10-703 Deep Reinforcement Learning and Control
Assignment 3 Report
Spring 2018

March 26, 2018

Due April 9, 2018, 11:59pm (EST)

Luyi Ma (luyim), Zhefan Zhu (zhefanz)

Problem 1: Imitation Learning (30 pts)

In this section, you will implement behavior cloning using supervised imitation learning from an expert policy, and test on the LunarLander-v2 environment. Please write your code in `imitation.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.

We have provided you with an expert, which you will use to generate training data that you can use with the Keras `fit` method. The expert model's network architecture is given in `LunarLander-v2-config.json` and the expert weights in `LunarLander-v2-weights.h5`. You can load the expert model using the following code snippet:

```
import keras

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

model_weights_path = 'LunarLander-v2-weights.h5f'
expert.load_weights(model_weights_path)
```

Tasks and questions:

1. Use the provided expert model to generate training datasets consisting of 1, 10, 50, and 100 expert episodes. You will need to collect states and a one-hot encoding of the expert's selected actions.
2. Use each of the datasets to train a cloned behavior using supervised learning. For the cloned model, use the same network architecture provided in `LunarLander-v2-config.json`. When cloning the behavior, we recommend using the Keras `fit` method. You should compile the model with cross-entropy as the loss function, Adam as the optimizer, and include 'accuracy' as a metric.

   For each cloned model, record the final training accuracy after training for at least 50 epochs. Training accuracy is the fraction of training datapoints where the cloned policy successfully replicates the expert policy. Make sure you include any hyperparameters in your report.

   Results:

   For each models trained over 4 different training datasets consisting of 1, 10, 50, 100 expert episodes collected from question 1.1, we use the model configuration same to the expert and use hyper-parameters listed in table 1. Since it is a supervised learning process, the cross entropy is used as the objective function to minimize. Table 2 shows the training results over these 4 datasets. With more samples from the expert model, the model could be trained better and have higher accuracy. We take video clips for all 4 models and find that, more samples from the expert model helps our agent to maintain a policy robust enough for different initial status and different trajectories.

Further analysis in Question 1.3 also proves this point.

The video clips are under the directory of "./video/Question1".

Hyper-parameter	Configuration
learning rate	0.001
epoch number	100
optimizer	Adam optimizer from Keras
batch size	16

Table1: hyper-parameters

Datasets	Final loss	Final accuracy
1 episode	0.6741	67.77 %
10 episodes	0.1626	92.84 %
50 episodes	0.0932	96.19 %
100 episodes	0.0772	96.88 %

Table 2: final loss and accuracy

3. Run the expert policy and each of your cloned models on `LunarLander-v2`, and record the mean/std of the total reward over 50 episodes. How does the amount of training data affect the cloned policy? How do the cloned policies' performance compare with that of the expert policy?

   Results:

   Figure ?? shows the values of "mean/std" of the total rewards over 50 episodes. Results of both the expert model and our agent are records. With more training data, the difference of normalized total rewards between the expert model and our agents is smaller, which indicates more data help the training process and our agent could memorize more types of polices given a states and our agents are more similar to the expert. From the video clips, our agent almost clones the behavior of the expert model with enough training data. But our agents trained from 50 and 100 sampled episodes show a common behavior pattern that the strategies of handling different initial status are not as diverse as the expert's, even though our agent can solve the task. So with even more data, our agent might be more intelligent.

