CMPUT 652, Fall 2019 - Assignment #2

October 15, 2019

Version: 1.1. October 16, 2019.

Due: October 29, 23:59 MST

This assignment consists of two parts. The first are written questions which are related to the course lectures. The second part is a programming assignment. You will need to submit both code and a write up. Please provide these as a single archive. Submit to both armahmood@ualberta.ca and sherstan@ualberta.ca.

1 Written Questions

Total 30 points. Complete within the given space using LaTeX or handwritten notes. Show the derivations or the steps for obtaining partial points. On the other hand, mistakes, missteps or bad reasoning behind a correct answer will result in points deducted.

1.1 Consider a finite MDP, where a state is represented by a d-dimensional feature vector $\mathbf{x}(s)$, and the feature vectors for all N states form a $N \times d$ feature matrix $\mathbf{X}$ with rank d. The Mean Squared Value Error (MSVE) objective is $\|\mathbf{v}_\pi - \mathbf{X}\mathbf{w}\|^2_{\mathbf{D}_\pi}$, where $\mathbf{v}_\pi$ is an N-dimensional vector, $[\mathbf{v}_\pi]_s = v_\pi(s)$ is the value of state s induced by policy π, and $\mathbf{D}_\pi$ is a $N \times N$ diagonal matrix with the steady-state probabilities $d_\pi(s)$ induced by π along the diagonal. Show that the solution to this objective is $\mathbf{w}^*_{\text{VE}} = (\mathbf{A}_{\text{VE}})^{-1}\mathbf{b}_{\text{VE}}$, where $\mathbf{A}_{\text{VE}} = \mathbf{X}^\top \mathbf{D}_\pi \mathbf{X}$ and $\mathbf{b}_{\text{VE}} = \mathbf{X}^\top \mathbf{D}_\pi \mathbf{v}_\pi$ (15 points).

1.2 Consider a vector y that may not be in the linear span of the features, that is, $\mathrm{y} \neq \mathbf{X}\mathrm{w}$ for any $\mathrm{w} \in \mathbb{R}^d$. The projection matrix $\mathbf{\Pi}$ is such that $\mathbf{\Pi}\mathbf{y} = \mathbf{X}\mathrm{w}_y$, where $\mathrm{w}_y = \arg\min_{\mathbf{w}} \|\mathbf{y} - \mathbf{X}\mathrm{w}\|^2_{\mathbf{D}}$, that is, it is the matrix that linearly transforms or "projects" y to the span of features according to norm $\|\cdot\|^2_{\mathbf{D}}$. Here $\mathbf{D}$ is a diagonal matrix with positive diagonal elements. Show that $\mathbf{\Pi} = \mathbf{X}(\mathbf{X}^\top\mathbf{D}\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{D}$, $\mathbf{\Pi}\mathbf{X} = \mathbf{X}$ and $\mathbf{X}^\top\mathbf{D}\mathbf{\Pi} = \mathbf{X}^\top\mathbf{D}$ (15 points).

2 Programming

100 points. You will implement a basic policy-gradient method: REINFORCE with a Baseline. The purpose is to gain experience implementing PG methods in PyTorch. Additionally, the questions are meant to help you develop the tools you will need for performing empirical analysis.

See Sections 13.3 and 13.4 of Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction* (2nd ed.). The MIT Press for information on the REINFORCE algorithm.

```
REINFORCE with Baseline (episodic), for estimating $\pi_{\boldsymbol\theta} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol\theta)$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Algorithm parameters: step sizes $\alpha^{\boldsymbol\theta} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol\theta \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
    Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol\theta)$
    Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
        $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$                                   $(G_t)$
        $\delta \leftarrow G - \hat{v}(S_t,\mathbf{w})$
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S_t,\mathbf{w})$
        $\boldsymbol\theta \leftarrow \boldsymbol\theta + \alpha^{\boldsymbol\theta} \gamma^t \delta \nabla \ln \pi(A_t|S_t,\boldsymbol\theta)$
```

Figure 1: (Sutton and Barto, 2018)

The episodic REINFORCE algorithm collects entire episodes of trajectories before updating policy parameters. It is a Monte Carlo method based on Stochastic Gradient Descent. In the its returns can have high variance, making learning difficult. By subtracting off a baseline, the value estimate, the variance of the policy parameter updates can be reduced and learning can be improved.

