University of California Berkeley

Computer Science Department

CS 294-112

HW2: Policy Gradient and Variance Reduction

Submitted by
Cristobal Pais
CS 294-112
Sept 19th
Fall 2018

HW2: Policy Gradient and Variance Reduction

Cristobal Pais

Sept 19th 2018

1 Problem 1: State-dependent baseline

a) Following the statement of the homework we can express $p_\theta(\tau)$ as a product of the state-action marginal (s_t, a_t) and the probability of the rest of the trajectory conditioned on (s_t, a_t), denoted as $(\tau/s_t, a_t|s_t, a_t)$. Therefore, we have:

$$p_\theta(\tau) \quad = \quad p_\theta(s_t, a_t) p_\theta(\tau/s_t, a_t|s_t, a_t) \tag{1}$$

Now, we proceed to show — using the law of iterated expectations — that subtracting a state-dependent baseline from the returns keep the policy gradient unbiased, by decoupling the state-action marginal from the rest of the trajectory:

$$\begin{aligned}
\mathbb{E}_{\tau\sim p_\theta}\left[\nabla_\theta log\pi_\theta(a_t|b_t)b(s_t)\right] &= \mathbb{E}_{(s_t,a_t)\sim p_\theta(s_t,a_t)}\left[\nabla_\theta log\pi_\theta(a_t|b_t)b(s_t)\right] && (2)\\
&\underbrace{=}_{def} \mathbb{E}_{s_t}\left[\mathbb{E}_{a_t}\left[\nabla_\theta log\pi_\theta(a_t|b_t)b(s_t)\right]\right] && (3)\\
&\underbrace{=}_{fixed\ s_t} \mathbb{E}_{s_t}\left[b(s_t)\underbrace{\mathbb{E}_{a_t}\left[\nabla_\theta log\pi_\theta(a_t|b_t)\right]}_{SF=0}\right] && (4)\\
&= \mathbb{E}_{s_t}\left[0\right] && (5)\\
&= 0 && (6)
\end{aligned}$$

Since we know from lectures that SF is equal to zero by the known "log-derivative trick", we can conclude that equation (12) of the problem statement holds and thus, the subtraction of a state-dependent baseline function is not introducing bias in our policy gradient.

b) Using the structure of the MDP, we can express $p_\theta(\tau)$ as follows:

$$p_\theta(\tau) = p_\theta(s_{1:t}, a_{1:t-1}) p_\theta(s_{t+1:T}, a_{t:T}|s_{1:t}, a_{1:t-1}) \tag{7}$$

We proceed to show (using the law of iterated expectations) that subtracting a state-dependent baseline from the return keeps the policy gradient unbiased, by conditioning on the trajectory τ up to time-step t.

i) Due to the memoryless property of the Markovian Decision Processes, we know that conditioning on a trajectory up to time t (in terms of states and actions) is equivalent to conditioning on the current state s_t since it contains all the information needed for calculating the transition to the next time step $t+1$ and thus, no past information of the trajectory is needed for the expectation. Therefore, we can simplify the notation and calculate the conditional expectation with respect to s_t, as we will see in the following item.

ii) We start by using the law of iterated expectations by conditioning on the trajectory $(s_1, a_1, ..., s_T, a_T)$ as follows:

$$\begin{aligned}
\mathbb{E}_{\tau\sim p_\theta}\left[\nabla_\theta log\pi_\theta(a_t|b_t)b(s_t)\right] &= \mathbb{E}_{s_{1:t},a_{1:t-1}}\left[\mathbb{E}_{s_{t+1:T},a_{t:T}}\left[\nabla_\theta log\pi_\theta(a_t|s_t)b(s_t)|s_{1:t},a_{1:t-1}\right]\right] && (8)\\
&\underbrace{=}_{i)} \mathbb{E}_{s_{1:t},a_{1:t-1}}\left[\mathbb{E}_{s_{t+1:T},a_{t:T}}\left[\nabla_\theta log\pi_\theta(a_t|s_t)b(s_t)|s_t\right]\right] && (9)\\
&\underbrace{=}_{(*)} \mathbb{E}_{s_{1:t},a_{1:t-1}}\left[b(s_t)\underbrace{\mathbb{E}_{s_{t+1:T},a_{t:T}}\left[\nabla_\theta log\pi_\theta(a_t|s_t)|s_t\right]}_{D}\right] && (10)\\
&\underbrace{=}_{(**)} \mathbb{E}_{s_{1:t},a_{1:t-1}}\left[b(s_t)\mathbb{E}_{a_t}\left[\nabla_\theta log\pi_\theta(a_t|s_t)\right]\right] && (11)\\
&\underbrace{=}_{pdf} \mathbb{E}_{s_{1:t},a_{1:t-1}}\left[b(s_t)\nabla_\theta 1\right] && (12)\\
&= 0 && (13)
\end{aligned}$$

First, we break the expectation using the law of iterated expectations, conditioning on the trajectory as suggested in the statement of the problem. Applying our discussion in i), we simplify the notation, conditioning on s_t. On (*), we notice that $b(s_t)$ is a constant function since s_t is fixed so it can be taken out from the expectation. Then, in (**) we notice that we can omit all the unnecessary variables of the inner expectation because we can break D as follows (assuming both discrete and continuous settings where A and S are the set of actions and states, respectively):

