2.4 Basic Q-Learning

Deliverables:

- Submit your logs of `CartPole-v1`, and a plot with environment steps on the x-axis and eval return on the y-axis.
- Run DQN with three different seeds on `LunarLander-v2`:

```
python cs285/scripts/run_hw3_dqn.py -cfg experiments/dqn/lunarlander.yaml --seed 1
python cs285/scripts/run_hw3_dqn.py -cfg experiments/dqn/lunarlander.yaml --seed 2
python cs285/scripts/run_hw3_dqn.py -cfg experiments/dqn/lunarlander.yaml --seed 3
```

  Your code may not reach high return (200) on Lunar Lander yet; this is okay! Your returns may go up for a while and then collapse in some or all of the seeds.

- Run DQN on `CartPole-v1`, but change the `learning rate` to 0.05 (you can change this in the YAML config file). What happens to (a) the predicted Q-values, and (b) the critic error? Can you relate this to any topics from class or the analysis section of this homework?

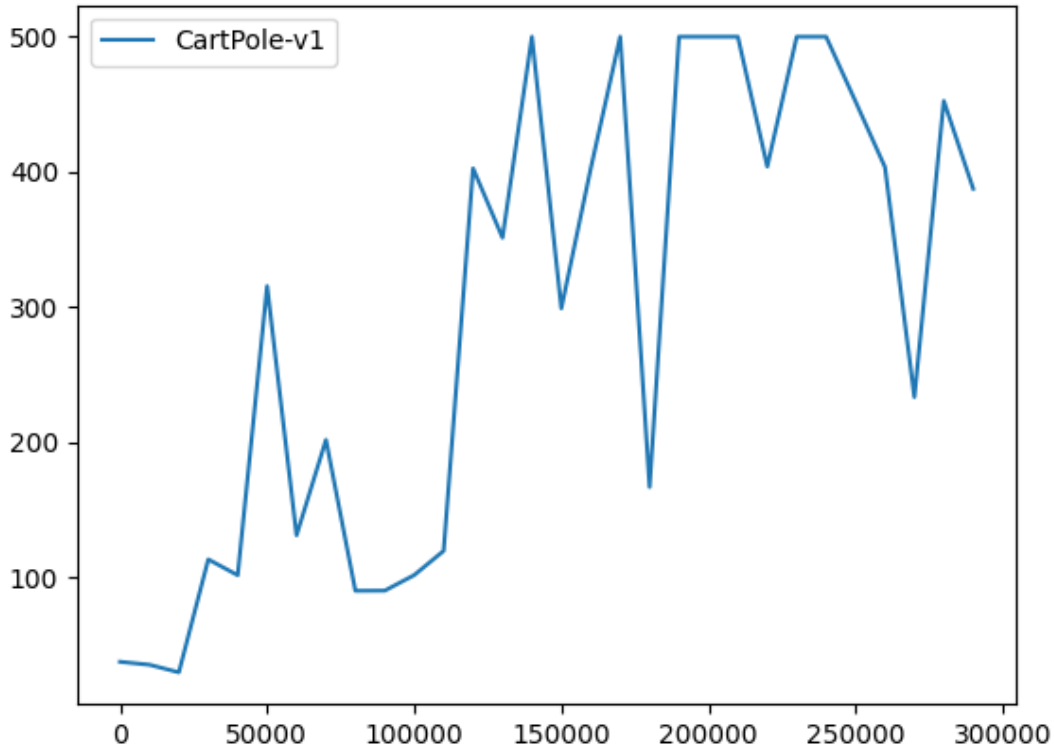

Figure 1: Estimated Q-Values

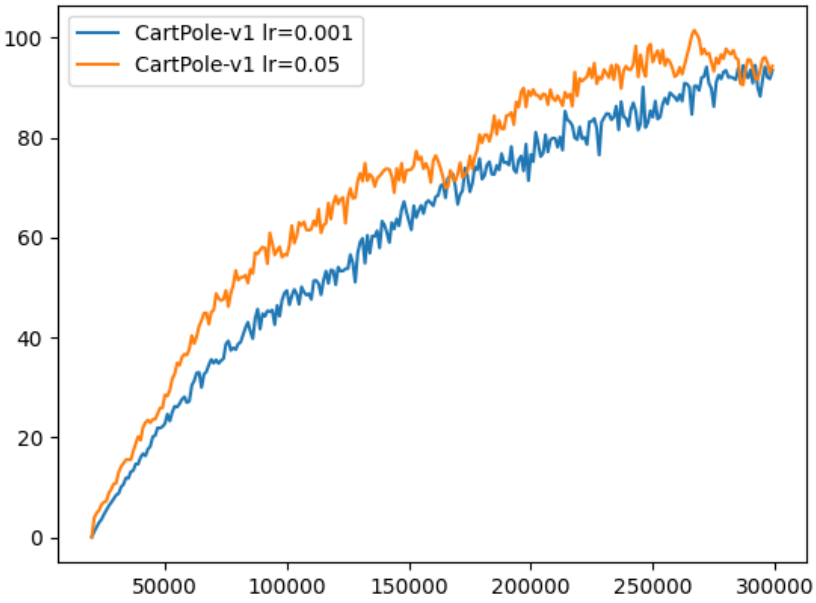

