce_mean(objectives)
grads = tape.gradient(objective, neural_net.trainable_variables )
```