Policy Gradient in practice

Don't become an alchemist :)

Olivier Sigaud

Sorbonne Université
http://people.isir.upmc.fr/sigaud

Outline

- This class is the practical counterpart of a more theoretical policy gradient class available here:
- `https://www.youtube.com/watch?v=_RQYWSvMyyc`
- It is meant to come with labs
- Github repository: `https://github.com/osigaud/Basic-Policy-Gradient-Labs`
- We investigate basic policy gradient algorithms and phenomena
- A prerequisite before going to SOTA deep RL algorithms
- Understanding phenomena is better than using black-box algorithms

Content

- Studies of policy gradient phenomena on continuous CartPole, continuous MountainCar, Pendulum
- Use of Bernoulli, Gaussian and squashed Gaussian policies
- Visualization of policies, critics, learning curves
- Study of sum, discounted sum and advantage variants of the policy gradient
- A specific part about learning a critic
- Presentation of several issues and tricks
- Another video about the code itself

Policy gradient algorithms

- Reminder: policy gradient calculations can be summarized as follows:

policy

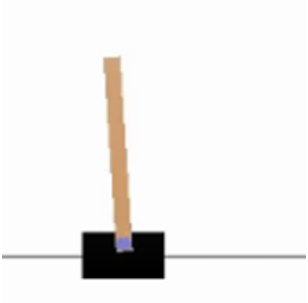
$$\nabla_\theta J(\theta) = \mathbb{E}[\psi_t \nabla_\theta \log \pi_\theta(a_t^{(i)}|s_t^{(i)})]$$

where ψ_t can be:

1. $\psi_t = \sum_{t=0}^{H} \gamma^t r_t$: total (discounted) reward of trajectory
2. $\psi_t = \sum_{t'=t}^{H} \gamma^{t'-t} r_{t'}$: sum of (discounted) rewards after a_t
3. $\psi_t = \sum_{t'=t}^{H} \gamma^{t'-t} r_{t'} - b(s_t)$: sum of rewards after a_t with baseline
4. $\psi_t = \delta_t = r_t + \gamma V^\pi(s_{t+1}) - V^\pi(s_t)$ with $V^\pi(s_t) = \mathbb{E}_{a_t}[\sum_{l=0}^{H} \gamma^l r_{t+l}]$
5. $\psi_t = Q^\pi(s_t, a_t) = \mathbb{E}_{a_{t+1}}[\sum_{l=0}^{H} \gamma^l r_{t+l}]$
6. $\psi_t = A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t) = \mathbb{E}[\delta_t]$

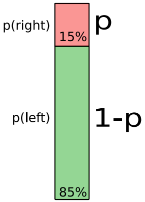
John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015

The CartPole-v0 environment



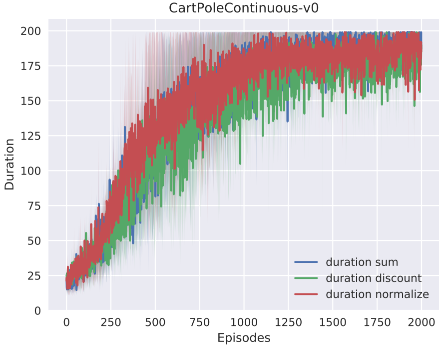
- The easiest gym classic control environment
- 4 state dimensions:
- Binary action: push left or right
- Custom continuous cartpole to study Gaussian continuous action policies (action in $[-1, 1]$)

Distributions over actions: Bernoulli



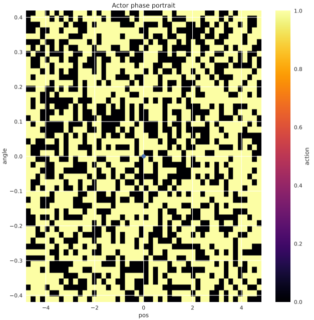
- Binary choice between two actions
- p is a probability, must keep between 0 and 1
- Use sigmoid, or tanh...
- Increasing $p(left)$ decreases $p(right)$