Figure 2 Comparison of estimated Q-values for different learning rates

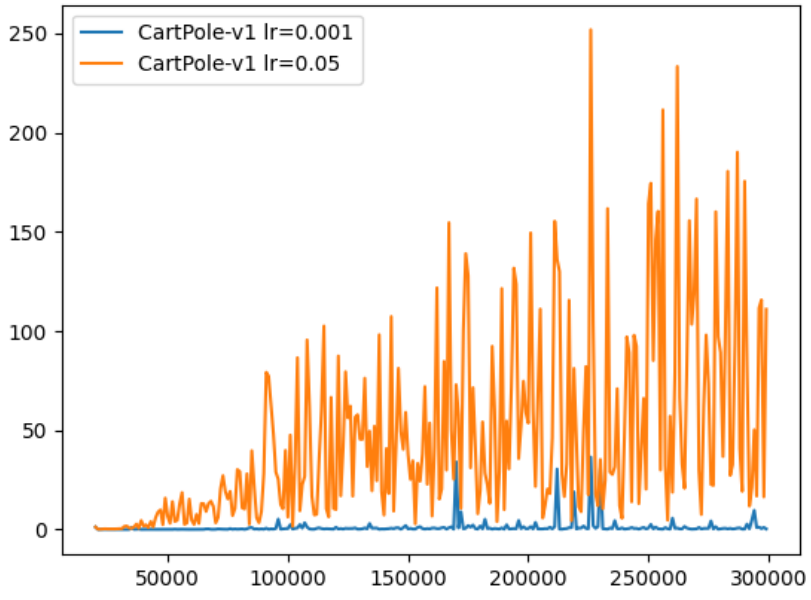

Figure 3 Comparison of critic_loss for different learning rates

Related topic: “Chasing the own tail”/unstable training with moving targets

2.5 Double Q-Learning

Deliverables:

- Run three more seeds of the lunar lander problem:

```
python cs285/scripts/run_hw3_dqn.py -cfg experiments/dqn/lunarlander_doubleq.yaml --
    seed 1
python cs285/scripts/run_hw3_dqn.py -cfg experiments/dqn/lunarlander_doubleq.yaml --
    seed 2
python cs285/scripts/run_hw3_dqn.py -cfg experiments/dqn/lunarlander_doubleq.yaml --
    seed 3
```

  You should expect a return of **200** by the end of training, and it should be fairly stable compared to your policy gradient methods from HW2.

  Plot returns from these three seeds in red, and the "vanilla" DQN results in blue, on the same set of axes. Compare the two, and describe in your own words what might cause this difference.