Your task is to implement REINFORCE with a baseline to solve the classic Cart Pole task: https://gym.openai.com/envs/CartPole-v1/.

Note that the algorithm specified in the above collects an episode of transitions and then loops over each transition of the episode computing gradients and updating weights at each timestep. You could also implement the update as a batch where the errors are taken across the entire episode trajectory before gradients are computed. Feel free to implement either approach. However, for reporting losses please average losses over the whole episode.

A note on γ. First, γ is sometimes approached in two different ways. One way claims that γ is part of the problem definition and the other treats γ as part of the solution. We will take the second view here. We will evaluate your algorithm based on the total undiscounted reward received over an entire episode. You are free to adjust γ as you need, but a good place to start would be to simply set $\gamma = 1$ and then try other values. Further, the last line of the equation in Figure 1 has the additional term γ^t. This term was not always included in the algorithm and you will find implementations online which leave out this term. I expect it will impact the values of γ for which your algorithm performs well. You are free to include it or leave it out, but you should be clear which version you implemented in your pseudocode.

2.1 Setup

Create a python virtualenv (feel free to use another dependency manager if you like) for running your code. We will use python3.6+. A requirements.txt file is included specifying all the packages which you will require for your virtualenv. Do not make your code dependent on any additional packages as they will not be part of the environment on which I test.

Create your virtualenv as follows (adjust paths as required):

```
# create a virtualenv called assign2
virtualenv --python=python3.6 assign2
# you will need to activate this environment before running your script.
```

```
source assign2/bin/activate
pip install -r requirements.txt
```

You have been provided with two skeleton files: main.py and network.py. Edit these files to implement your code.

2.2 Deliverables

Note, that I am not evaluating your code based on whether or not it can beat some baseline or the performance of your classmate's algorithm. I am looking for code that works. Your agent should be able to fairly frequently achieve a perfect score of 200 by at least 10k episodes. I will be lenient in grading, but I do expect that everyone put in the effort and does their own work - of course you are free to consult with one another. The key to full marks is to simply implement the algorithm, achieve reasonable performance, and put in sufficient effort in answering the questions and formatting your plots where possible. If everything looks good from a high level, I won't go looking for problems.

The plots provided below should only be taken as examples, not optimal.

In the following code we have chosen to plot returns as a function of the episode. We do this for simplicity. A fairer comparison would be to compare based on the number of observations made. In the next assignment we will compare in this way.

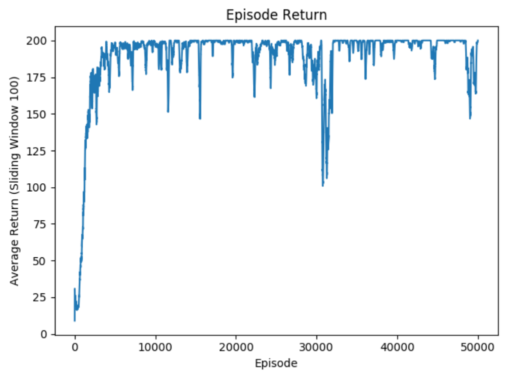
2.2.1 Running Code (50 points)

Submit code for training a policy using REINFORCE with a baseline on CartPole. Use the included main.py as the skeleton for your code base. You are free to make changes to this file, such as adding additional imports or command line arguments, but I expect it to still run with just

```
python main.py --episodes 10000
```

Fill in network.py with your NN architecture, but do not change the function definitions.

After a run the script will output a graph like this:



Please provide this plot.

In LaTeXwrite the pseudocode for the algorithm that you actually implemented. Be sure to include any implementation details, e.g. gradient clipping.

2.2.2 Saved Policy (5 points)

Train your policy to 50k+ episodes. Save your policy by calling

```
torch.save(network.state_dict(), <path>.pkl)
```

I have a script which will load your policy and evaluate it by averaging over 100 episodes. Your average should be close to 200 (say 190+).

2.2.3 Plot Return vs. Episode (15 points)

Perform 30 different runs for your algorithm (each run should be a different random initialization). Each run should be 5000 episodes in length. For each run you will need to save the list of episode returns. Using this data you will produce two different plots looking at the mean performance across different runs. Plots should be labeled and legible - they should look nice and be something you would include in a paper submission.

