continuous_control_tasks

April 14, 2018

Reinforcement learning algorithms can generally be divided into two categories: model-free, which learn a policy or value function, and model-based, which learn a dynamics model. In both categories, conventional wisdom has been that methods based on random search in the parameter space of policies are not very efficient for reinforcement learning problems.

The methodology from "Augmented Random Search" opens up the possibility to use simple policies for continuous control problems.

In this fork, several policies have been created as a starting point for interested developers or researchers. Moreover, given that DeepMind recently released thier control suite for continuous problems, a simple wrapper was also created to build policies on top of thier control suite.

To follow along, you will first need to install "Mujoco Pro v1.50". Then make sure you install "dm_control".

"Ray" and "Ray RLib" are used to speed up the training process.

Even though this fork has some MLP policies, all the policies have been implemented in Numpy. So a somewhat recent numpy version should work.

Finally, moviepy is used to render the agents.

1 Prelims

```
In [1]: import matplotlib.pyplot as plt
        import matplotlib.mlab as mlab
        import matplotlib.animation as animation
        import numpy as np
        import time
        from IPython.display import Image
        from dm_control import suite
        from basic_env import BasicEnv
        from moviepy.editor import *
        from moviepy.video.io.bindings import mplfig_to_npimage
        from IPython.display import Math
        import seaborn
        seaborn.set_context(context="talk")
        %matplotlib inline
```

1.1 Table of Contents

2 Background

Reinforcement learning is an area of machine learning that tries to understand how agents should take actions in an environment to maximize potential rewards.

```
In [2]: %%latex
        \begin{equation}
        R_t = \mathbb{E}[r|\pi] = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2}^2 + ... = \sum\limits_
        \end{equation}
```

$$R_t = \mathbb{E}[r|\pi] = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2}^2 + ... = \sum_{i=0}^{\infty} \gamma^i r_{t+i} \tag{1}$$

```
In [3]: %%latex
        with $\pi$ as the policy and the discount factor $\gamma \in [0, 1]$.
```

with π as the policy and the discount factor $\gamma \in [0, 1]$.

The agent uses the policy to choose which action it is going to play or select next. Neural networks are powerful function approximators and are used either directly as policy functions, taking a state as input and outputting a probability distribution over the action set; or as value estimators, by outputting a scalar which estimates the value of states and actions related to the estimated achievable discounted reward from those states.

Recent research, however, suggests that linear policies may produce similar impressive results in reinforcement learning. We will test both linear and non-linear policies in this repository.

In this fork, we use the recently released environment from DeepMind, dm_control.

3 The DeepMind Control Suite

The DeepMind Control Suite (aka dm_control) provides a set of benchmarks for evaluating and comparing learning algorithms for continuous control problems.

3.0.1 The Walker domain

A planar walker with three available tasks: stand, walk, and run. For the walk task, for example, the reward motivates an upright torso and forward velocity.

```
In [4]: env = suite.load('walker','walk', visualize_reward=True)

In [5]: print('Action space:', env.action_spec())

Action space: BoundedArraySpec(shape=(6,), dtype=dtype('float64'), name=None, minimum=[-1. -1. -
```

For the environment mentioned above, the action is a vector of size 6. Thus, any function could be used to represent the policy as long as the output is consistent with the action space.

```
In [6]: time_step = env.reset()
        print('Observation space:', time_step.observation)
```

```
Observation space: OrderedDict([('orientations', array([-0.32438406,  0.94592546,  0.44794237,
        0.96730809,  0.04759251,  0.99886683, -0.28646093,  0.95809192,
       -0.99998084, -0.00618975, -0.85123174, -0.52478998])), ('height', 1.3), ('velocity', arra
```

time_step.observation provides the "observations" available to the learning agent. The dimension or shape of the observation is domain/task specific. For the walker-walk, for example, 24 elements are sent to the learning agent.

Basic RL steps The following code demostrates how to run a single episode with a random agent, and compute the reward. As we run 1000 steps, the maximum possible score is 1000.