- Run your DQN implementation on the `MsPacman-v0` problem. Our default configuration will use double-Q learning by default. You are welcome to tune hyperparameters to get it to work better, but the default parameters should work (so if they don't, you likely have a bug in your implementation). Your implementation should receive a score of around **1500** by the end of training (1 million steps. **This problem will take about 3 hours with a GPU, or 6 hours without, so start early!**

```
python cs285/scripts/run_hw3_dqn.py -cfg experiments/dqn/mspacman.yaml
```

- Plot the average training return (`train_return`) and eval return (`eval_return`) on the same axes. You may notice that they look very different early in training! Explain the difference.

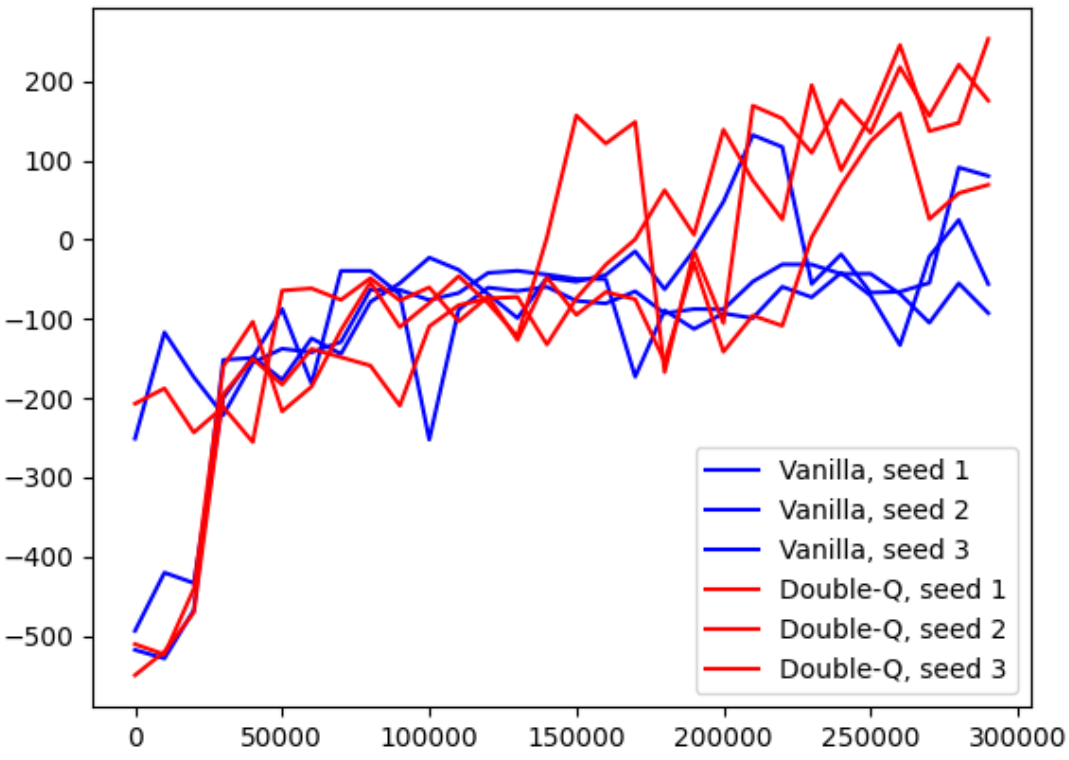

Figure 4 Comparison of Vanilla and Double-Q approach for different random seeds

Double-Q leads to a more stable and promising training process.