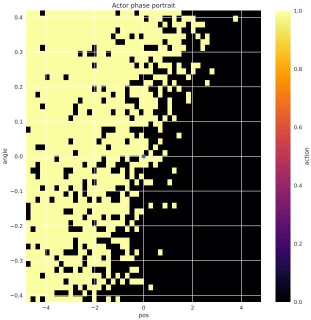
Results: Policy Gradient without critic



- Variance over 10 runs
- Sum, discounted sum and normalized advantage work well
- Stochasticity of the binary policy is enough
- No additional exploration

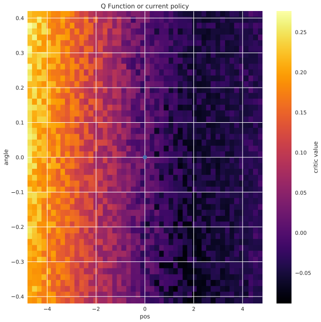
Initial/Final policy

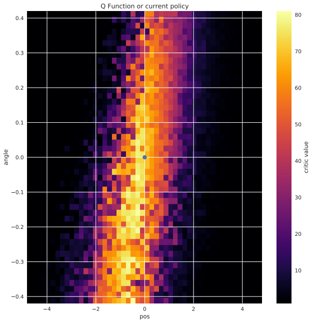




- 4 dimensions: pos, speed, angle and angular velocity
- `FeatureInverter` wrapper to display with pos and angle (see video about coding)
- black = push left, yellow = push right
- General idea: push left when right, right when left, then manage pole

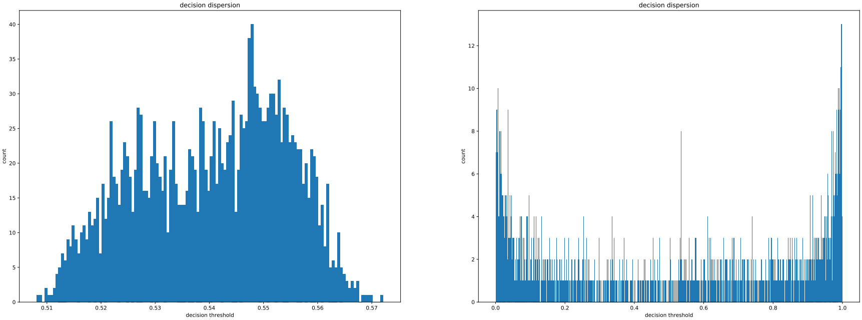
Initial/Final critic





- Obtained from Monte Carlo evaluation method
- Batches obtained from policies along training
- General idea: it is better to be with null angle and position

Initial/Final randomness

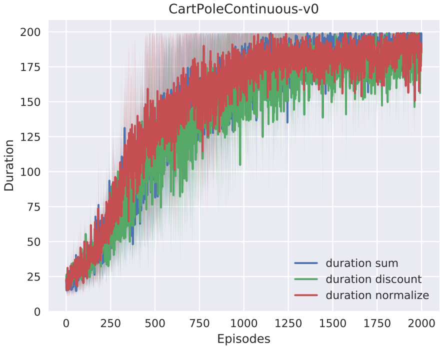


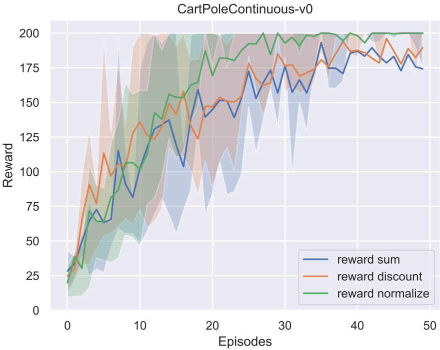
- Mind the scope on x-axis: initially very small (0.5 → 0.58, not centered)
- At the end of training, the policy is much less stochastic (more 0 and 1)
- Less exploration

