10-703 Deep Reinforcement Learning and Control
Assignment 3
Fall 2018

October 14, 2018

Due October 28, 2018, 11:59pm (EST)

Instructions

You may work in teams of **2** on this assignment. Only one person should submit the writeup on Gradescope. Additionally, the same person who submitted the writeup to Gradescope must upload the code to Autolab. Make sure you mark your partner as a collaborator on Gradescope (you do not need to do this in Autolab) and that both names are listed in the writeup. Writeups should be typeset in LaTeXand submitted as PDF. All code, including auxiliary scripts used for testing, should be submitted with a README.

Please limit your writeup to 8 pages or less (excluding the provided instructions).

We've provided some code templates (using Keras) that you can use if you wish. Abiding to the function signatures defined in these templates is not mandatory; you can write your code from scratch if you wish. You can use any deep learning package of your choice (e.g., Keras, Tensorflow, PyTorch). However, if you choose not to use Keras, then you'll need to figure out how to load the policy network architecture (`LunarLander-v2-config.json`).

You should not need the cluster or a GPU for this assignment. The models are small enough that you can train on a laptop CPU.

It is expected that all of the work you submit is your own. Submitting a classmate's code or code which copied from online and claiming it is your own is not allowed. Anyone who does this will be violating University policy, and risks failure of the assignment, course and possibly disciplinary action taken by the university.

Introduction

In this assignment, you will implement different RL algorithms and evaluate them on the LunarLander-v2 environment. This environment is considered solved if the agent can achieve an average score of at least 200.

Installation instructions (Linux)

We've provided Python packages that you may need in `requirements.txt`. To install these packages using pip and virtualenv, run the following commands:

```
apt-get install swig
virtualenv env
source env/bin/activate
pip install -U -r requirements.txt
```

If your installation is successful, then you should be able to run the provided template code:

```
python reinforce.py
python a2c.py
```

Note: You will need to install `swig` and `box2d` in order to install `gym[box2d]`, which contains the `LunarLander-v2` environment. You can install `box2d` by running

```
pip install git+https://github.com/pybox2d/pybox2d
```

If you simply do `pip install box2d`, you may get an error because the pip package for `box2d` depends on an older version of `swig`.[1] For additional installation instructions, see https://github.com/openai/gym.

[1] https://github.com/openai/gym/issues/100

Problem 1: REINFORCE (30 pts)

In this section, you will implement episodic REINFORCE, a policy-gradient learning algorithm. Please write your code in `reinforce.py`; the template code provided inside is there to give you an idea on how you can structure your code, but is not mandatory to use.

Policy gradient methods directly optimize the policy $\pi(A \mid S, \theta)$, which is parameterized by θ. The REINFORCE algorithm proceeds as follows. We keep running episodes. After each episode ends, for each time step t during that episode, we alter the parameter θ with the REINFORCE update. This update is proportional to the product of the return G_t experienced from time step t until the end of the episode and the gradient of $\ln \pi(A_t \mid S_t, \theta)$. See Fig. 1 for details.

Algorithm 1 REINFORCE

procedure REINFORCE
    Start with policy model π_θ
    repeat:
        Generate an episode $S_0, A_0, r_0, \ldots, S_{T-1}, A_{T-1}, r_{T-1}$ *following* $\pi_\theta(\cdot)$
        for t *from* $T-1$ *to* 0:
            $G_t = \sum_{k=t}^{T-1} \gamma^{k-t} r_k$
        $L(\theta) = \frac{1}{T} \sum_{t=0}^{T-1} G_t \log \pi_\theta(A_t|S_t)$
        Optimize π_θ *using* $\nabla L(\theta)$
end procedure

For the policy model $\pi(A \mid S, \theta)$, use the network config provided in `LunarLander-v2-config.json`. It already has a softmax output so you shouldn't have to modify the config. As shown in the template code, you can load the model by doing:

```
with open('LunarLander-v2-config.json', 'r') as f:
    model = keras.models.model_from_json(f.read())
```

You can choose which optimizer and hyperparameters to use, so long as they work for learning on `LunarLander-v2`. We recommend using Adam as the optimizer. It will automatically adjust the learning rate based on the statistics of the gradients it's observing. Think of it like a fancier SGD with momentum. Both Tensorflow and Keras provide versions of Adam.

Downscale the rewards by a factor of 1e-2 (i.e., divide by 100) when training (but not when plotting the learning curve). When you implement A2C in the next section, this will help with the optimization since the initial weights of the Critic are far away from being able to predict a large range such as $[-200, 200]$.