Discrete

$$\begin{aligned}
D &= \mathbb{E}_{s_{t+1:T},a_{t:T}}\left[\nabla_\theta log\pi_\theta(a_t|s_t)|s_t\right] && (14)\\
&= \sum_{a_t\in A}\sum_{s_{t+1}\in S}...\sum_{s_T\in S}\pi_\theta(a_t|s_t)p(s_{t+1}|s_t,a_t)...p(s_T|s_{T-1},a_{T-1})\nabla_\theta log\pi_\theta(a_t|s_t) && (15)\\
&\underbrace{=}_{\text{re-arranging}} \sum_{a_t\in A}\pi_\theta(a_t|s_t)\nabla_\theta log\pi_\theta(a_t|s_t)\sum_{s_{t+1}\in S}p(s_{t+1}|s_t,a_t)...\sum_{s_T\in S}p(s_T|s_{T-1},a_{T-1}) && (16)\\
&\underbrace{=}_{pdfs} \sum_{a_t\in A}\pi_\theta(a_t|s_t)\nabla_\theta log\pi_\theta(a_t|s_t) && (17)\\
&= \mathbb{E}_{a_t}\left[\nabla_\theta log\pi_\theta(a_t|s_t)\right] && (18)\\
&= \nabla_\theta 1 && (19)\\
&= 0 && (20)
\end{aligned}$$

Continuous

$$\begin{aligned}
D_{continuous} &= \mathbb{E}_{s_{t+1:T},a_{t:T}}\left[\nabla_\theta log\pi_\theta(a_t|s_t)|s_t\right] && (21)\\
&\underbrace{=}_{definition} \int_a\int_s\left(\nabla_\theta log\pi_\theta(a_t|s_t)\right)\pi_\theta(a|s)q_{s_t}(s|s_t)dads && (22)\\
&\underbrace{=}_{pdf} \nabla_\theta\mathbb{E}\left[1_{a_t\in A}|s_t\right] && (23)\\
&= 0 && (24)
\end{aligned}$$

The previous equations are correct since we can simply re-arrange all the terms inside the expectation such that we get a series of summations over densities which are equal to one. Therefore, we are able to obtain the same expression as with the constant baseline since we are computing the expectation of the conditional distribution on the state s_t ($\mathbb{E}_{a_t}$) that is clearly equal to 1, and thus, we obtain the last step indicated in equations (20) and (24).

Note: $\pi_\theta(a|s)q_{st}(s|s_t)$ is the conditional distribution of the state s given the current state s_t as defined by the MDP equation in (7).

2 Problem 4: CartPole experiments

In this section, we compare the performance of three different settings for our training framework: (1) a simple implementation without reward-to-go or advantage centering (sb_no_rtg_dna), with a reward-to-go function (sb_rtg_dna), and including reward-to-go and advantage centering (sb_rtg_na). In addition, a comparison between the performance obtained by using small (1000) or large (5000) batch sizes is performed.

2.1 Small batch-size experiments (sb)

In order to test the impact of our reward-to-go and advantages centering approaches when using small batches of data ($b = 1000$), we run the following experiments:

```
python train_pg_f18.py CartPole-v0 -n 100 -b 1000 -e 3 -dna
    --exp_name sb_no_rtg_dna
```

```
python train_pg_f18.py CartPole-v0 -n 100 -b 1000 -e 3 -rtg -dna
    --exp_name sb_rtg_dna
```

```
python train_pg_f18.py CartPole-v0 -n 100 -b 1000 -e 3 -rtg
    --exp_name sb_rtg_na
```

From them, we generate a unique plot with the average returns on the y-axis and the number of iteration on the x-axis using our wrapper implementation of the original plot.py file using the following command:

```
python MultiPlot.py -re "^sb.*\_CartPole" -o CartPoleSB
```

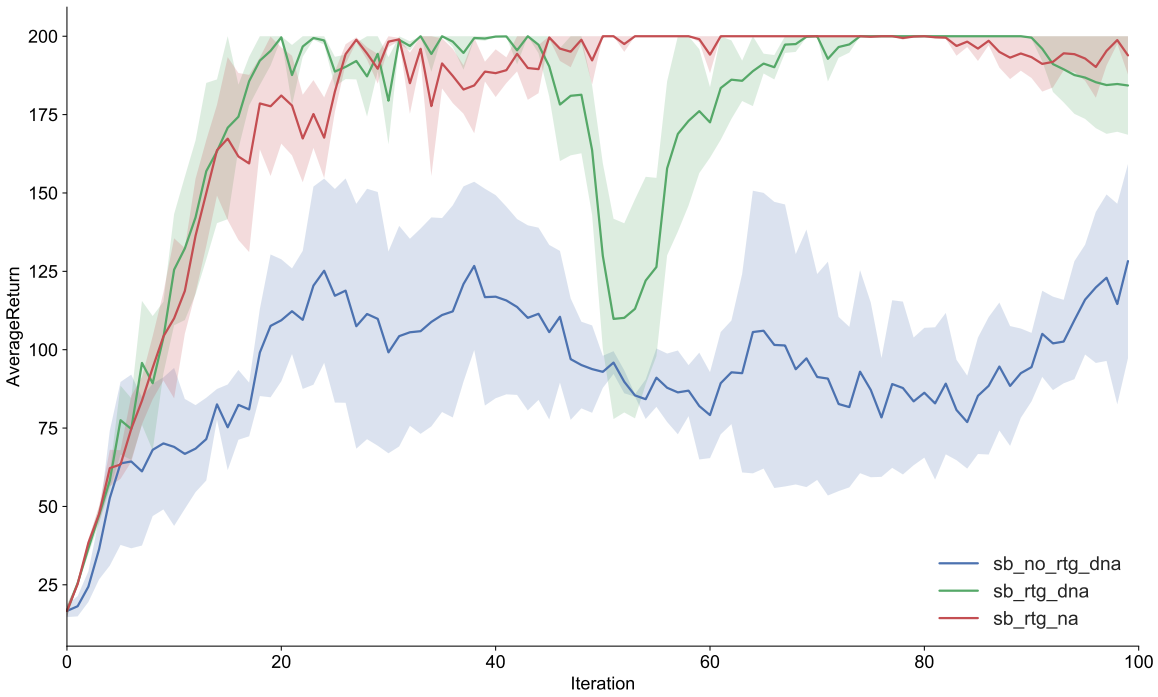

Figure 1: Comparison of the average returns obtained by the small batch size experiments with the CartPole instance. Curves without/with advantage centering and using a reward-to-go model are shown.