1. From your data set sample 3 runs and take the mean across the runs. Do this 10 times, each time resampling from your set of 30 runs. Plot each of these means together on a single plot. To be clear the x-axis will be the episode number.
2. From your dataset plot means for the following: 3 runs, 10 runs, 30 runs. Plot these all on the same plot. Label the lines in some way.

   Question: There are different statistics which might be reported for each mean: standard deviation, max-min, standard error. Which of these would you use? When? Why? In the previous plot which would you use to give a comparison between the series? Include your answer to these questions.

   Add this statistic to the series using `ax.fill_between` and setting the alpha to some low value (You only need to include one plot containing the means and the selected statistic).

2.2.4 Algorithm Comparison: (15 points)

Choose some algorithm variant to compare against your baseline implementation. Create a plot showing the performance for each averaged over 30 runs. Plot the standard error for each mean using shading as described in 2.3. While it would be normal to do sweeps over parameters for each variant being compared, you may skip that here.

There are many variants you could look at:

- Updating for each timestep vs. batch.
- Another approach to dealing with variance is to *whiten* the returns instead of using a baseline. That is scaling and shifting the returns such that they have a mean of 0 and variance of 1.
- Use the lambda-return.
- In the weight update step shown in Figure 1,

  $$\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \gamma^t \delta \nabla \ln \pi(A_t | S_t, \mathbf{w}),$$

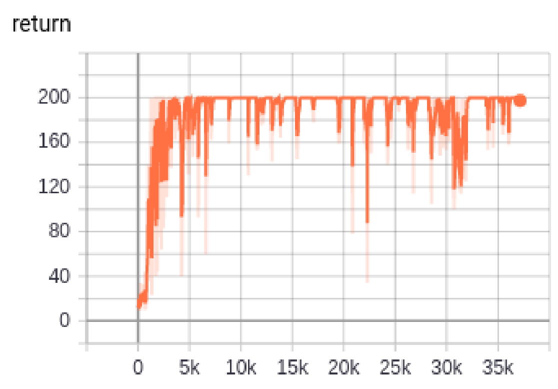
  there is a term γ^t. The need for this correction was not always recognized. Try with and without this term.
- Try different values of γ.
- Come up with your own.

You only have to choose one variant. Keep the values for the common hyperparameters the same between episodic REINFORCE and the variant you choose. Provide a short writeup, say no more than a page, which describes the comparison and explains your plot.

2.2.5 Tensorboard file: (15 points)

While I (Craig) find PyTorch much simpler to use than Tensorflow, the TensorBoard visualization tool is extremely helpful. Luckily, PyTorch now offers partial support for using TensorBoard. Use the tensorboard interface to plot:

1. **Episode return**. One entry for each episode. The total undiscounted reward for each episode.



2. **Objectives**. Your implementation should have two objectives: policy loss $\mathcal{L}_\pi$ and value function loss $\mathcal{L}_{\hat{v}}$. The average policy loss for an episode of length T is given by

$$\mathcal{L}_\pi = -\frac{1}{T}\sum_{t=0}^{T-1} \gamma^t (G_t - \hat{v}(S_t)) \ln(\pi(A_t|S_t)),$$

where the γ^t term may or may not be included depending on your implementation (As discussed in Section 2).

The value loss is simply given by average squared value error over the episode:

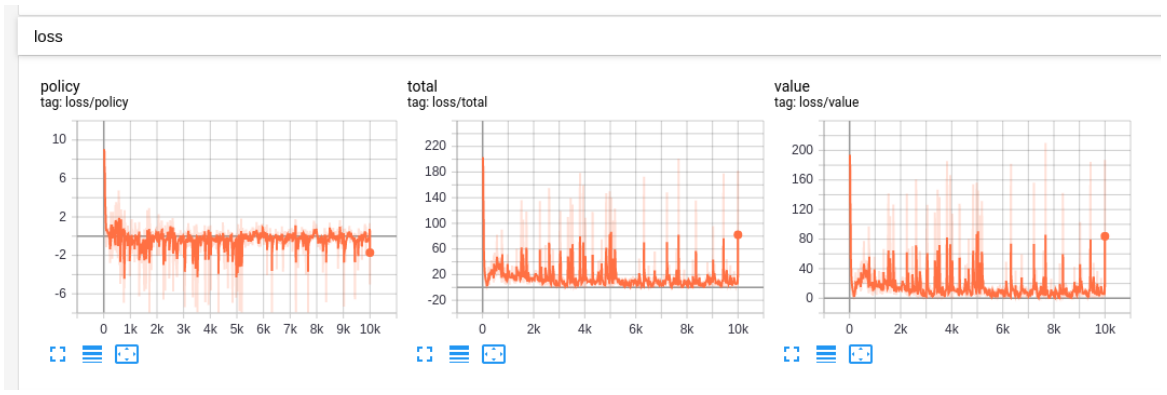
$$\mathcal{L}_{\hat{v}} = \frac{1}{T}\sum_{t=0}^{T-1} (G_t - \hat{v}(S_t))^2.$$

Plot each of these losses.