Normalization issue

- In CartPole and CartPoleContinuous, $r = 1$ for all steps before failure
- Thus, in the advantage case, at all steps, $r - \bar{r} = 0$
- By discounting the reward, we avoid this
- In the sum case, longer trajectories are more rewarded
- Globally, poorly informative gradient

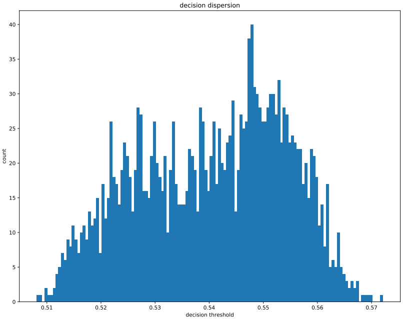
Deterministic vs stochastic evaluation

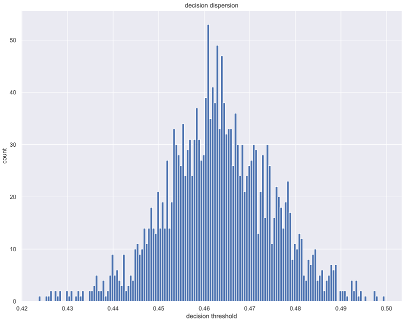




- Evaluating a deterministic version of the policy leads to less noisy results
- Less episodes because only evaluation episodes are displayed
- Separating training and evaluation epochs is a good standard

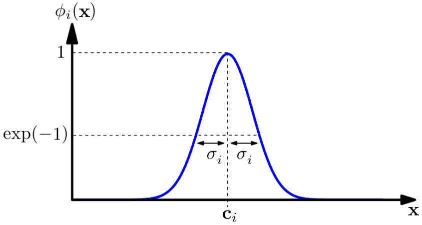
Two Initial Bernoulli policies





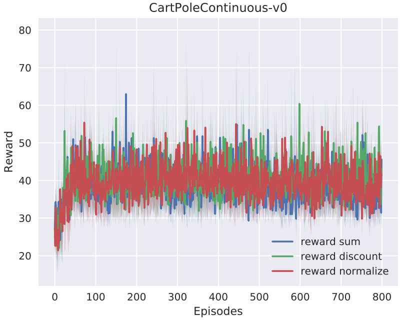
- With the default initialization
- Most often, initial decision thresholds are all above 0.5, or all below 0.5
- To make deterministic policy, choice if threshold > 0.5 or < 0.5
- Not random at all: always takes the same action!

Distributions over actions: Normal



- Choice of a continuous action (extension to mutidimensional with multivariate Gaussian)
- The integral must keep to 1
- Standard approach: keep variance σ constant
- Or apply gradient descent to variance too

Policy Gradient with Normal Policies



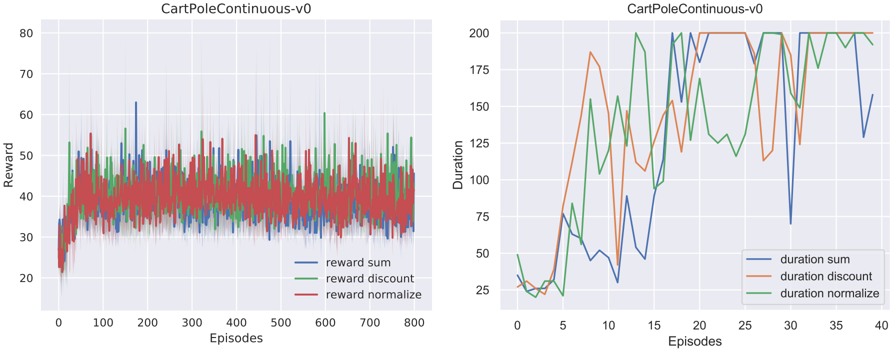
- Coded with adaptive variance
- Does not reach optimal performance