To train the policy model, you need to take the gradient of the log of the policy. This is simple to do with Tensorflow: take the output tensor of the Keras model, call `tf.log` on that tensor, and use `tf.gradients` to get the gradients of the network parameters with respect to this log. You will also have to scale by the returns from your sampled policy runs (i.e., scale by G).

Train your implementation on the `LunarLander-v2` environment until convergence[2], and answer the following questions:

1. Describe your implementation, including the optimizer and any hyperparameters you used (learning rate, γ, etc.).
2. Plot the learning curve: Every k episodes, freeze the current cloned policy and run 100 test episodes, recording the mean/std of the cumulative reward. Plot the mean cumulative reward μ on the y-axis with $\pm\sigma$ standard deviation as error-bars vs. the number of training episodes.

   Hint: You can use matplotlib's `plt.errorbar()` function. https://matplotlib.org/api/_as_gen/matplotlib.pyplot.errorbar.html

Problem 2: Advantage-Actor Critic (40 pts)

In this section, you will implement N-step Advantage Actor Critic (A2C). Please write your code in `a2c.py`; the template code provided inside is there to give you an idea on how you can structure your code, but is not mandatory to use.

Algorithm 2 N-step Advantage Actor-Critic

procedure N-STEP ADVANTAGE ACTOR-CRITIC
  Start with policy model π_θ *and value model* V_ω
  repeat:
    Generate an episode $S_0, A_0, r_0, \ldots, S_{T-1}, A_{T-1}, r_{T-1}$ *following* $\pi_\theta(\cdot)$
    for t *from* $T-1$ *to* 0:
      $V_{end} = 0$ if $(t+N \geq T)$ *else* $V_\omega(s_{t+N})$
      $R_t = \gamma^N V_{end} + \sum_{k=0}^{N-1} \gamma^k \left(r_{t+k} \text{ if } (t+k < T) \text{ else } 0\right)$
    $L(\theta) = \frac{1}{T}\sum_{t=0}^{T-1}(R_t - V_\omega(S_t)) \log \pi_\theta(A_t|S_t)$
    $L(\omega) = \frac{1}{T}\sum_{t=0}^{T-1}(R_t - V_\omega(S_t))^2$
    Optimize π_θ *using* $\nabla L(\theta)$
    Optimize V_ω *using* $\nabla L(\omega)$
end procedure

N-step A2C provides a balance between bootstraping using the value function and using the full Monte-Carlo return, using an N-step trace as the learning signal. See Algorithm 2 for details. N-step A2C includes both REINFORCE with baseline ($N = \infty$) and the 1-step A2C covered in lecture ($N = 1$) as special cases and is therefore a more general algorithm.

The Critic updates the state-value parameters ω, and the Actor updates the policy parameters θ in the direction suggested by the N-step trace.

As in Problem 2, use the network architecture for the policy model $\pi(A \mid S, \theta)$ provided in `LunarLander-v2-config.json`. Play around with the network architecture of the Critic's

[2] `LunarLander-v2` is considered solved if your implementation can attain an average score of at least 200.

state-value approximator to find one that works for `LunarLander-v2`. You can choose which optimizer and hyperparameters to use, so long as they work for learning on `LunarLander-v2`. Downscale the rewards by a factor of 1e-2 during training (but not when plotting the learning curves); this will help with the optimization since the initial weights of the Critic are far away from being able to predict a large range such as $[-200, 200]$.

Answer the following questions:

1. Describe your implementation, including the optimizer, the critic's network architecture, and any hyperparameters you used (learning rate, γ, etc.).
2. Train your implementation on the `LunarLander-v2` environment several times with N varying as [1, 20, 50, 100] (it's alright if the N=1 case is hard to get working). Plot the learning curves for each setting of N in the same fashion as Problem 2.
3. Discuss (in max 500 words) how A2C compares with REINFORCE and how A2C's performance varies with N. Which algorithm and N setting learns faster, and why do you think this is the case?

Guidelines on implementation

This homework requires a significant implementation effort. It is hard to read through the papers once and know immediately what you will need to be implement. We suggest you to think about the different components (e.g., Tensorflow or Keras model definition, model updater, model runner, ...) that you will need to implement for each of the different methods that we ask you about, and then read through the papers having these components in mind. By this we mean that you should try to divide and implement small components with well-defined functionalities rather than try to implement everything at once. Much of the code and experimental setup is shared between the different methods so identifying well-defined reusable components will save you trouble.

Please note, that while this assignment has a lot of pieces to implement, most of the algorithms you will use in your project will be using the same pieces. Feel free to reuse any code you write for your homeworks in your class projects.

This is a challenging assignment. **Please start early!**