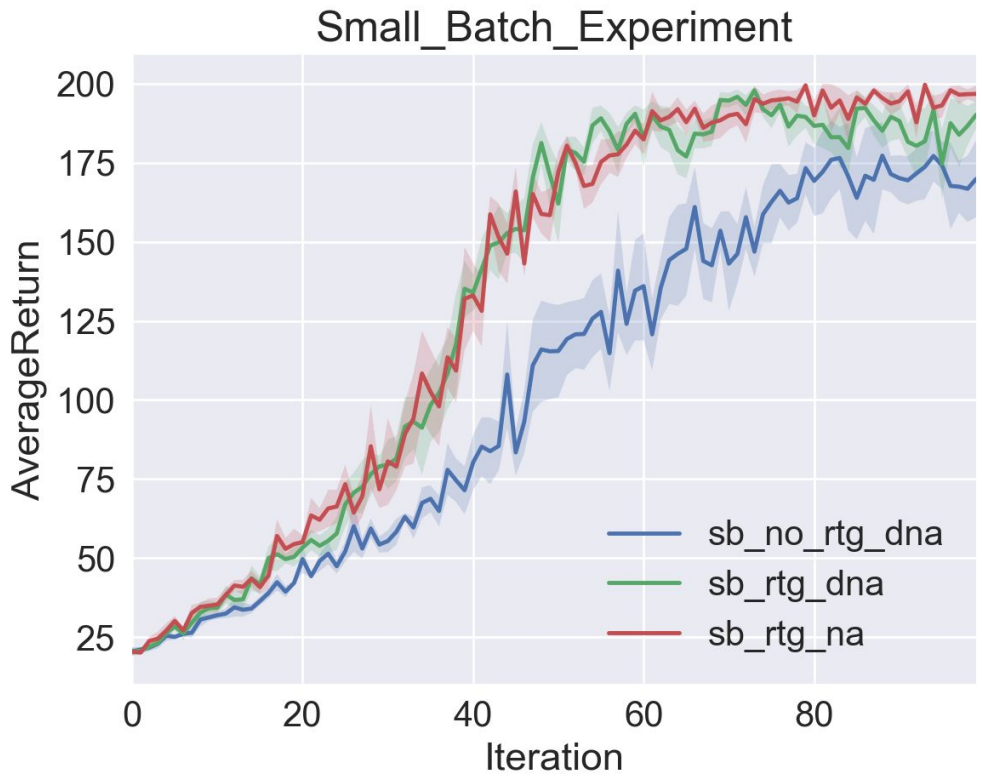
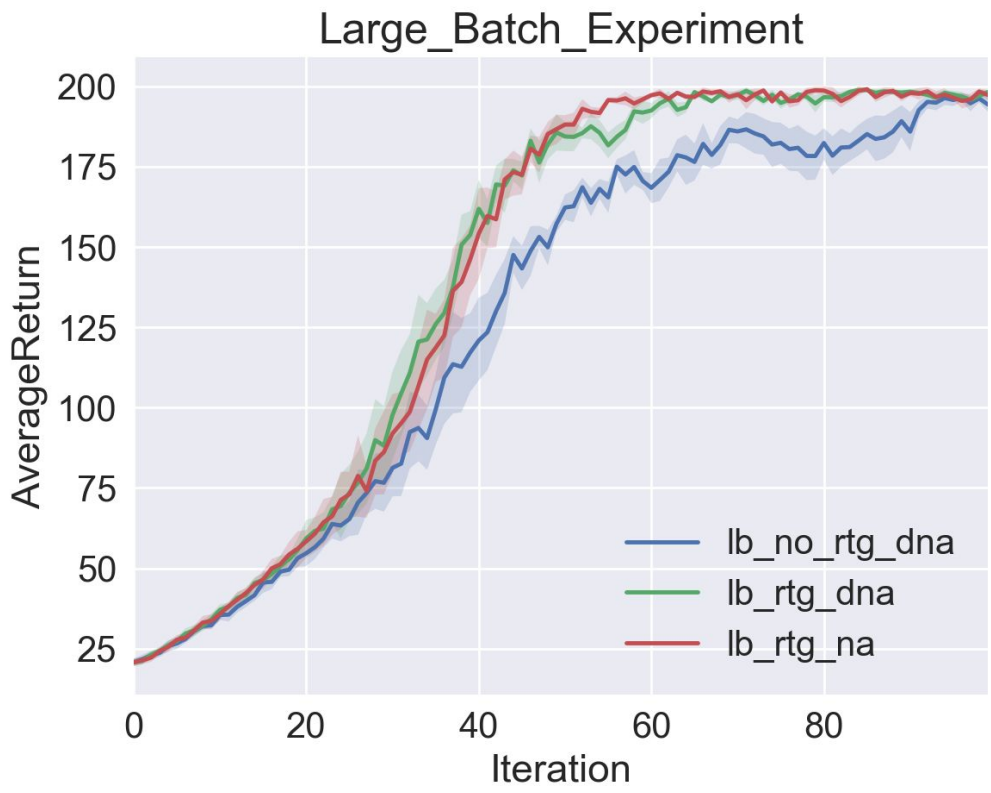
Section 4.2.1