The NormalPolicy python class

```python
class NormalPolicy(GenericNet):
    def __init__(self, l1, l2, l3, l4, learning_rate):
        super(NormalPolicy, self).__init__()
        self.relu = nn.ReLU()
        self.fc1 = nn.Linear(l1, l2)
        self.fc2 = nn.Linear(l2, l3)
        self.fc_mu = nn.Linear(l3, l4)
        self.fc_std = nn.Linear(l3, l4)
        self.tanh = nn.Tanh()
        self.softplus = nn.Softplus()
        self.optimizer = torch.optim.Adam(self.parameters(), lr=learning_rate)

    def forward(self, state):
        state = torch.from_numpy(state).float()
        state = self.relu(self.fc1(state))
        state = self.relu(self.fc2(state))
        mu = self.tanh(self.fc_mu(state))
        std = 2  # self.softplus(self.fc_std(state))
        return mu, std

    def select_action(self, state):
        with torch.no_grad():
            mu, std = self.forward(state)
            n = Normal(mu, std)
            action = n.sample()
        return action.data.numpy().astype(int)

     def train_pg(self, state, action, reward):
        action = torch.FloatTensor(action)
        reward = torch.FloatTensor(reward)
        mu, std = self.forward(state)
        # Negative score function x reward
        loss = -Normal(mu, std).log_prob(action) * reward
        self.update(loss)
        return loss
```

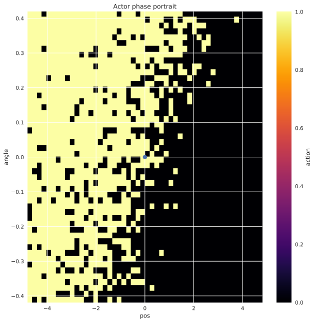
- Bug fix: all actions were too close to 0. Tuning std helps

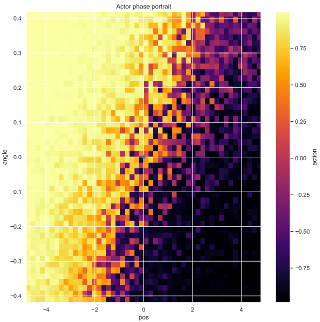
Policy Gradient with Normal Policies: bug fixed



- One might rather use a trained Gaussian width
- Or a squashed Gaussian (see SAC video)

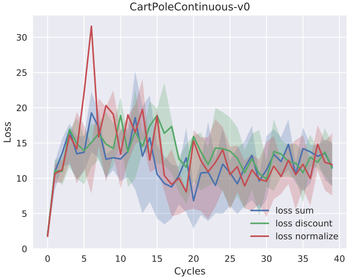
Normal Policy

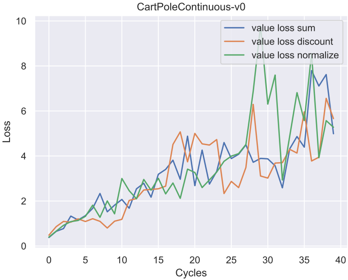




- Bernoulli (left) and Normal (right) policies
- Actions in a smaller range, and more continuous

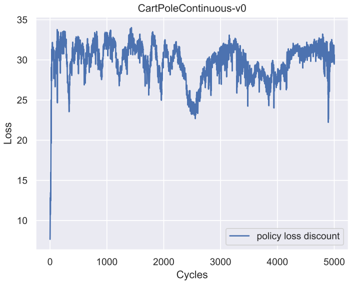
Losses of the critics

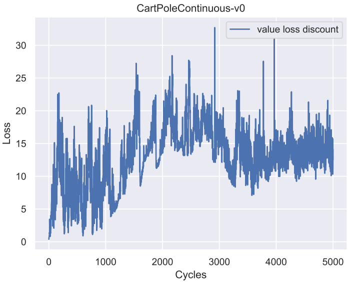




- Bernoulli (left) and Normal (right) policies
- The critic loss does not go to 0
- Same with the policy loss