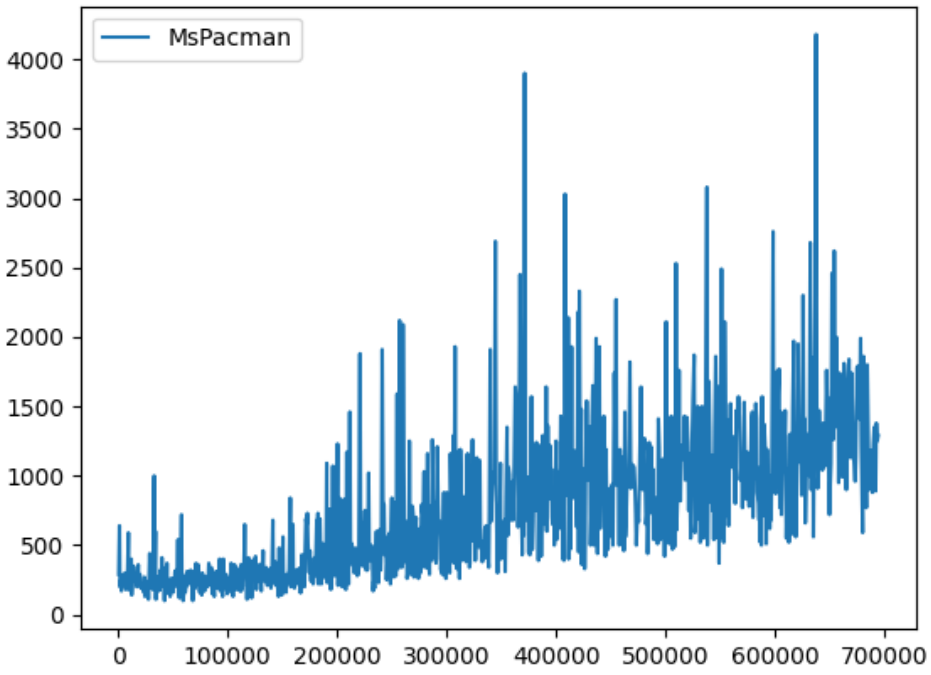

Figure 5 MsPacman train_return results

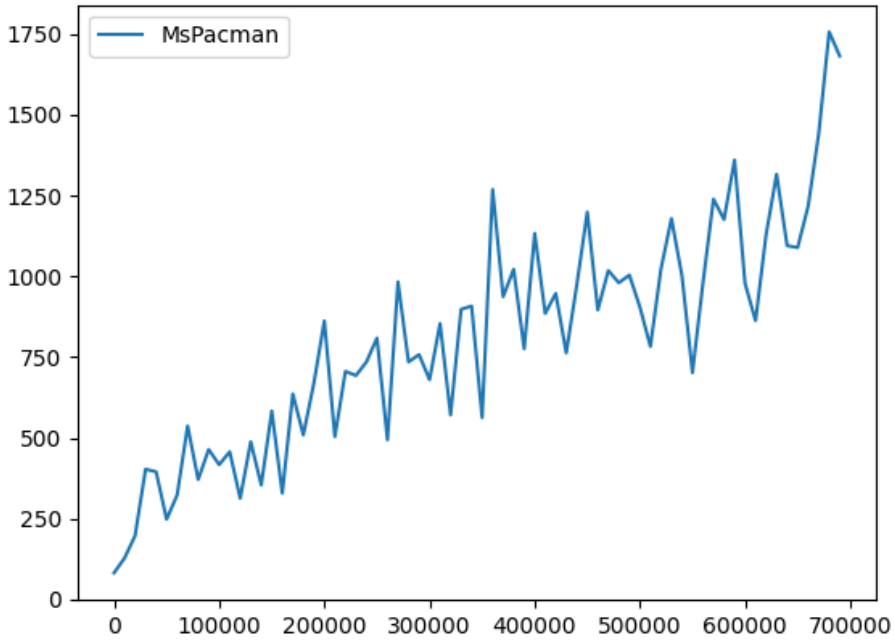

Figure 6 MsPacman eval_return results

2.6 Experimenting with Hyperparameters

Now let's analyze the sensitivity of Q-learning to hyperparameters. Choose one hyperparameter of your choice and run at least three other settings of this hyperparameter, in addition to the one used in Question 1, and plot all four values on the same graph. Your choice what you experiment with, but you should explain why you chose this hyperparameter in the caption. Create four config files in `experiments/dqn/hyperparameters`, and look in `cs285/env_configs/basic_dqn_config.py` to see which hyperparameters you're able to change. You can use any of the base YAML files as a reference.