Small_Batch_Experiment
AverageReturn
Iteration
sb_no_rtg_dna
sb_rtg_dna
sb_rtg_na
200
175
150
125
100
75
50
25
0
20
40
60
80

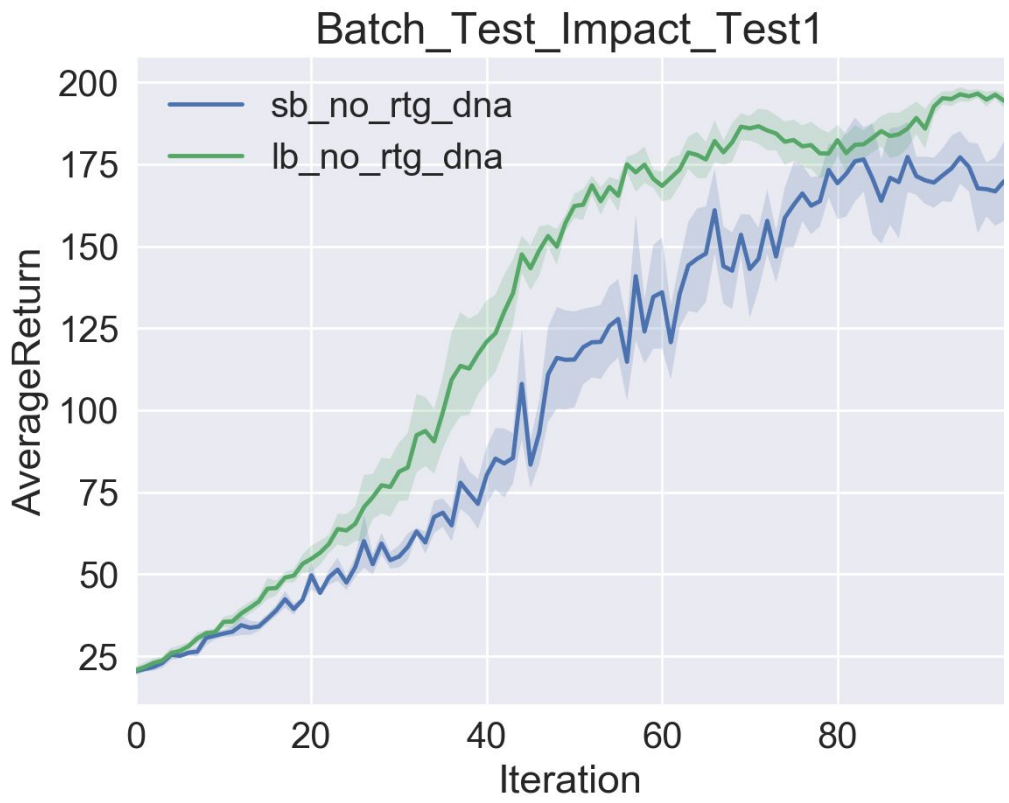

Large_Batch_Experiment
AverageReturn
Iteration
lb_no_rtg_dna
lb_rtg_dna
lb_rtg_na
200
175
150
125
100
75
50
25
0
20
40
60
80

