# Instructions from the project using reinforcement learning to balance a bicycle

Jakob Wadman - wadmanj

## Learning rate

- A suggestion for how to decrease the learning rate is to start with a learning rate of $3 \cdot 10^{-4}$ and keep training until the progression of the average reward stagnates or crashes. Then load the best performing network and start training with a lower learning rate. Repeat this process for the learning rates $1 \cdot 10^{-4}$, $7 \cdot 10^{-5}$, $3 \cdot 10^{-5}$ and $1 \cdot 10^{-5}$ (if more fine-tuning is needed, you can try $9 \cdot 10^{-6}$, $8 \cdot 10^{-6}$, ..., i.e. don't go too low at once) until satisfactory results have been met (this is just a suggestion, could be more optimal choices on how to lower the learning rates).
- It could be tempting to just pick a small learning rate from the beginning to stop having to update the learning rate manually so often, but from our experience it is not worth it since a too small learning rate at a particular stage in the training can really prolong the training time substantially.
- In the beginning when the learning rate is large, e.g. $3 \cdot 10^{-4}$, then the average reward should increase steadily until the stagnation/crash, otherwise there is probably an error somewhere in the environment. However when it is small, e.g. $1 \cdot 10^{-5}$, you may have to be a bit patient before you see an increase from the previous best. Maybe wait around 300 updates before you give up and for example lower the learning rate or change something else.

## Discount factor

- The crashes seem to reduce when changing the discount factor $\gamma$ (in the notebook the parameter is called gamma), but can still happend. We almost always had $\gamma = 0.99$ during the project, it was just at the very end that we tried with $\gamma = 0.999$. With $\gamma = 0.999$ the crashes are not as common. However you still have to reduce the learning rate in the same manner since it stagnates if no crash happends, so in a way having the lower value of $\gamma$ can be more convenient since a crash makes it obvious when to change the learning rate. But the discount factor influences more things, see next bullet point. It could also be the case that the scale factor was the cause of the fewer crashes. We for example tried $R_k = -10^6 \cdot (x_k^T Q x_k + \delta^2 R)$, i.e. with a scale factor of $10^6$. What is certain is that the scale factor has, for some for us unknown reason, a big impact and you should at least have $10^3$, but we didn't do too much exploration with that.
- The discount factor has in theory no direct connection to crashes, instead it influences how long the horizon is when optimizing from the state the bike is currently in (see e.g. the report

for more information regarding the discount factor). Since the LQR has an infininte horizon from a certain state $x$, then you would want a large $\gamma$ if you want get similar results as the LQR. We noticed that when we used $\gamma = 0.999$ instead of $\gamma = 0.99$ that the resulst got closer to that of the LQR as expected, but it was more difficult to get rid of the tiny oscillations around the upright position (less than 0.05 degrees for the roll angle and less than 0.1 degrees for the steering rate).

# Cost function

- We got our first good results using $\gamma = 0.99$ and the logarithm around the LQR-cost: $R_k = -\log\big(x_k^T Q x_k + \delta_k^2 R\big)$. However, when having the logarithm around the cost, it is actually the product of the $x_k^T Q x_k + \delta_k^2 R$ elements that are minimized (see the report for more information regarding this). When trying with $\gamma = 0.999$ and the log-cost, then the results got very poor, so it seems as if minimizing the product is only close to minimizing the sum when optimizing over a short horizon.
- We used the LQR-based cost functions for all the results in the report. However, when saturation of the steering rate was applied, the training process became very slow when having the LQR-cost so we also tried other cost-functions. The training process was dramatically decreased when instead using the simple cost function:

$$\begin{cases} R_k = 1 \text{ if } |\varphi_k| < 0.1^\circ \\ R_k = 0 \text{ else} \end{cases}, \tag{1}$$

  and variants of it. However, we didn't explore it that much, but it produced almost as good (or bad) results as the one in the report that used the LQR-cost. But the main message is that one should not be limited to using the LQR-cost, since it may not be ideal for training the agent even if it is the cost that you de facto want to minimize.

# Miscellanius useful information

- When choosing the length of the sequences to train the RL-algorithm on, it's advisable to have long enough sequences so that an ideal controller would have a period at the end where all states are close to zero. That way you force the RL-algoritm to not just get fast to zero, but also to stay there with small oscillations. For a network trained on all velocities, 3 second long sequences is probably enough. If you only want to train for a higher speed, e.g. 5 m/s, then 2 seconds or even 1.5 seconds would suffice. Since the sequence length affects the training time a lot, you want to choose as small length as possible, without suffering any drawbacks.
- If one would like to use a preexisting PPO algorithm, don't use OpenAI's baseline, since it seem to not really be working good for the continuous case (not just a problem for us, but we saw some comments on forums as well). Use for example OpenAI's spinningup, instead.
- The time it took to train an RL-agent varied based on the current problem and the computer, but it usually took a couple of hours to get a fine-tuned agent. To just get something that works ok can take much shorter time.
- Useful code on how to lower the learning rate of the optimizer:

```
for param_group in optimizer.param_groups:
    param_group['lr'] = lr
```