In Figure 1, we can see the average returns of the three previous configurations. From it, we can observe that the best performance is obtained when using the reward-to-go and advantage normalization approaches (red line). When advantage centering is not present, performance is similar but it requires more iterations in order to reach the theoretical maximum performance (green line). On the other hand, poor performance is achieved when using the vanilla implementation, not being able to reach the theoretical maximum performance (average return equals to 200).

2.2 Large batch-size experiments (lb)

In order to test the impact of our reward-to-go and advantages centering approaches when using large batches of data ($b = 5000$), we run the following experiments:

```
python train_pg_f18.py CartPole-v0 -n 100 -b 5000 -e 3 -dna
    --exp_name lb_no_rtg_dna
```

```
python train_pg_f18.py CartPole-v0 -n 100 -b 5000 -e 3 -rtg -dna
    --exp_name lb_rtg_dna
```

```
python train_pg_f18.py CartPole-v0 -n 100 -b 5000 -e 3 -rtg
    --exp_name lb_rtg_na
```

From them, we generate a unique plot with the average returns on the y-axis and the number of iteration on the x-axis using our wrapper implementation of the original plot.py file using the following command:

```
python MultiPlot.py -re "^lb.*\_CartPole" -o CartPoleLB
```

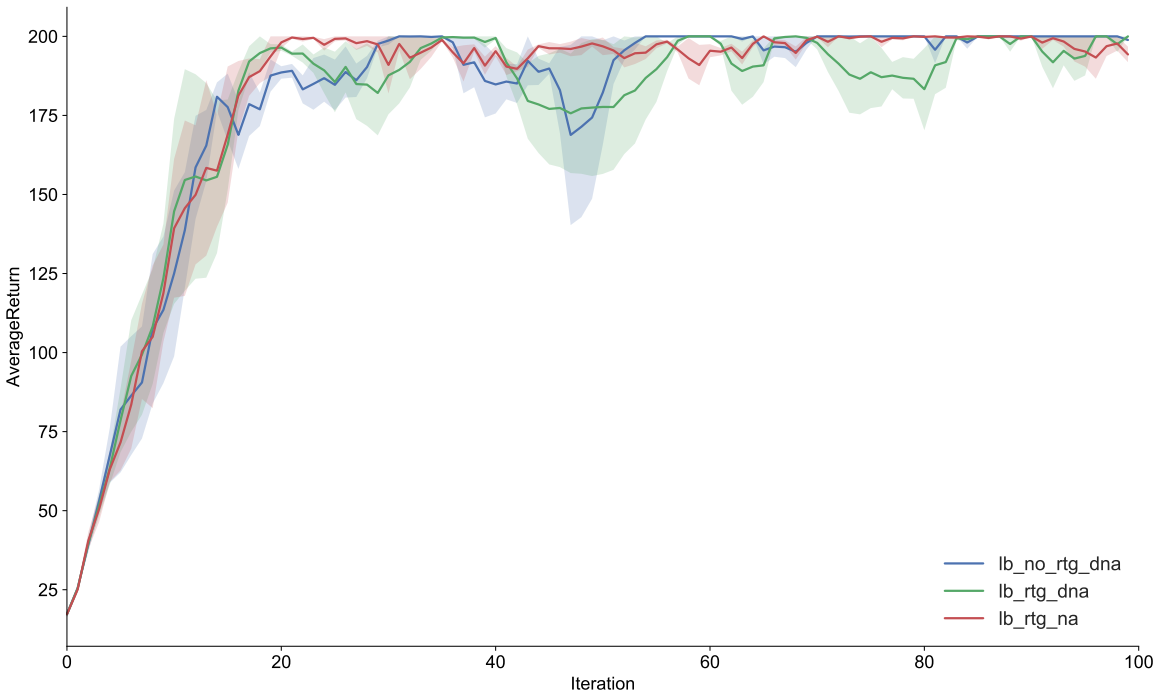

Figure 2: Comparison of the average returns obtained by the large batch size experiments with the CartPole instance. Curves without/with advantage centering and using a reward-to-go model are shown.

In Figure 2, we can see the average returns of the three previous configurations. From it, we can observe that the best performance is (again) obtained when using the reward-to-go and advantage normalization approaches (red line) in terms of convergence speed and stability. When advantage centering is

Figure 3: Summary comparison of the average returns obtained by all CartPole experiments.

not present (green line), performance is similar but it requires more iterations to reach the theoretical maximum performance. In contrast to the small batch size experiments, a similar performance is achieved when using the vanilla implementation, being able to reach the theoretical maximum performance (average return equals to 200), even improving the final results obtained by the reward-to-go implementation.

2.3 Results & Discussion

In order to compare all the implementations, we generate Figure 3 using the following command:

```
python MultiPlot.py -re "CartPole" -o CartPoleAll
```

Based on the the results presented in Figures 1, 2, and 3, we can answer observe the following patterns:

i) As seen in all previous results, the gradient estimator which achieved the best performance is the one including the reward-to-go and advantage centering — as expected from the theory. It tends to learn faster and stable.

ii) The advantage centering approach seems to slightly impact (positive) the performance of the learning curve, obtaining very similar results in comparison to the non-centered approach in terms of average returns and convergence rate. However, its impact is not as strong as the inclusion of the reward-to-go approach.

iii) The batch size has a significant impact on both the variation and the convergence speed towards the optimal value. A larger batch size leads to smoother learning curves (smaller variance) as can be clearly seen in the Figures and also, it requires less number of iterations to reach higher average return levels.