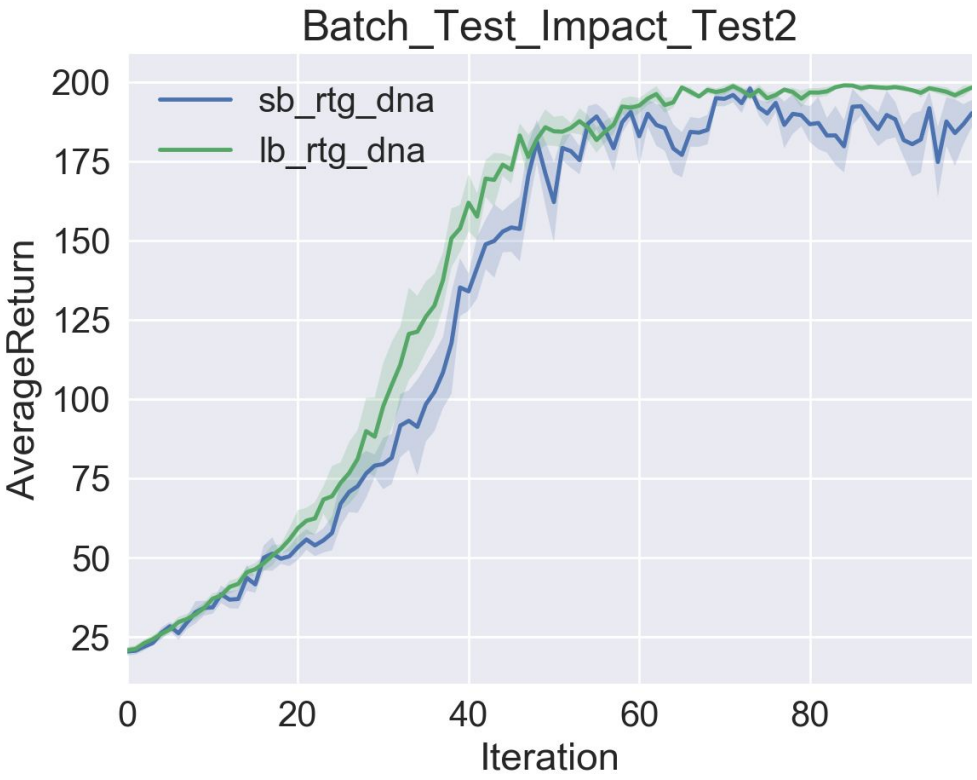
1. Which gradient estimator has better performance without advantage centering, the trajectory-centric one, or the one using reward-to-go?
Answer: Definitely the one using reward-to-go. The agent learn much faster in reward-to-go mode.

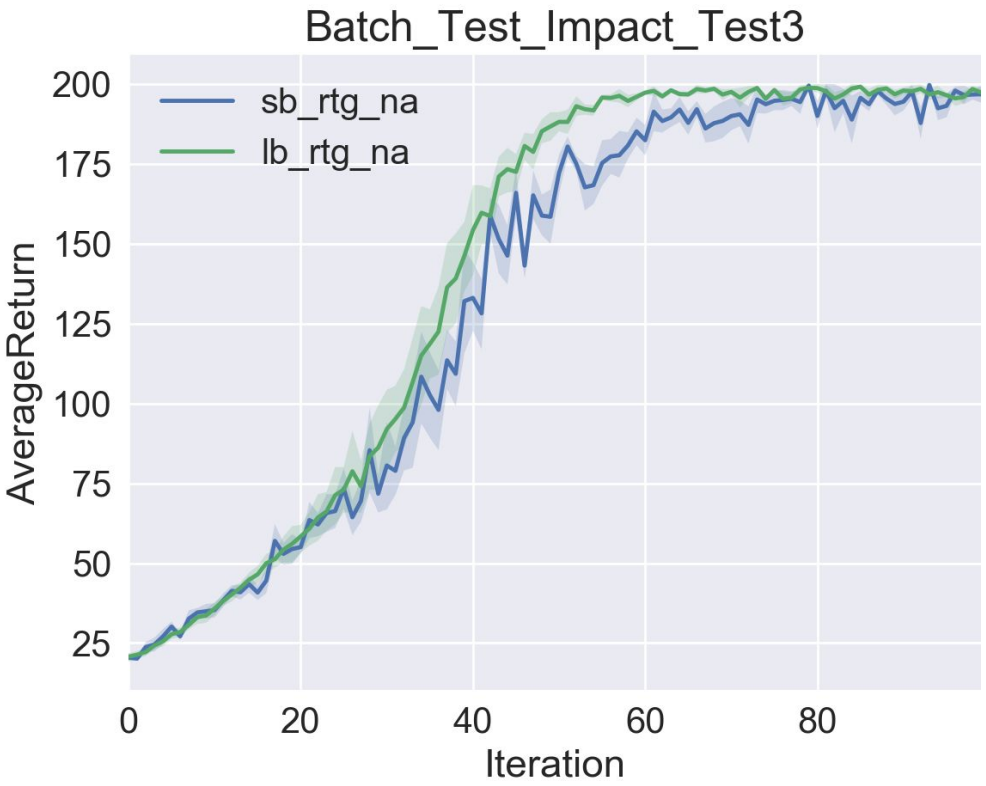
2. Did advantage centering help?
Answer: No. At least from two graphs above, we cannot see any performance improvement from advantage centering.

3. Describe what you expect from the math-do the empirical results match the theory?
Answer: We expect a lower variance in reward-to-go mode. As a result, the agent should be able to learn faster, which is indeed the case. As for advantage normalization, while the math says that it can lower the variance of the estimator, rescale the learning rate while leave the policy gradient unchanged in expectation, we can not observe any proofs for these theories.