Hyperparameter options could include:

- Learning rate
- Network architecture
- Exploration schedule (or, if you'd like, you can implement an alternative to ϵ-greedy)

3 Continous Actions with Actor Critic

Deliverables

- Train an agent on `HalfCheetah-v4` using the provided config (`halfcheetah_reinforce1.yaml`). Note that this configuration uses only one sampled action per training example.
- Train another agent with `halfcheetah_reinforce_10.yaml`. This configuration takes many samples from the actor for computing the REINFORCE gradient (we'll call this REINFORCE-10, and the single-sample version REINFORCE-1). Plot the results (evaluation return over time) on the same axes as the single-sample REINFORCE. Compare and explain your results.

Deliverables:

- Train (once again) on `HalfCheetah-v4` with `halfcheetah_reparametrize.yaml`. Plot results for all three gradient estimators (REINFORCE-1, REINFORCE-10 samples, and REPARAMETRIZE) on the same set of axes, with number of environment steps on the x-axis and evaluation return on the y-axis.
- Train an agent for the `Humanoid-v4` environment with `humanoid_sac.yaml` and plot results.

3.1.5 Stabilizing Q-Values

Deliverables:

- Run single-Q, double-Q, and clipped double-Q on `Hopper-v4` using the corresponding configuration files. Which one works best? Plot the logged `eval_return` from each of them as well as `q_values`. Discuss how these results relate to overestimation bias.
- Pick the best configuration (single-Q/double-Q/clipped double-Q, or REDQ if you implement it) and run it on `Humanoid-v4` using `humanoid.yaml` (edit the config to use the best option). You can truncate it after 500K environment steps. If you got results from the humanoid environment in the last homework, plot them together with environment steps on the x-axis and evaluation return on the y-axis. Otherwise, we will provide a humanoid log file that you can use for comparison. How do the off-policy and on-policy algorithms compare in terms of sample efficiency? *Note: if you'd like to run training to completion (5M steps), you should get a proper, walking humanoid! You can run with videos enabled by using* `--nvid 1`. *If you run with videos, you can strip videos from the logs for submission with this script.*

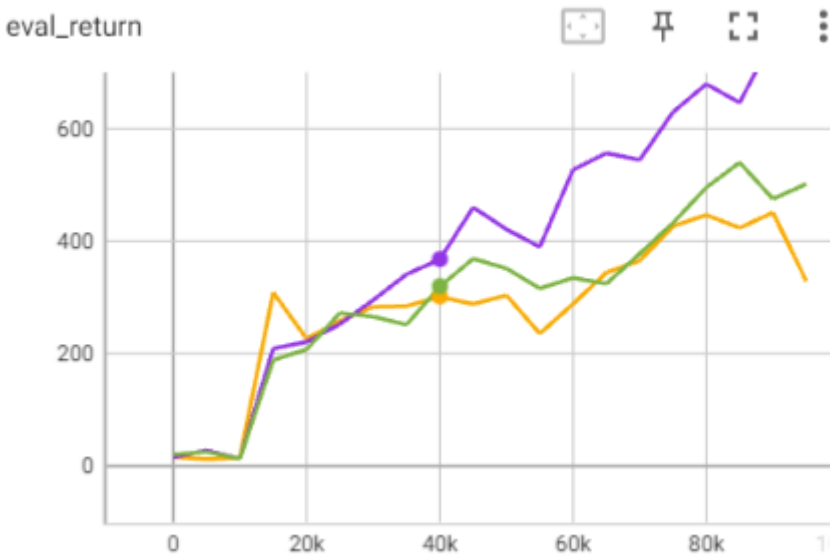

Figure 7 Comparison of different Methods. Orange = Single Q-Value, Green = Double and Purple = Min-Clip