3 Problem 5: Inverted Pendulum

In order to find the smallest batch size b^* and largest learning rate r^* that gets the optimum (maximum score of 1000) in less than 100 iterations, we perform a 2-dimensional search following a brute force approach by running a simple bash code inside the InvertedPendulum.sh file:

```
#!/bin/bash
LR=0.001
BS=8

for i in {1..128};
    do
        for j in {1..100};
            do
                echo "BS: "${BS}"  Learning-Rate: "${LR}
                python train_pg_f18_Full.py InvertedPendulum-v2
                      -ep 1000 --discount 0.9 -n 100 -e 3 -l 2 -s 64
                      -b ${BS} -lr ${LR} -rtg
                      --exp_name ip_b${BS}_r${LR}
                LR="$(echo ${LR} + 0.001 | bc -l)"
            done
        BS="$(echo ${BS} + 8| bc -l)"
        LR=0.001
    done
```

Using this simple code, we are able to generate all the combinations of batch sizes $b \in [8, 128]$ by 8 units and all learning rates $r \in [0.001, 1]$ separated by 0.001. Then, a simple visual analysis is performed using our MultiPlot.py wrapper, plotting all learning rates combinations per batch size, registering the tuples that were able to reach the maximum score of 1000. The following command is used for this step, where $< b >$ is replaced by a fixed batch size value and X is replaced by any integer $\in [0, 10]$:

```
python MultiPlot.py -re "ip_<b>_r.0[X]_" -o InvertedPendulum
```

Once we filtered the combinations that are able to reach the maximum score, a simple manual filtering procedure is performed, eliminating those combinations that are dominated (e.g. (10, 0.02) is dominated by (9, 0.03) since the batch size of the second configuration is smaller and the learning rate is larger than in the first one). Finally, we plot the best results in a unique plot and one of them as an example using the following commands (by previously appending IPOptimal to the selected combinations data folders):

```
python MultiPlot.py -re "ipOPT" -o ipAll
```

```
python MultiPlot.py -re "ip_b8_r.012" -o BestInvertedPendulum
```

Obtaining the results shown in Figures 4 and 5. From it, we can clearly see how the selected combinations are able to reach the maximum score in less than 100 iterations. Different variance levels and fluctuations in the convergence rate can be seen depending on the combination of batch size and learning rate.

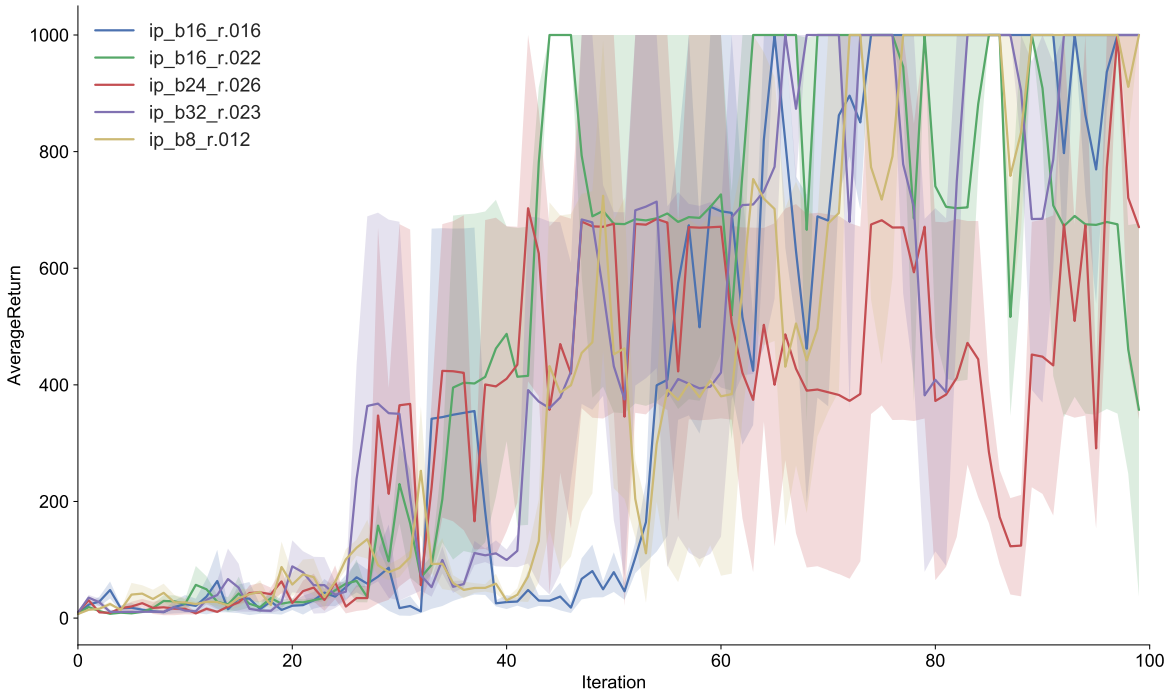

Figure 4: Summary of some of the best combinations of batch size and learning rate obtained for the Inverted Pendulum instance. Several other "optimal combinations" were obtained and are not included for visualization purposes.