4. Did the batch size make an impact?
Yes, as can be seen from the three graphs below, larger batch size generally helps the agent to learn faster, but the marginal benefit is decreasing as we use more techniques to lower the variance of the policy gradient.




Batch_Test_Impact_Test2
sb_rtg_dna
lb_rtg_dna
AverageReturn
Iteration
200
175
150
125
100
75
50
25
0
20
40
60
80


Batch_Test_Impact_Test3
sb_rtg_na
lb_rtg_na
AverageReturn
Iteration
200
175
150
125
100
75
50
25
0
20
40
60
80

Command Line Configuration:

1. python train_pg.py CartPole-v0 -n 100 -b 1000 -e 5 -dna --exp_name sb_no_rtg_dna
2. python train_pg.py CartPole-v0 -n 100 -b 1000 -e 5 -rtg -dna --exp_name sb_rtg_dna
3. python train_pg.py CartPole-v0 -n 100 -b 1000 -e 5 -rtg --exp_name sb_rtg_na
4. python train_pg.py CartPole-v0 -n 100 -b 5000 -e 5 -dna --exp_name lb_no_rtg_dna
5. python train_pg.py CartPole-v0 -n 100 -b 5000 -e 5 -rtg -dna --exp_name lb_rtg_dna
6. python train_pg.py CartPole-v0 -n 100 -b 5000 -e 5 -rtg --exp_name lb_rtg_na

Default settings used:

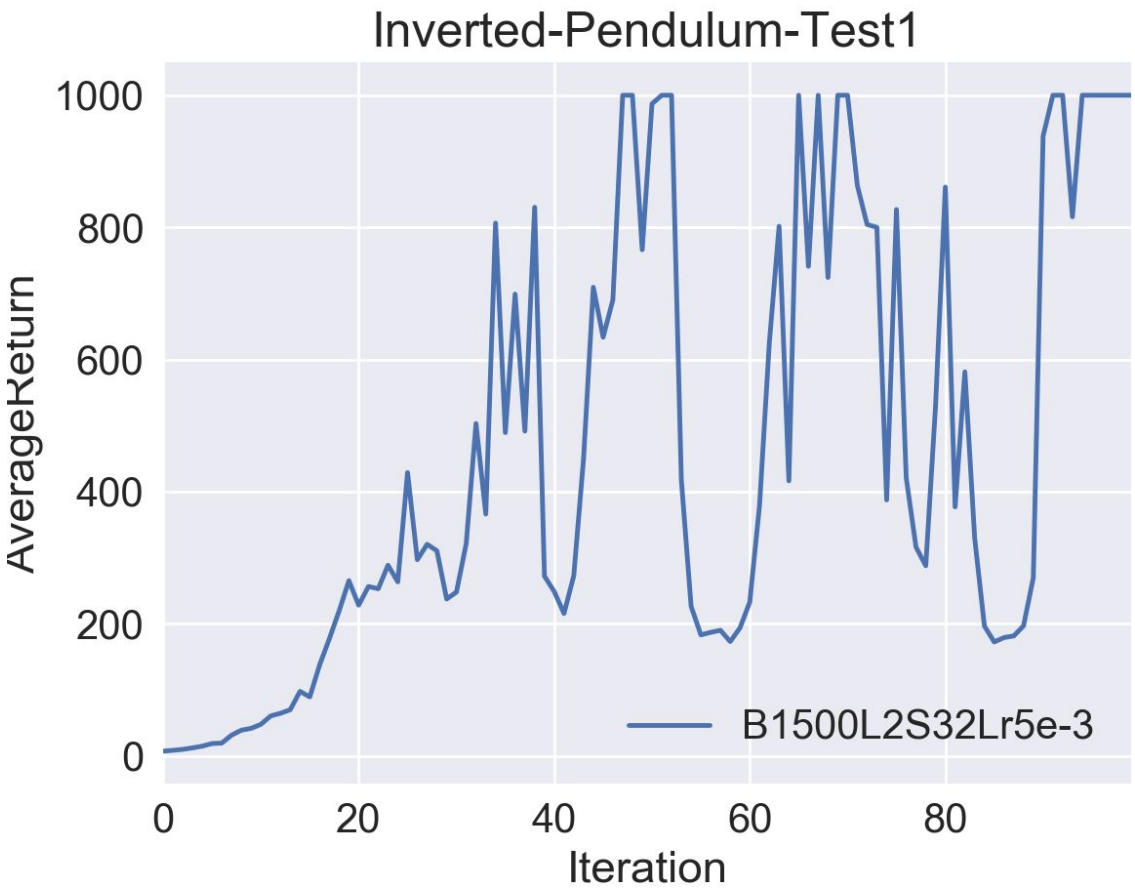
1. Discount factor: 1
2. Learning rate: 5e-3
3. Random seed: 1
4. Number of layers of the policy network: 1
5. The hidden dimension of the perceptron layer in the policy network: 32
6. Policy network activation function: tanh
7. Policy network output activation function: None
8. Optimization: the default Adam optimizer in Pytorch

