Reinforcement Learning Lab

Lesson 9: REINFORCE

Gabriele Roncolato and Alberto Castellini

University of Verona
email: gabriele.roncolato@univr.it

Academic Year 2024-25

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
```

Today Assignment

In today's lesson, we will implement the REINFORCE algorithm to solve the CartPole problem. In particular, the file to complete is:

```
RL—Lab/lessons/lesson_8_code.py
```

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

- **def training_loop()**
- **def REINFORCE()**

Your task is to fill in the code where there are #TODO, or "pass" words. Expected results can be found in:

```
RL—Lab/results/lesson_8_results.png
```

REINFORCE

This is the pseudo-code of REINFORCE ("Reinforcement Learning an Introduction" Sutton and Barto)

REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ $\qquad (G_t)$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$

In REINFORCE, the update rule of a policy π_θ is based on the results of the *policy-gradient* theorem. The objective function $G(t)$ uses information only on future actions (not on the past). The update rule to implement is the following:

$$G(t) \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$$

$$\theta \leftarrow \theta + \alpha \gamma^t G(t) \nabla \log \pi(A_t | S_t, \theta)$$

In practice, we first compute the reward-to-go $G(t)$, then use it in the update rule.

Reward-to-go computation

```
for ep in range(len(memory_buffer)):
    # Extraction of the information from the buffer (for the considered episode)
    states = get_memory_buffer_states()
    actions = get_memory_buffer_actions()
    rewards = get_memory_buffer_rewards()

G = [] # list of rewards-to-go
g = 0 # current reward to be discounted

# calculate the return G reversely
for r in reversed(rewards):
    # update g at the timestep t value as the immediate reward + the future discoun
    g = gamma * g + r

    # insert g as the first element of the list G
    G.insert(0, g)
```

The "reward-to-go" approach calculates the return for each step by considering only the rewards that occur after that step until the end of the episode.

Important note

Remember that in REINFORCE (and in general in all the gradient-based methods), the neural network's output 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 softmax layer (keras)
model.add(Dense(nOutputs, activation="softmax"))
# Add the softmax operation output layer (PyTorch)
def forward(self, x):
    ...
    ...
    ...
    x = self.output(x)
    return torch.nn.functional.softmax(x, dim=1)
# Select the action to perform
distribution = neural_net( state.reshape(-1, 4) ).numpy()[0]
action = np.random.choice(env.action_space.n, p=distribution)
```

Policy update

By default, PyTorch-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!
a_prob = neural_net(state)
policy_loss = -gamma**t * G(t) * log(a_prob[action])
# perform backpropagation of the loss
...
```

Maximizing the likelihood of taking good actions (which corresponds to minimizing the negative log probability) is equivalent to maximizing the expected return, which precisely corresponds to our goal.

REINFORCE with Baseline

Let us recall the pseudo-code of REINFORCE with Baseline ("Reinforcement Learning an Introduction" Sutton and Barto)

REINFORCE with Baseline (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Algorithm parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^d$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ $\qquad (G_t)$
    $\delta \leftarrow G - \hat{v}(S_t,\mathbf{w})$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S_t,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \gamma^t \delta \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$

In REINFORCE with Baseline, the update rule of a policy π_θ is also based on reward-to-go computation, but in addition, we use an estimator, i.e., a DNN $\pi_{\theta'}$ for the state value $\hat{v}(S_t, \theta')$ to compute:

$$
\begin{aligned}
G(t) &\leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \\
\delta &\leftarrow G - \hat{v}(S_t, \theta') \\
\theta' &\leftarrow \theta' + \beta \delta \nabla \hat{v}(S_t, \theta') \\
\theta &\leftarrow \theta + \alpha \gamma^t \delta \nabla \log \pi(A_t | S_t, \theta)
\end{aligned}
$$

Why? **Variance Reduction**: When estimating the policy gradient using REINFORCE, the estimates can have high variance, meaning they fluctuate widely between different episodes or samples. This high variance can slow learning and make the optimization process less stable. By subtracting a baseline value (often the state value estimate) from the returns, we reduce the variance of the policy gradient estimates.

Pseudo code REINFORCE with baseline

Here we provide the snippet of the code you have to implement for a single step of the trajectory stored in one episode:

```
for t in range(len(rewards)):
    state = states[t]
    a = actions[t]
    g = G[t]
    v_s = value_net(state)

    # Update policy
    a_prob = neural_net(state)
    policy_loss = -gamma**t * (g - v_s) * log(a_prob[a])
    # backpropagation of the policy loss
    ...


    # Update value function
    value_loss = MSE(v_s, g)
    # backpropagation of the value function loss
    ...
```

Expected Results

Updates

These results are obtained by performing an update after each episode (i.e., trajectory). It is also possible to perform *multi-trajectories* updates, computing an average performance between trajectories to increase the stability of the learning process.

Baseline improving

As we can notice from the plot, using *baseline*, we obtained increased stability during the learning process.

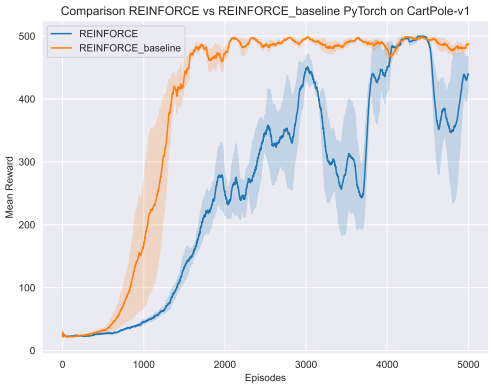

Figure: Mean and Standard Error using 3 random seeds for PyTorch REINFORCE (with baseline) implementation.

Time required

These results are obtained by performing an update after each episode (i.e., trajectory). Obtaining the results presented in the previous slides (5k episodes) could require time. You can stop the training after fewer episodes if you observe a growth in the reward.

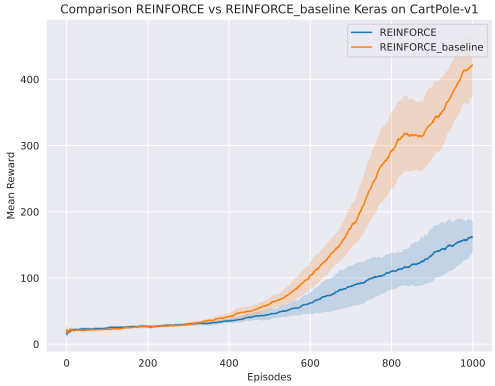

Figure: Mean and Standard Error using 3 random seeds for Keras REINFORCE (with baseline) implementation.