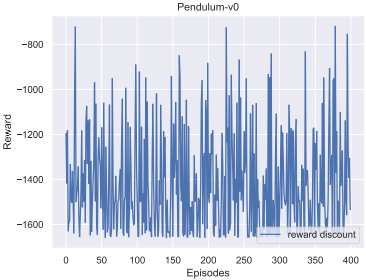
Losses of Bernoulli, longer run

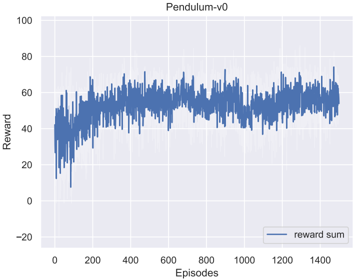




- In Bernoulli policies, randomness does not go down to 0
- In Normal policies, fixed Gaussian variance
- Squashed Gaussian policy: tunable variance, but same story
- If the loss goes to 0, the policy degenerates
- A “dirty secret” of RL papers...

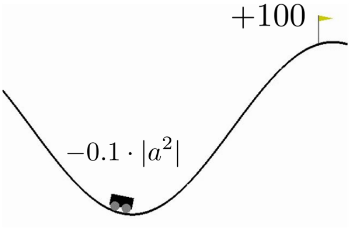
Reward Normalization issue: Pendulum





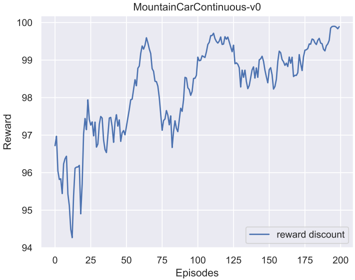
- A neural network learns better if the loss is around 0
- Rescale the reward function with a PendulumWrapper
- Then training works better
- Took 3 hours on a laptop computer (5 seeds), after slow tuning

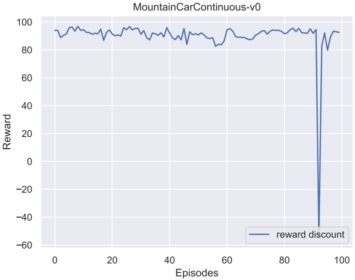
Continuous Mountain Car: Setup



- The slope is too strong for the engine
- Need to move left before going right
- Reward penalty for acting
- A Bernoulli policy cannot find weak actions
- Deceptive gradient effect: may stop moving

Initial Exploration Issue

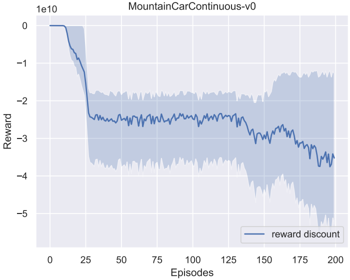


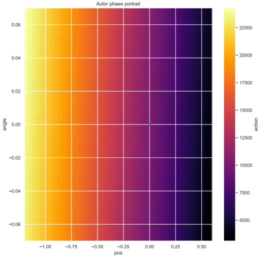


- Slowly converges to not moving (return + 100)
- No initial Bernoulli nor Normal policy can find the reward
- Initialize policy with regression from an expert policy: sometimes it works!
- Better idea: use a more efficient exploration method

Cédric Colas, Olivier Sigaud, and Pierre-Yves Oudeyer (2018) GEP-PG: Decoupling exploration and exploitation in deep reinforcement learning algorithms. *arXiv preprint arXiv:1802.05054*