The layers of the policy network (in order) generated from the settings above is then:

1. nn.Linear(input_dim, hidden_dim=32)
2. nn.functional.tanh
3. nn.Linear(hidden_dim=32, output_dim)

Then the output, seen as logits, generate the categorical distribution from which we will sample our next action.

4.2.2



Command Line Configuration:

1. python train_pg.py InvertedPendulum-v2 -n 100 -b 1500 -l 2 -s 32 -rtg -lr 5e-3 --exp_name B1500L2S32Lr5e-3 --seed 3

Settings:

1. Discount factor: 1
2. Learning rate: 5e-3
3. Random seed: 3
4. Number of layers of the policy network: 2
5. The hidden dimension of the perceptron layer in the policy network: 32
6. Policy network activation function: tanh
7. Policy network output activation function: None
8. Optimization: the default Adam optimizer in Pytorch

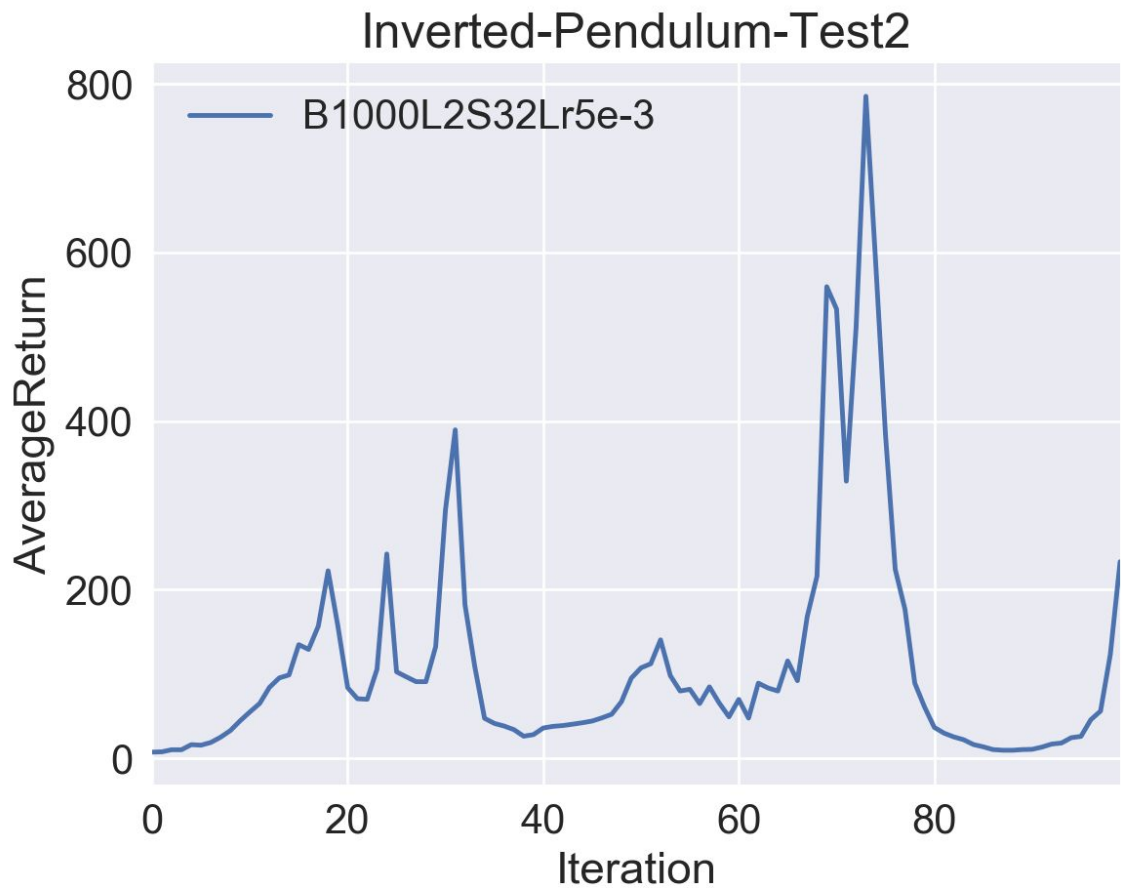
The layers of the policy network (in order) generated from the settings above is then:

1. nn.Linear(input_dim, hidden_dim=32)

2. nn.functional.tanh
3. nn.Linear(hidden_dim=32, hiddem_dim=32)
4. nn.functional.tanh
5. Header for mean after 4: nn.Linear(hidden_dim=32, output_dim)
6. Header for std after 4: nn.Linear(hidden_dim=32, output_dim)

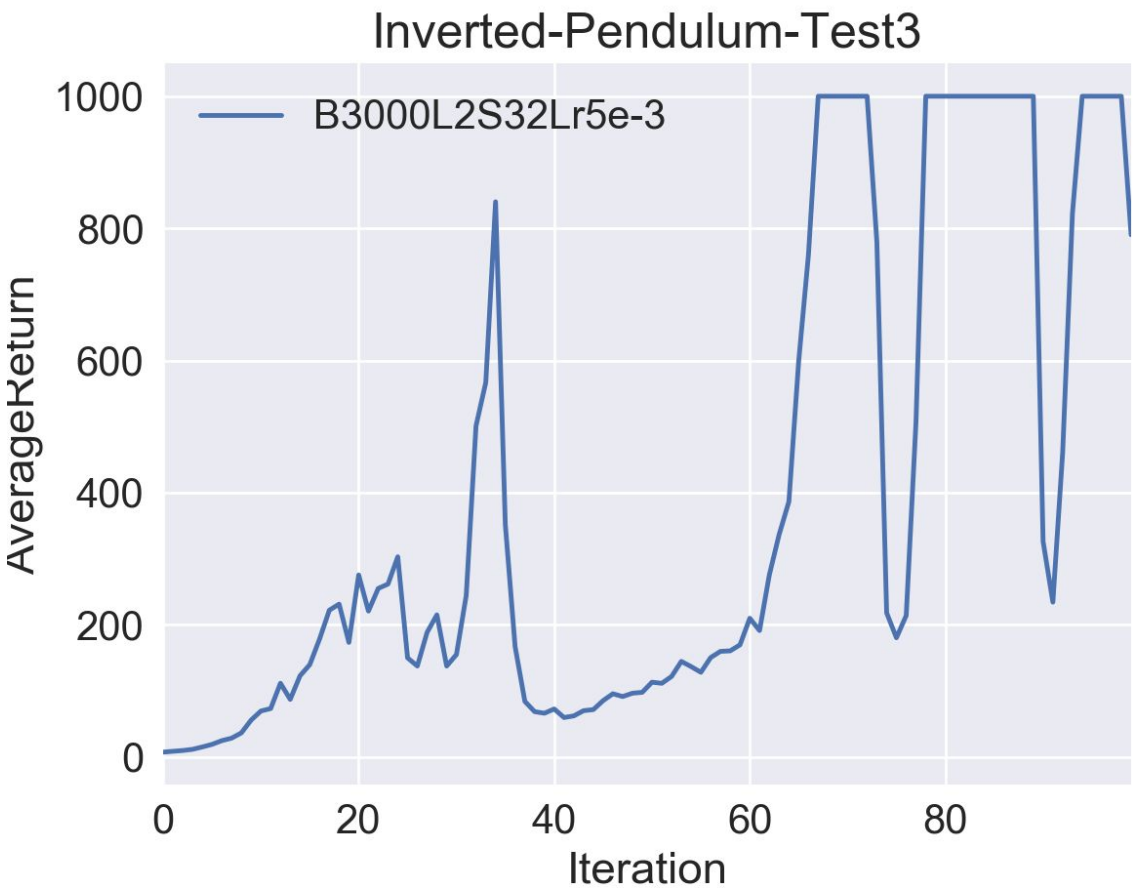
Then the output will generate the normal distribution from which we will sample our next action. The reward-to-go gradient estimator is used in the experiment, based on my experience in the previous section.

The smallest possible batch size that I found is 1500. Batch size below 1500 cannot solve the problem, at least according to my experience, while batch size larger than 1500 definitely helps, since it provides a better approximate of the gradient. The two graphs below show in order the learning curves I gained with batch size=1000 and 3000.