When training on multiple losses the losses are commonly combined and gradients are computed with respect to the combined loss:

$$\mathcal{L}_{Total} = \lambda_\pi \mathcal{L}_\pi + \lambda_{\hat{v}} \mathcal{L}_{\hat{v}},$$

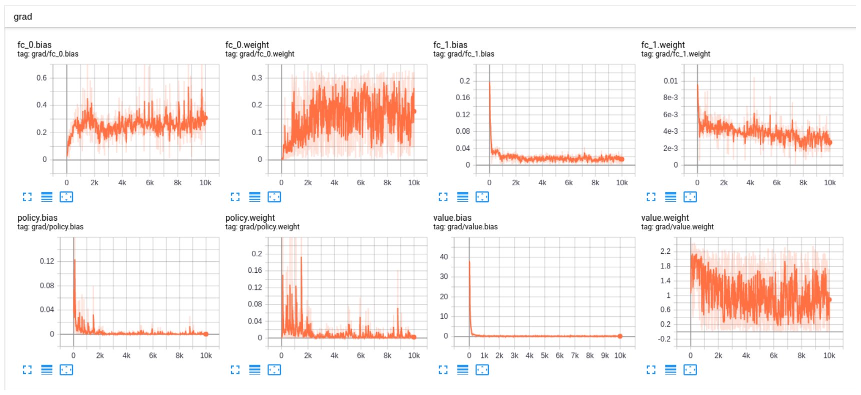
where each of the losses is weighted by a scalar λ. In my experience (Craig) in NN the performance of the network can be fairly sensitive to the values of λ and you might normally want to sweep over this parameter. However, for my own implementation I just set these values to 1. Plot the total loss as well.



3. **Gradients**. Observing gradients can be helpful in identifying problems in your policy network. Observing the scale of the gradient can allow you to identify potential problems such as exploding gradients. I also find it

helpful to test that different heads of my network are providing gradient to the rest of the network. For example, if you remove the policy loss from the gradient calculation do all the lower network layers still receive non-zero gradient? You can also use this technique to ensure that you are correctly blocking gradients when appropriate. You can also use the behavior of the gradients over time to identify potential problems. In my experience gradients may rise during earlier training and as training progresses they will eventually stabilize and start to decay if there is sufficient capacity in the network. The decay typically proceeds from the output layers towards the input layers. This is just my observations though, let me know if you disagree with this or if you observe different behaviors. Also, do let me know if you have any other suggestions for debugging network behavior.

For each layer of the network (after activation), plot the average squared gradient of that layer for each episode (so the average is taken across the layer nodes and the episode).



Submit your tensorboard events file.

2.3 Submission Checklist

- ☐ **2.2.1** Source Code and output plot from one run. Written algorithm.
- ☐ **2.2.2** One saved policy.
- ☐ **2.2.3** Two plots and 1 answered question.
- ☐ **2.2.4** A short writeup of your comparison. One plot showing the comparison.
- ☐ **2.2.5** One TensorBoard events file.

2.4 Common Problems to Watch For

- Forgetting to link your optimizer with your policy parameters.

```
torch.optim.Adam(network.parameters())
```

- Forgetting to `zero_grad` before calling `backward`.
- Broadcasting issues. If two vectors of different sizes are attempted to combine, e.g. [26] and [26,1], torch and numpy may happily combine these, but the results may not be what you expect. This can give incorrect results.
- Wrapping data in `torch.Tensor()` breaks the gradient flow. For example, if you want to convert a list of tensors to a tensor use `torch.stack()` instead.
- Forgetting to call `detach()` on a bootstrapped target. You don't, usually, want the gradient taken through your target. When we use a bootstrapped target, like in TD learning, we need to make sure to detach the gradient through the target.