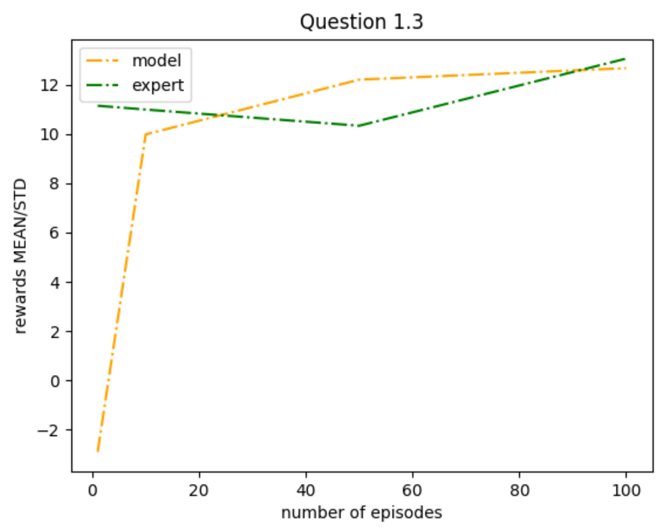


Figure 1: Question 1.3 plot

Problem 2: 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

```
1: procedure REINFORCE
2:     Start with policy model $\pi_\theta$
3:     repeat:
4:         Generate an episode $S_0, A_0, r_0, \ldots, S_{T-1}, A_{T-1}, r_{T-1}$ following $\pi_\theta(\cdot)$
5:         for $t$ from $T-1$ to 0:
6:             $G_t = \sum_{k=t}^{T-1} \gamma^{k-t} r_k$
7:         $L(\theta) = \frac{1}{T} \sum_{t=0}^{T-1} G_t \log \pi_\theta(A_t|S_t)$
8:         Optimize $\pi_\theta$ using $\nabla L(\theta)$
9: 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[1], and answer the following questions:

1. Describe your implementation, including the optimizer and any hyperparameters you used (learning rate, γ, etc.).

   Result:

   We follow the provided pseudo-code to implement the REINFORCE model. For the policy model, we use Adam optimizer, and implement the customized loss function. The learning rate of the actor model is 0.0005, which is the default setting. Training episodes are sampled according to current policy model. We treat each episode as a mini-batch, i.e. update the policy model in each episode. Test episode reward is computed every 500 episodes, by averaging 100 sample episodes. γ is 0.99 in our setting. The model is trained for 30000 episodes.

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

   Result: Figure 2 shows the plot of the mean cumulative reward vs. the number of training episodes.

3. Discuss, in detail, how the learned policy compares to our provided expert and your cloned models.

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

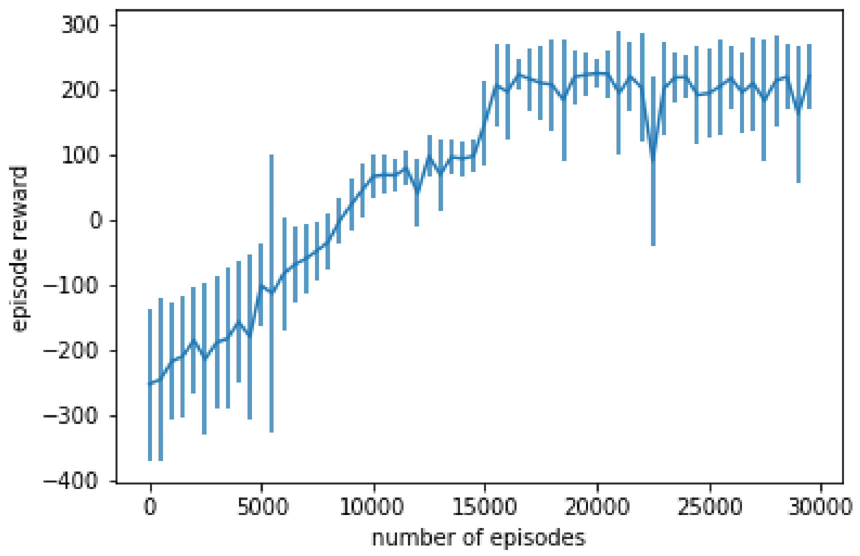


Figure 2: The average and standard deviation of episode rewards for REINFORCE model.

Result: Compared with the learned policy with behavior cloning, the policy learned by REINFORCE algorithm is more robust. Some video clips are shown under the movie directory. For initial states in which the Lunar Lander doesn't drop directly instead have some initial horizontal speed, the trajectories formed by two policies are different. The cloned policy shows more similarity with the expert model, which may produce many similar rigid trajectories given the initial states with horizontal speed. But the trained policy with REINFORCE doesn't perform like this, instead, the Lunar Lander can have flexible solution to land on the target. More intuitively, the cloned policy shows a conditioned reflex pattern but the reinforced policy makes decisions based on the states and future rewards, which may have some flexibility but the variance of rewards are larger then the cloned policy because the final state seen by the cloned model are limited. Once the cloned model encounters a state quite different from the training data, the model may fail.

Problem 3: 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.

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 param-

Algorithm 2 N-step Advantage Actor-Critic

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

eters θ in the direction suggested by the N-step trace.

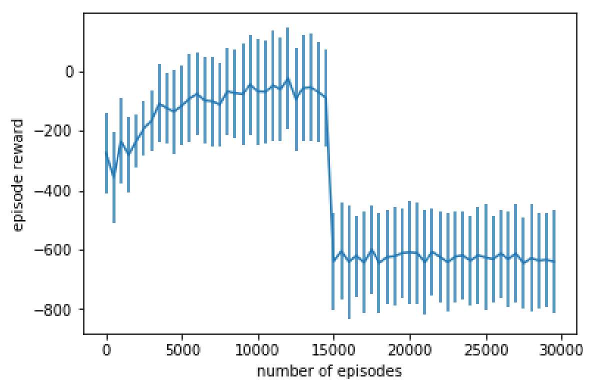
As in Problem 2, use the network architecture for the policy model $\pi(A \mid S, \theta)$ provided in `LunarLander-v2-config.json`. Play around with the network architecture of the Critic's state-value approximator to find one that works for `LunarLander-v2`. You can choose which optimizer and hyperparameters 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:

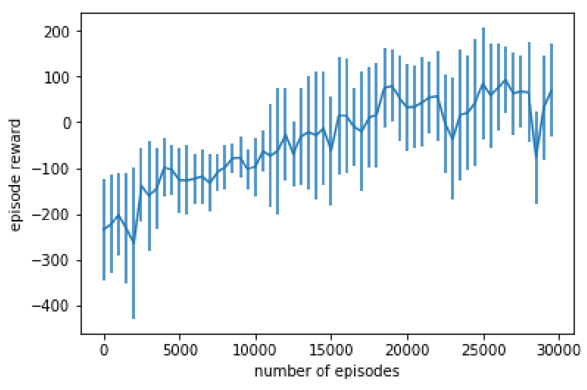
1. Describe your implementation, including the optimizer, the critic's network architecture, and any hyperparameters you used (learning rate, γ, etc.).

   Answer:

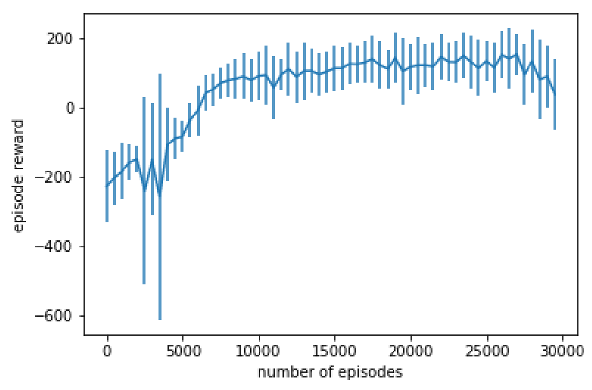
   Basically, we follow the provided pseudo-code in the writeup. For the actor model, we use Adam optimizer, and implement the customized loss function. The learning rate of the actor model is 0.0005, which is the default setting. For the critic model, we implement a 4-hidden-layer forward DNN, with 32, 64, 64, 32 dimensions separately. The reason for this architecture of the critic model is that simpler model can't model the mapping between states and the value function. We have tried some configuration of the critic model like even more complicated 5-layer DNN model (8 input-32 units-64 units-128 units-64 units-32 units-1 output). Complex models can approximate the value function. Finally, we choose a simpler one as the critic model. The output layer is simply a fully-connected layer. The learning rate of the critic model is 0.0001, which is also the default setting. We treat each episode as a mini-batch, i.e. update both actor and critic model in each episode. Test episode reward is computed every 500 episodes, by averaging 100 sample episodes. For the discount rate, we try some values and find that, if γ is small, the agent can only consider the effect of few future rewards while a large γ like 1 will make the decision making at the early steps more difficult



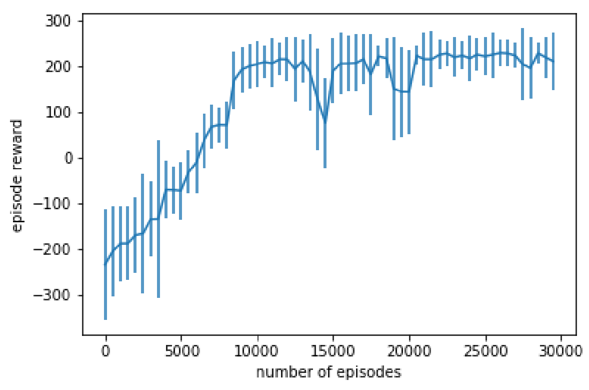
(a) n = 1



(b) n = 20



(c) n = 50



(d) n = 100

Figure 3: The average and standard deviation of episode rewards for advantage actor-critic models.

since the final step (with rewards -100 or 100) will have a great effect on the early steps. In the end, γ is 0.99 in our setting. All models are trained for 30000 episodes.

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.

   Result:

   Figure 3 is the plot of the learning curves for N equals [1, 20, 50, 100] separately.

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?

   Discussion:

   - We find that A2C algorithm learns faster than REINFORCE algorithm. It takes REINFORCE model around 16000 episodes to reach 200 rewards, but A2C model (N = 100) can reach 200 rewards with only 9000 training episodes.

- We find that when N becomes larger, the A2C is easier to converge and the test standard deviation is smaller. This is because we are using TD(n) to estimate the value function. Small n introduces large randomness while larger n can provide better estimation of the V^{π}. In our experiment, only when n = 100, the model can converge to 200 rewards. When n = 50 or 20, it can achieve some progresses (reach 170 rewards for n = 50, 80 rewards for n = 20).

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!**

References

[1] J Andrew Bagnell. An invitation to imitation. Technical report, DTIC Document, 2015.

[2] Stephane Ross. Interactive learning for sequential decisions and predictions. 2013.

[3] Stephane Ross, Geoffrey J Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In AISTATS, volume 1, page 6, 2011.