Command Line Configuration:

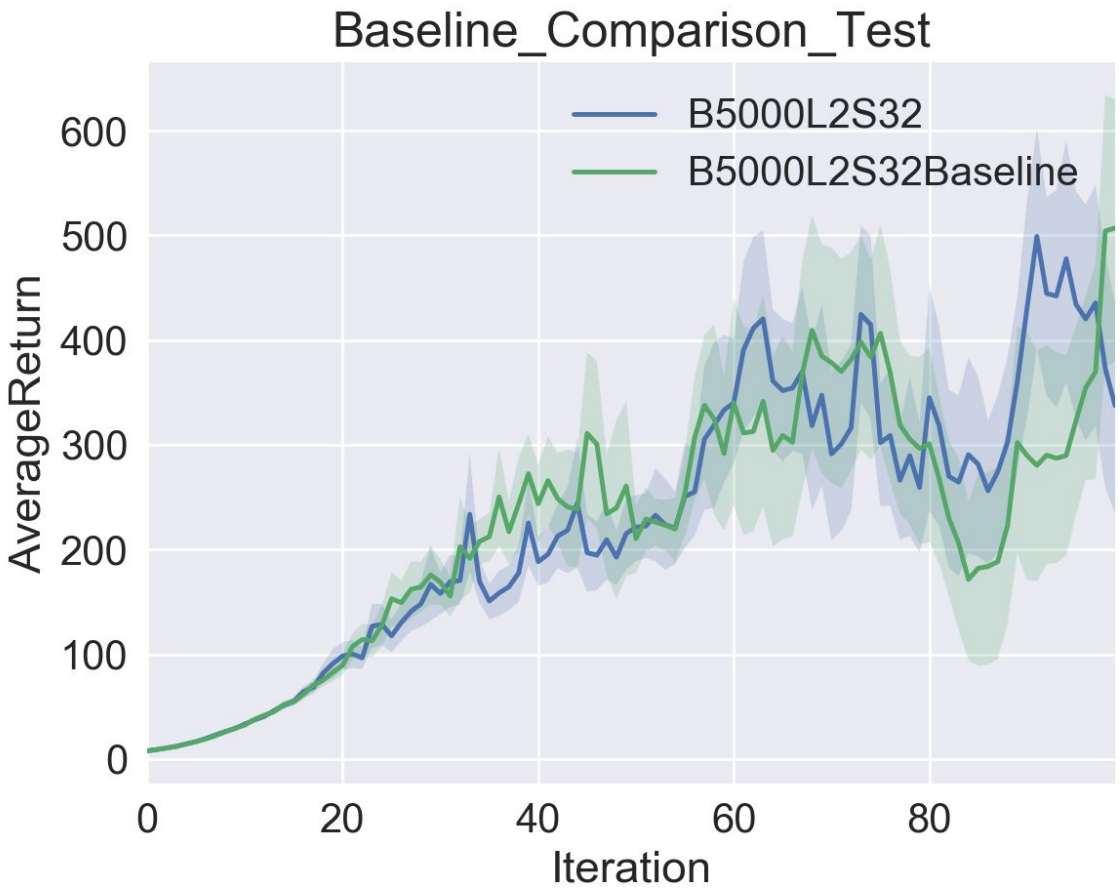
1. python train_pg.py InvertedPendulum-v2 -n 100 -b 1000 -l 2 -s 32 -rtg -lr 5e-3 --exp_name B1000L2S32Lr5e-3 --seed 3



Command Line Configuration:

1. python train_pg.py InvertedPendulum-v2 -n 100 -b 3000 -l 2 -s 32 -rtg -lr 5e-3 --exp_name B3000L2S32Lr5e-3 --seed 3

5.1



As can be seen from the graph above, the green line, the learning curve with a neural network baseline function and advantage normalization, does not outperform the blue line, the learning curve without a neural network baseline function.

10 fair trials with random seeds (1,11,21,31,41,51,61,71,81,91) are conducted for the comparison test and the performances are averaged. Neither of the networks perform well consistently in all trials, though they perform relatively well in most of them. Hence, after averaging over 10 trials, both learning curves do not reach the optimal reward value 1000, and have large standard deviation.

To conclude, the experiment does not support that a neural network baseline function can help lower the variance and thus resulting in a better performance, at least not in the inverted pendulum environment.

Command Line Configuration:

1. python train_pg.py InvertedPendulum-v2 -n 100 -b 5000 -l 2 -s 32 -rtg -lr 4e-3 -e 10 --exp_name B5000L2S32