As basic as this code block looks, it summarizes pretty nicely the steps of any RL algorithm: Given the observations or state that the environment provides, the RL algorithm needs to be able to find a strategy to maximime the total reward.

```
In [7]: spec = env.action_spec()
        time_step = env.reset()
        total_reward = 0.0
        for _ in range(1000):
            action = np.random.uniform(spec.minimum,spec.maximum,size=spec.shape)
            time_step = env.step(action)
            total_reward += time_step.reward
        print("Total reward after 1000 time steps is:", total_reward)
```

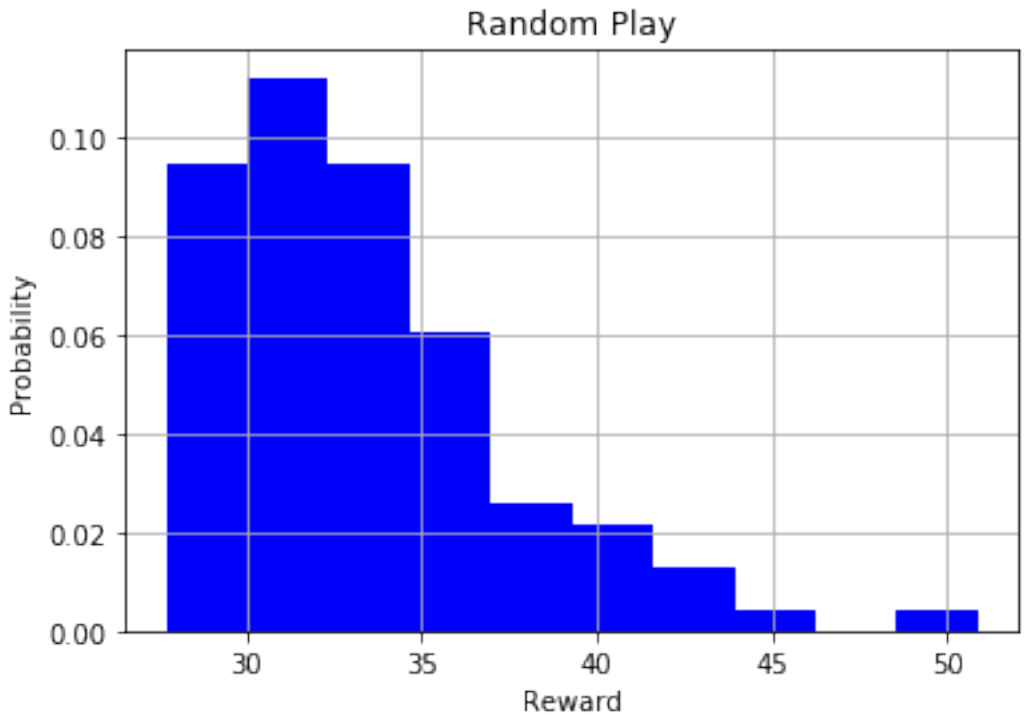
```
Total reward after 1000 time steps is: 29.9514277539404
```

Random play As much as we would like to jump into complicated policies, sometimes it is a good idea to check the behavior of random agents. The code below uses once again a plain-vanilla random policy. This time we run 100 rollouts as a way to obtain a distribution of total rewards.

```
In [96]: action_spec = env.action_spec()
         time_step = env.reset()
         returns = []
         max_frame = 1000
         frames = 0
         width, height = 480, 480
         video = []
         for i in range(100):
             obs = env.reset()
             done = False
```

```
        totalr = 0.
        steps = 0
        while not done:
            action = np.random.uniform(action_spec.minimum,action_spec.maximum,size=action_
            time_step = env.step(action)
            if frames < max_frame:
                video.append(env.physics.render(height, width, camera_id=0))
            totalr += np.float32(time_step.reward)
            done = time_step.last()
            steps += 1
            frames += 1
        returns.append(totalr)
```

In [97]:
```
# the histogram of the rewards
n, bins, patches = plt.hist(returns,facecolor='blue',  normed=1)
plt.xlabel('Reward')
plt.ylabel('Probability')
plt.title('Random Play')
plt.grid(True)
plt.show()
```



The max reward per rollout is by design 1000, yet the random agent cannot obtain more than 50. This illustrates the complexity of the optimization problem.