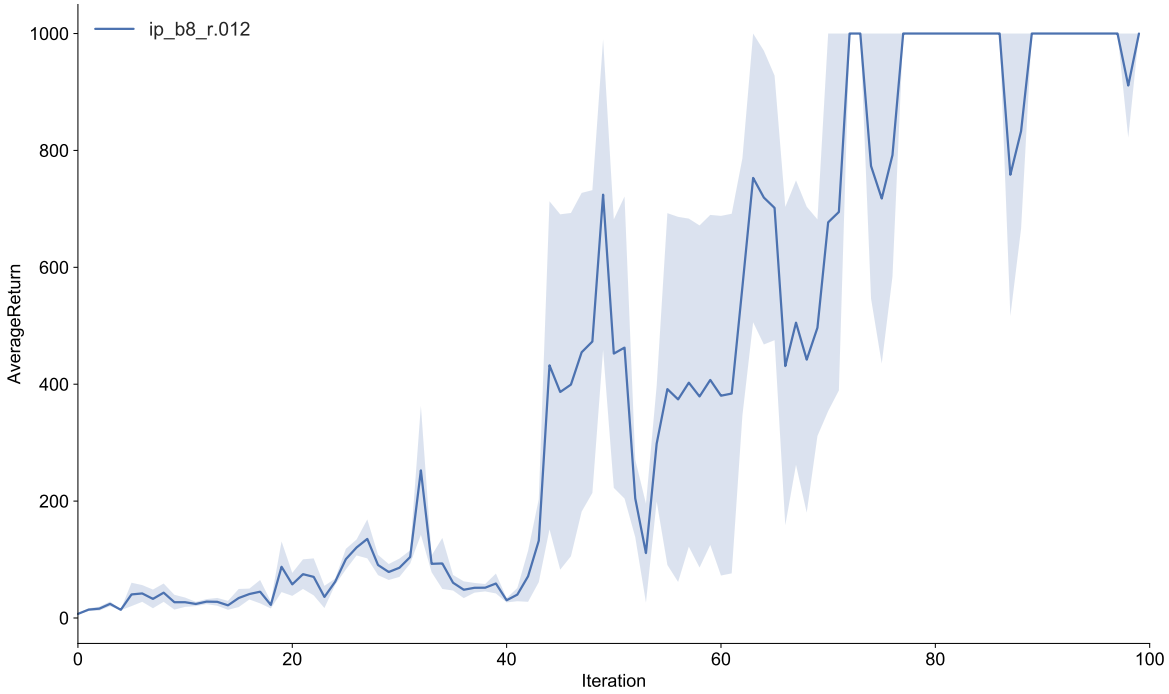

Figure 5: Detailed evolution for combination $(b, r) = (8, 0.012)$ tested in the Inverted Pendulum instance.

4 Problem 7: Lunar Landing

In order to test our baseline implementation, we use our learning framework on the LunarLanderContinuous-v2 instance using the following command:

```
python train_pg_f18.py LunarLanderContinuous-v2 -ep 1000
    --discount 0.99 -n 100 -e 3 -l 2 -s 64 -b 40000
    -lr 0.005 -rtg --nn_baseline --exp_name ll_b40000_r0.005
```

Once the training is complete, we generate the Average return versus the number of iteration graph using the following command:

```
python MultiPlot.py -re "ll" -o LunarLanding
```

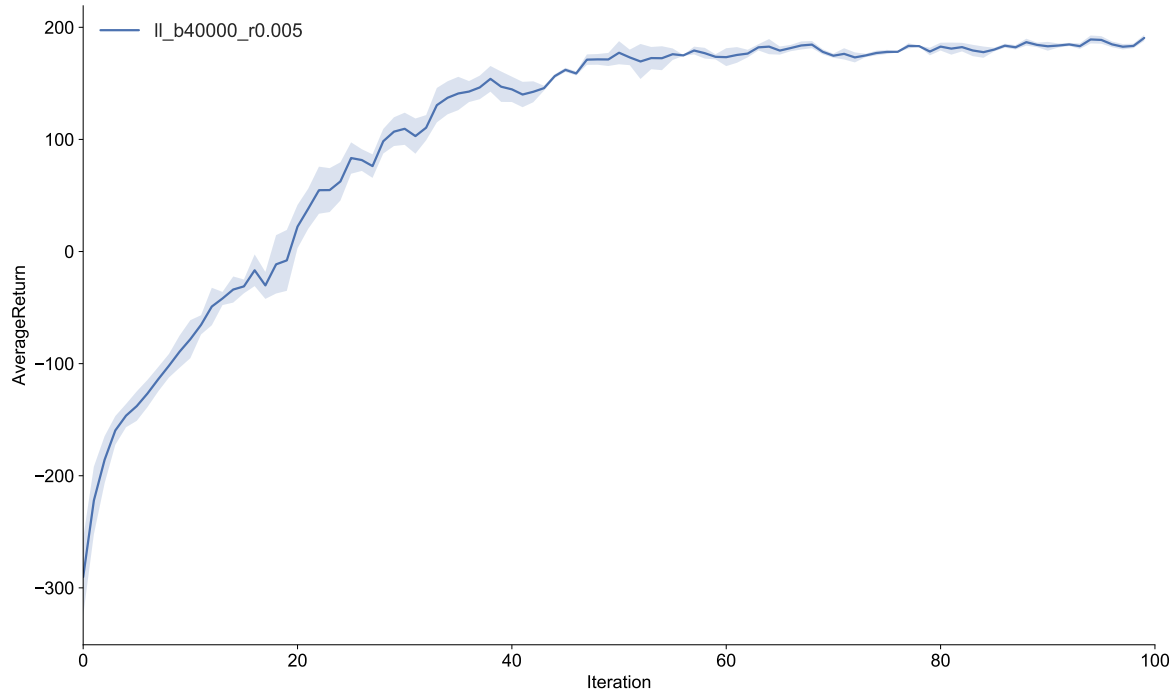

Figure 6: Lunar Landing average return evolution using a baseline implementation and reward-to-go approach during 100 iterations. The theoretical maximum average return value (180) is achieved around iteration 50 and a very low variance can be appreciated along its evolution.

From the results in Figure 6, we can clearly see how our baseline implementation applied on the Lunar Landing environment is able to reach an average return around 180 within the first 100 iterations, allowing us to conclude that the implemented baseline method is working properly.