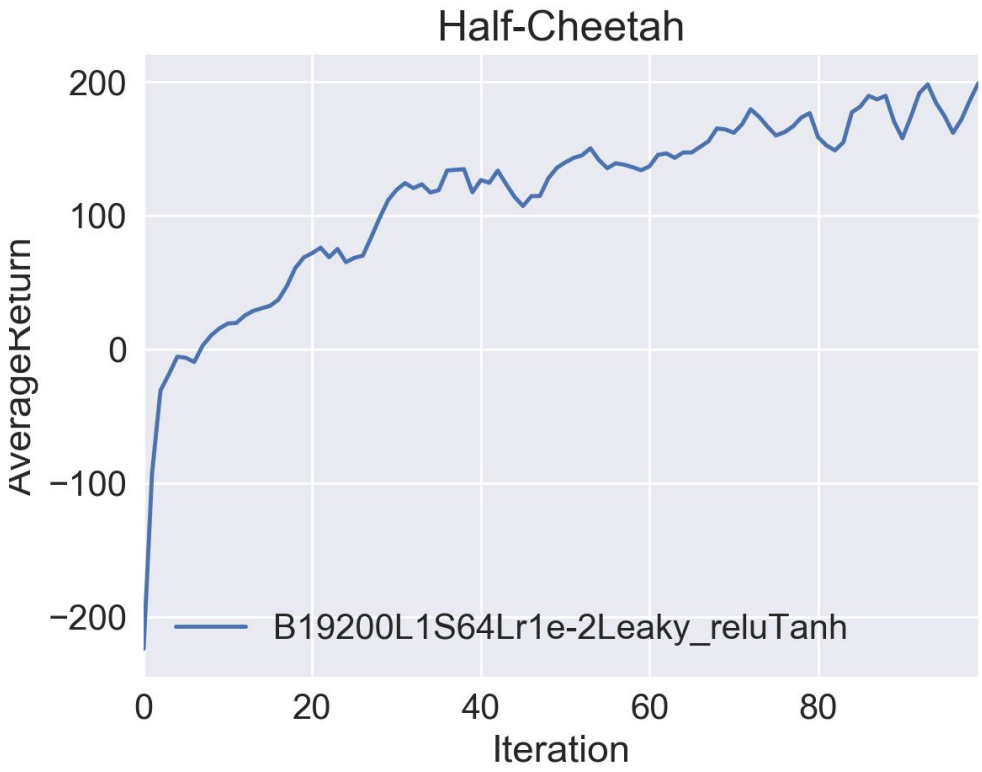
2. python train_pg.py InvertedPendulum-v2 -n 100 -b 5000 -l 2 -s 32 -rtg -lr 4e-3 -e 10 --nn_baseline --exp_name B5000L2S32Baseline

Settings:

1. Discount factor: 1
2. Learning rate: 4e-3
3. Random seed: 1, 11, 21, 31, 41, 51, 61, 71, 81, 91
4. Number of layers of the policy network: 2
5. The hidden dimension of the perceptron layer in the policy network: 32
6. Policy network activation function: tanh
7. Policy network output activation function: None
8. Optimization: the default Adam optimizer in Pytorch
9. Batch size: 5000

Network Architecture: see the previous section 4.2.2

6.1



Command Line Configuration:

1. python train_pg.py HalfCheetah-v2 -ep 150 --discount 0.9 -n 100 -b 19200 -l 1 -s 64 -rtg -lr 0.01 --activation leaky_relu --output_activation tanh --exp_name B19200L1S64Lr1e-2Leaky_reluTanh

Settings:

1. Discount factor: 0.9
2. Learning rate: 0.01
3. Random seed: 1
4. Number of layers of the policy network: 1
5. The hidden dimension of the perceptron layer in the policy network: 64
6. Policy network activation function: Leaky Relu function with Pytorch default parameters
7. Policy network output activation function: tanh
8. Optimization: the default Adam optimizer in Pytorch
9. Batch size: 19200=150*128
10. Episode length: 150

The layers of the policy network (in order) generated from the settings above is then:

1. nn.Linear(input_dim, hidden_dim=64)
2. Leaky Relu

3. Header for mean after 2: nn.Linear(hidden_dim=64, output_dim)
4. Header for mean after 3: nn.functional.tanh
5. Header for std after 2: nn.Linear(hidden_dim=64, output_dim)
6. Header for std after 5: nn.functional.tanh

Then the output will generate the normal distribution from which we will sample our next action.

Note:

1. We apply output activation tanh function mainly because the Half-Cheetah environment punishes networks that produces large action vectors (see line 15 of the source code here).
2. We use Leaky Relu instead of tanh as our activation function to avoid gradient vanishing problem.
3. We do not use a neural network baseline. In all experiments I conducted, there were no improvements observed when a neural network baseline was used and this setting slowed down the training process. One possible reason is that I used the same learning rate for both the actor network and critic network, but then find the optimal learning rate for each network will definitely take a much longer hyper-parameter testing time. Hence, I decide it discard it completely for this section.
4. If we are allowed to continue our training after the 100th iteration, as the graph below shows, the network can improve further to have rewards around 250, by the end of the 200th iteration. This suggests that the 100 training iterations requirement might be a little too tight for us to train a good network.