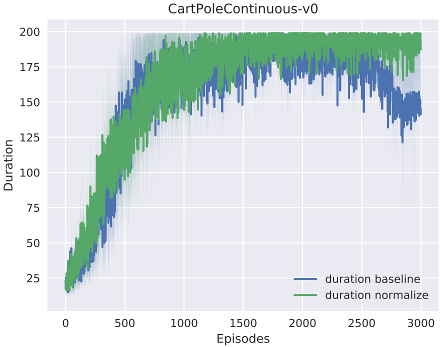
Still gradient failures





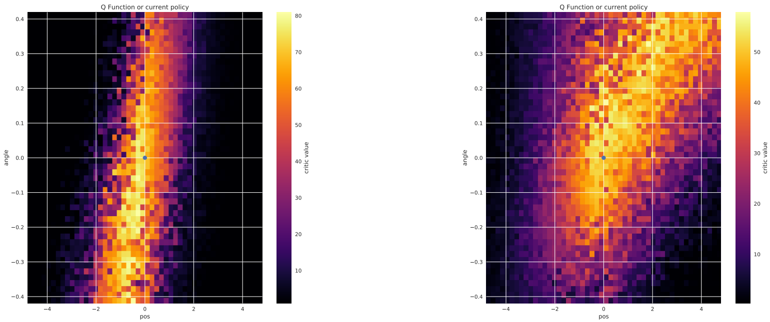
- With Gaussian policy, huge negative reward
- The action is unbounded, and goes far away from 1 (the reward considers the unbounded action)
- A squashed Gaussian policy may avoid this

Policy Gradient with critic baseline



- Learning the baseline (here, a Q-function) works well
- Until the lack of exploration results in critic degeneracy
- Sometimes, degeneracy is much more abrupt

Monte Carlo critic from optimal policy



- Trained MC critic from random policy versus from top policy
- From a top policy, it does not work anymore
- Data along the same optimal trajectory: not enough exploration

Monte Carlo critic estimation

```
def train_critic_mc(self, gamma, critic, n, train):
    """
    Trains a critic through a Monte Carlo method. Also used to perform n-step training
    :param gamma: the discount factor
    :param critic: the trained critic
    :param n: the n in n-step training
    :param train: True to train, False to just compute a validation loss
    :return: the average critic loss
    """
    if n == 0:
        self.discounted_sum_rewards(gamma)
    else:
        self.nstep_return(n, gamma, critic)
    losses = []
    targets = []
    for j in range(self.size):
        episode = self.episodes[j]
        state = np.array(episode.state_pool)
        action = np.array(episode.action_pool)
        reward = np.array(episode.reward_pool)
        target = torch.FloatTensor(reward).unsqueeze(1)
        targets.append(target.mean().data.numpy())
        critic_loss = critic.compute_loss_to_target(state, action, target)
        if train:
            critic.update(critic_loss)
        critic_loss = critic_loss.data.numpy()
        losses.append(critic_loss)
    mean_loss = np.array(losses).mean()
    return mean_loss
```

```
def compute_loss_to_target(self, state, action, target):
    """
    Compute the MSE between a target value and the critic value for
    :param state: a state or vector of state
    :param action: an action or vector of actions
    :param target: the target value
    :return: the resulting loss
    """
    val = self.forward(state, action)
    return self.loss_func(val, target)
```

- The algorithm collects a batch of trajectories with states, actions and reward
- Reward values are discounted along trajectories
- Target values are these discounted values for any (state, action) pair
- The loss is the difference between the target and the output of the critic network for the same (state, action) pair
- Target values are independent from critic values

Bootstrap (Temporal Difference) critic estimation

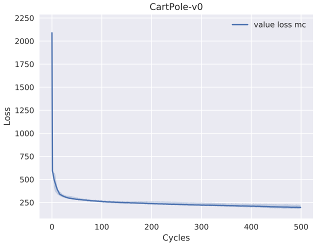
```
def train_critic_td(self, gamma, policy, critic, train):
    """
    Trains a critic through a temporal difference method
    :param gamma: the discount factor
    :param critic: the trained critic
    :param policy:
    :param train: True to train, False to compute a validation loss
    :return: the average critic loss
    """
    losses = []
    for j in range(self.size):
        episode = self.episodes[j]
        state = np.array(episode.state_pool)
        action = np.array(episode.action_pool)
        reward = np.array(episode.reward_pool)
        done = np.array(episode.done_pool)
        next_state = np.array(episode.next_state_pool)
        next_action = policy.select_action(next_state)
        target = critic.compute_bootstrap_target(reward, done, next_state
        target = torch.FloatTensor(target).unsqueeze(1)
        critic_loss = critic.compute_loss_to_target(state, action, target)
        if train:
            critic.update(critic_loss)
        critic_loss = critic_loss.data.numpy()
        losses.append(critic_loss)
    mean_loss = np.array(losses).mean()
    return mean_loss
```