5 Problem 8: HalfCheetah

5.1 Batch size and learning rate tuning

Following the statement of the problem and the latest update in Piazza, we perform a search for the optimal combination of batch size $b \in \{10000, 30000, 50000\}$ and learning rate $r \in \{0.005, 0.01, 0.02\}$ with reward-to-go and baseline function using the following commands:

```
python train_pg_f18.py HalfCheetah-v2 -ep 150 --discount 0.95 -n 100
    -e 3 -l 2 -s 32 -b 10000 -lr 0.005 -rtg --nn_baseline
    --exp_name hc_b50000_r0005_tuning
```

```
python train_pg_f18.py HalfCheetah-v2 -ep 150 --discount 0.95 -n 100
    -e 3 -l 2 -s 32 -b 30000 -lr 0.005 -rtg --nn_baseline
    --exp_name hc_b30000_r0005_tuning
```

```
python train_pg_f18.py HalfCheetah-v2 -ep 150 --discount 0.95 -n 100
    -e 3 -l 2 -s 32 -b 50000 -lr 0.005 -rtg --nn_baseline
    --exp_name hc_b50000_r0005_tuning
```

```
python train_pg_f18.py HalfCheetah-v2 -ep 150 --discount 0.95 -n 100
    -e 3 -l 2 -s 32 -b 10000 -lr 0.01 -rtg --nn_baseline
    --exp_name hc_b10000_r001_tuning
```

```
python train_pg_f18.py HalfCheetah-v2 -ep 150 --discount 0.95 -n 100
    -e 3 -l 2 -s 32 -b 30000 -lr 0.01 -rtg --nn_baseline
    --exp_name hc_b30000_r001_tuning
```

```
python train_pg_f18.py HalfCheetah-v2 -ep 150 --discount 0.95 -n 100
    -e 3 -l 2 -s 32 -b 50000 -lr 0.01 -rtg --nn_baseline
    --exp_name hc_b50000_r001_tuning
```

```
python train_pg_f18.py HalfCheetah-v2 -ep 150 --discount 0.95 -n 100
    -e 3 -l 2 -s 32 -b 10000 -lr 0.02 -rtg --nn_baseline
    --exp_name hc_b10000_r002_tuning
```

```
python train_pg_f18.py HalfCheetah-v2 -ep 150 --discount 0.95 -n 100
    -e 3 -l 2 -s 32 -b 30000 -lr 0.02 -rtg --nn_baseline
    --exp_name hc_b30000_r002_tuning
```

```
python train_pg_f18.py HalfCheetah-v2 -ep 150 --discount 0.95 -n 100
    -e 3 -l 2 -s 32 -b 50000 -lr 0.02 -rtg --nn_baseline
    --exp_name hc_b50000_r002_tuning
```

As can be seen in Figure 7, the combination that achieved the best performance is the one using a batch size $b = 50000$ and a learning rate $r = 0.02$. Analyzing the plot, it is possible to understand the impact of the batch size and learning rate in the performance of the training:

- Increasing the batch size benefits the performance of the model but requires more computational resources and thus, running times tend to be larger than with smaller values. This result is expected since we are including more information every step to the model, being able to significantly improve its performance.

- As expected, increasing the learning rate allows the model to reach better performance in shorter training periods (less number of iterations). This happens since the larger value (0.02) is still small enough to be able to capture the complexities of the instance as well as "pushing" the policy to reach better average return levels. For completeness, we tested higher and smaller learning rates values leading to worse performance than the optimal $(50000, 0.02)$.
- In addition, we can observe that the impact of the learning rate value seems stronger than the selected batch size, as can be seen — for example — when comparing the performance of $(10000, 0.02)$ and $(30000, 0.01)$ where the first combination is clearly superior to the second one, noticing that the size of the batch in the later one is three times the size of the first combination.

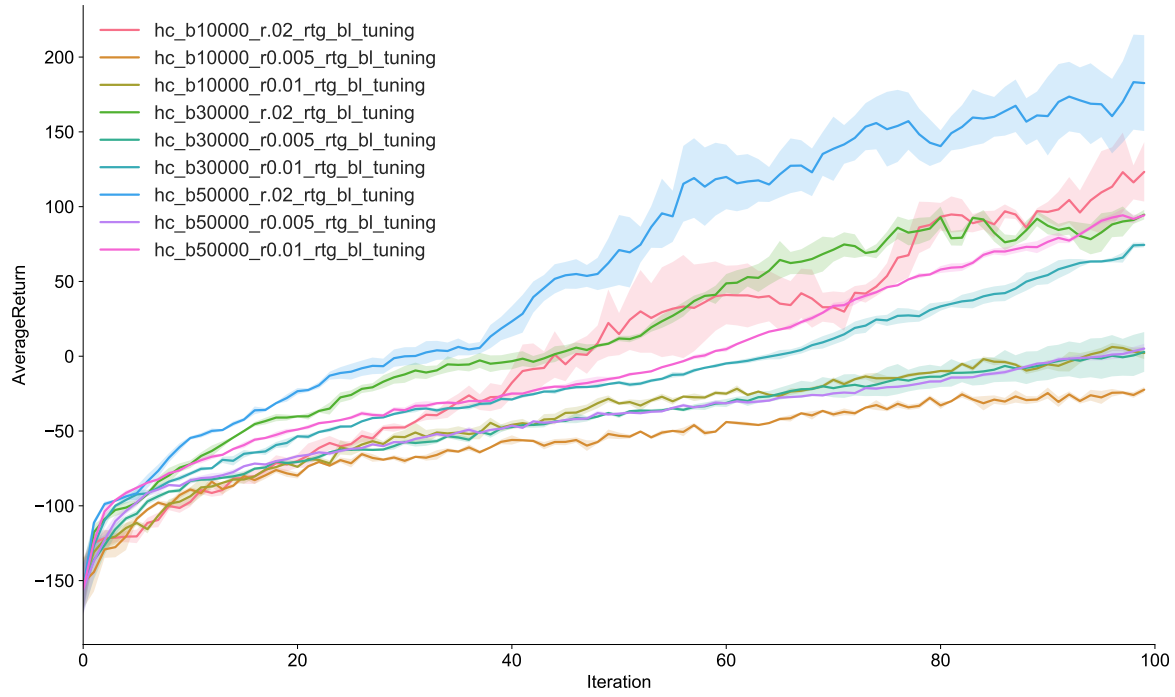

Figure 7: Performance comparison of different batch sizes and learning rates for the Half-Cheetah-v2 instance including reward-to-go and baseline implementation. Best performance is achieved with $(b, r) = (50000, 0.02)$ reaching average returns close to the expected maximum (200).