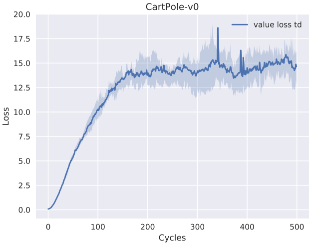
```
def compute_bootstrap_target(self, reward, done, next_state, next_action, gamma):
    """
    Compute the target value using the bootstrap (Bellman backup) equation
    The target is then used to train the critic
    :param reward: the reward value in the sample(s)
    :param done: whether this is the final step
    :param next_state: the next state in the sample(s)
    :param next_action: the next action in the sample(s) (used for SARSA)
    :param gamma: the discount factor
    :return: the target value
    """
    next_value = np.concatenate(self.forward(next_state, next_action).data.numpy()
    return reward + gamma * (1 - done) * next_value
```

- Implements the Bellman update equation
- Minimize $\delta_t = r_t + \gamma Q_\theta(s_{t+1}, a_{t+1}) - Q_\theta(s_t, a_t)$
- The target is $y_i = r_i + \gamma Q_\theta(s_{i+1}, a_{t+1})$
- Update the critic by minimizing $L = 1/N \sum_i (y_i - Q_\theta(s_i, a_i))^2$
- Target values depend on critic values

MC vs TD estimation

- Obtained from Monte Carlo batches from a top policy with low variance





- MC case:
- The targets keep the same: this is a regression problem
- No need to recompute the targets from the batch when the critic changes

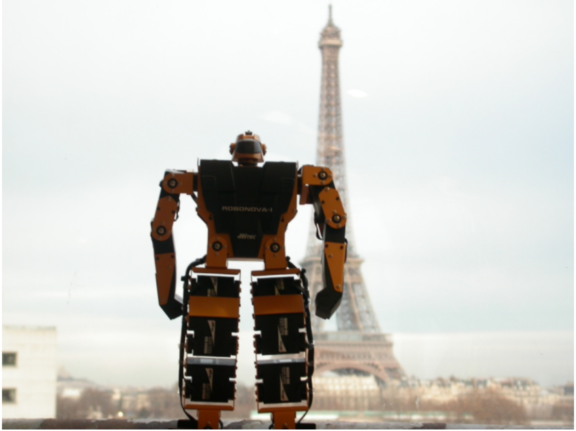
- TD case:
  - In the beginning, critic values are all 0
  - Thus the loss are all low
  - But critic values must reach their optimal values (as MC values)
  - The TD error ↑, then should ↓ to 0
  - Need to recompute the targets at each iteration

Take home messages

- Each environment comes with its own issues
- CartPole is the easiest gym classic control benchmark
- Basic policy gradient algorithms somewhat work after some tuning
- Making it work requires investigating and understanding phenomena
- SOTA Deep RL algorithms are more powerful, but may still fail on simplistic benchmarks

Guillaume Matheron, Nicolas Perrin, and Olivier Sigaud. (2019) The problem with ddpg: understanding failures in deterministic environments with sparse rewards. *arXiv preprint arXiv:1911.11679*

Any question?



Send mail to: `Olivier.Sigaud@upmc.fr`

Cédric Colas, Olivier Sigaud, and Pierre-Yves Oudeyer.
GEP-PG: Decoupling exploration and exploitation in deep reinforcement learning algorithms.
arXiv preprint arXiv:1802.05054, 2018.

Guillaume Matheron, Nicolas Perrin, and Olivier Sigaud.
The problem with DDPG: understanding failures in deterministic environments with sparse rewards.
arXiv preprint arXiv:1911.11679, 2019.

John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan, and Pieter Abbeel.
High-dimensional continuous control using generalized advantage estimation.
arXiv preprint arXiv:1506.02438, 2015.