Note: The following command was used to generate Figure 7:

```
python MultiPlot.py -re "hc.*tuning" -o HCTuning
```

5.2 Optimal Half-Cheetah performance: reward-to-go and baselines

Using the optimal batch size $b^* = 50000$ and learning rate $r = 0.02$, we proceed to run the following commands for performance comparison:

```
python train_pg_f18.py HalfCheetah-v2 -ep 150 --discount 0.95 -n 100
    -e 3 -l 2 -s 32 -b 50000 -lr 0.02
    --exp_name hc_b50000_r002_vanilla_best
```

```
python train_pg_f18.py HalfCheetah-v2 -ep 150 --discount 0.95 -n 100
    -e 3 -l 2 -s 32 -b 50000 -lr 0.02 -rtg
    --exp_name hc_b50000_r002_rtg_best
```

```
python train_pg_f18.py HalfCheetah-v2 -ep 150 --discount 0.95 -n 100
    -e 3 -l 2 -s 32 -b 50000 -lr 0.02 --nn_baseline
    --exp_name hc_b50000_r002_bl_best
```

```
python train_pg_f18.py HalfCheetah-v2 -ep 150 --discount 0.95 -n 100
    -e 3 -l 2 -s 32 -b 50000 -lr 0.02 -rtg --nn_baseline
    --exp_name hc_b50000_r002_rtg_bl_best
```

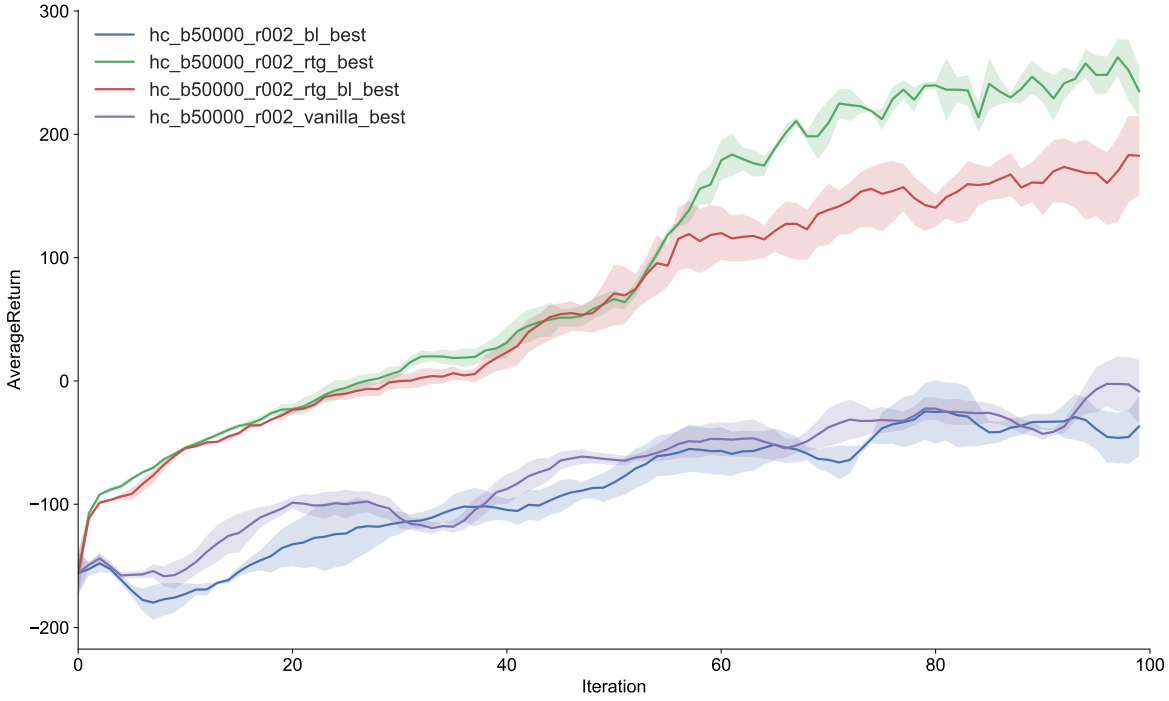

Figure 8: Different configurations performance comparison for the Half-Cheetah-v2 instance using the optimal batch size and learning rate found in section 5.2. Best combinations implement reward-to-go wit/without baseline flags, reaching the theoretical maximum value (200). Other random seeds were tested, obtaining very similar performance with and without baseline when using reward-to-go.

From the results in Figure 8 we can see that the best training performance is obtained by the model including reward-to-go and (or without) baseline implementations — as expected from the theory — where a final average return close to the theoretical maximum 200 is obtained. On the other hand, poor performance is achieved when using the vanilla implementation reaching average return around -50, showing the known limitations of this simple algorithmic approach. Interesting is to observe that a similar performance is achieved when a baseline (but no reward-to-go) is implemented, showing the

small impact of this technique in this environment. From the plot, we can see the impact on the training performance when using a reward-to-go and/or a baseline function:

i) The reward-to-go approach seems to significantly impact the performance of our policy gradient, reflecting the fact that without it, our implementations tend to reach average returns around -50 while including it allows our policy to reach the theoretical average level around 200 (red and green lines).

ii) Including a state-dependent baseline function does not significantly impact the performance during the training phase by itself since similar results are obtained when comparing its implementation (blue line) and the vanilla algorithm (purple line).

Note: Figure 8 is generated using the following command:

```
python MultiPlot.py -re "hc.*best" -o HCBest
```

6 Bonus: GAE-λ implementation

Following the lecture notes as well as the reference included in the homework statement (1), we include a simple implementation of the Generalized Advantage Estimation inside the Agent's *compute_advantage()* method. Using a new argument from the command line *–GAE_Lambda* or *-gl*, we pass the λ value to be used with GAE while modifying the value of the Q function inside the same method (we modified it such that it returns both the advantage and Q values instead of only the advantage array as in the original code).

In order to test our implementation, we proceed to perform a series of experiments with the suggested environment Walker2d-v1 from MuJoCo. Different values of $\lambda \in \{0, 0.5, 0.7, 0.9, 0.95, 0.97, 0.98, 0.99, 1\}$ are tested, recording the average returns versus the number of iterations for each model. For simplicity, we will test all values with reward-to-go and including a baseline implementation using the commands from the Half-Cheetah-v2 instance as a basis.

Therefore, we run the following commands (by changing λ value):

```
python train_pg_f18.py Walker-2d-v1 -ep 150 --discount 0.95 -n 100
       -e 3 -l 2 -s 32 -b 50000 -lr 0.02 -rtg --nn_baseline
       -gl <lambda> --exp_name w2d_b50000_r002_rtg_bl_gae<lambda>
```

Then, we generate Figure 9 using the following command:

```
python MultiPlot.py -re "w2d" -o W2dGAE
```

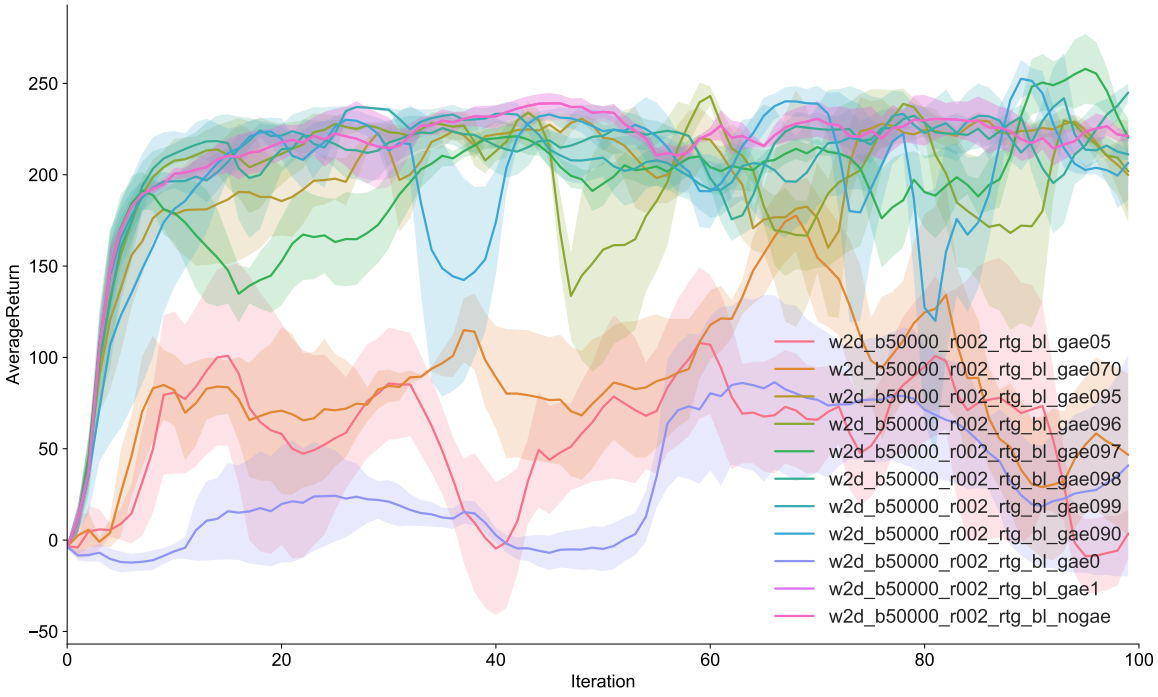

Figure 9: GAE-λ comparison results for the Walker-2d-v1 MujoCo's environment. Significant performance differences can be seen when varying the λ parameter, showing the impact of the method.

From the results in Figure 9, we can see the impact of different *lambda* values in the performance of the models: larger values of *lambda* tend to reach better average returns in shorter training times (see Table 1) and smaller values tend to require more iterations to reach reasonable results. In addition, very poor performance is obtained when using small values for λ such as 0, 0.5, and 0.75. As expected from the results in (1), *"choosing an appropriate intermediate value of* λ *in the range* $[0.9, 0.99]$ *usually results in*

λ	**AVG Time [s]**
0.00	5663.96
0.50	6147.55
0.70	4717.39
0.90	4752.80
0.95	3400.08
0.96	3253.51
0.97	5751.88
0.98	3490.81
0.99	4840.35
1.00	4797.29
No GAE	4813.05

Table 1: Walker2d-v2 average execution times for different GAE-*lambda* values. Best performance is obtained with values between 0.95 and 0.98.

the best performance", by adjusting and optimize the inherent bias-variance trade-off of these learning models.

A detailed summary for the average running times (across three different random seeds) can be seen in Table 1 where λ values between 0.95 and 0.99 tend to reach the best performance. In particular *lambda* = 0.98 obtains the best combined results in terms of final average return ($\sim$ 250) and total average running time (3480 [s]).

References

[1] Schulman, J., Moritz, P., Levine, S., Jordan, M., & Abbeel, P. (2015). High-dimensional continuous control using generalized advantage estimation. arXiv preprint arXiv:1506.02438.