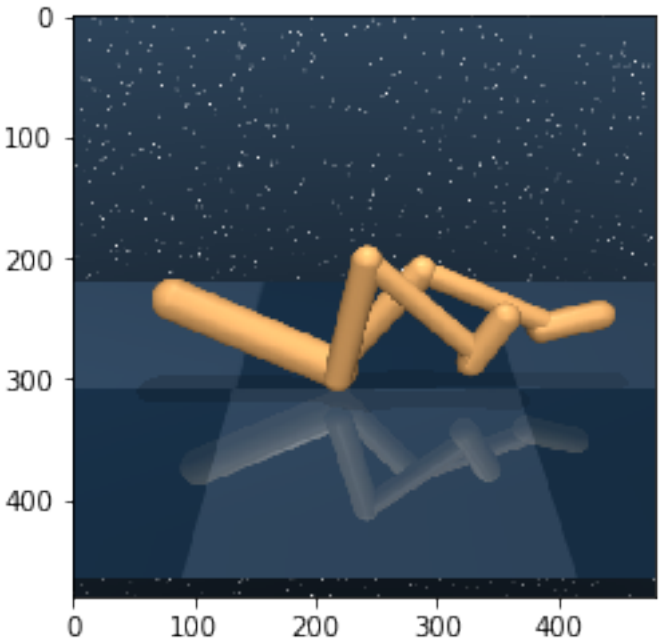
Given the "video" array created above, we can visualize the agent for several episodes. Below is an animation of the random agent.

```
In [38]: from IPython.display import HTML

In [101]: fig = plt.figure()
          ims = []
          for i in range(100):
              im = plt.imshow(video[i])
              ims.append([im])

          anim = animation.ArtistAnimation(fig, ims, interval = 50, blit= True)
          HTML(anim.to_html5_video())

Out[101]: <IPython.core.display.HTML object>
```



How can we estimate a policy for this domain?

4 Linear policies

Let's stick to the walker-walk domain for the time being. We just saw the impact of taking random actions.

Somehow, we need to take into account the state of the environment (ie, the observations) in order to take an "informed" action or decision. After all, we are after some sort of intelligent behavior.

```
In [36]: import six
         output_key='observations'
         time_step = env.reset()
```

```
        observation = time_step.observation
        keys = six.iterkeys(time_step.observation)
        observation_arrays = [observation[key].ravel() for key in keys]
        flatten_observation = type(observation)([(output_key, np.concatenate(observation_arrays
        print('flatten_observations:', flatten_observation["observations"])
```

```
flatten_observations: [-0.43414985  0.90084067  0.74237176  0.66998819 -0.5534474   0.83288413
 -0.79495278  0.60667131  0.82495251  0.56520205  0.3423599   0.93956889
  0.03322552  0.99944788  1.3         0.          0.          0.
  0.          0.          0.          0.          0.          0.        ]
```

What would be the easiest approach to produce an informed decision?

Given this vector of 24 observations or 1x24 matrix, we need a 24x6 matrix in order to obtain a 1x6 vector of actions. And this exactly the definition of a linear policy. A simple parameter matrix that when multiplied by the vector of observations produces a vector of actions.

Let's create a function to flatten the observations now as we will use it going forward.

```
In [12]: def flatten_observations(observation):
             keys = six.iterkeys(time_step.observation)
             observation_arrays = [observation[key].ravel() for key in keys]
             flatten_observation = type(observation)([(output_key, np.concatenate(observation_ar
             return flatten_observation
```

```
In [13]: weights = np.random.randn(6*24).reshape(24, 6)
         predicted_actions = np.dot(flatten_observation["observations"], weights)
         print('predicted_actions:', predicted_actions)
```

```
predicted_actions: [-2.80178807  1.93377489 -0.47818649  2.44143362  2.98622059 -6.00529312]
```

Now we only need to understand how to modify the initial weights in order to increase the rewards of the agent.

Let's define a rollout function.

```
In [14]: def rollouts(env, steps, weights):
             weights = weights.reshape(24, 6)
             spec = env.action_spec()
             time_step = env.reset()
             ob = flatten_observations(time_step.observation)
             total_reward = 0.0
             for _ in range(steps):
                 action = np.dot(ob["observations"], weights)
                 action = np.random.uniform(spec.minimum,spec.maximum,size=spec.shape)
                 time_step = env.step(action)
                 total_reward += time_step.reward
             return total_reward
```

```
In [17]: # Test the rollouts function
         rollouts(env, 1000, np.random.randn(6*24))

Out[17]: 38.187552360791976
```

If we start with zero weights, we could shock these weights with noise. Let's try it out:

```
In [27]: w_star = np.zeros(6*24)
         v = 0.03 # Variance
         delta = v*np.random.randn(6*24)

         w_test_up = w_star + delta
         w_test_down = w_star - delta

         rewards_up = rollouts(env, 1000, w_test_up)
         rewards_down = rollouts(env, 1000, w_test_down)

         print("Rewards with a positive shock: ", rewards_up)
         print("Rewards with a negative shock: ", rewards_down)

Rewards with a positive shock:  29.421268597033784
Rewards with a negative shock:  31.764229222161347
```

The parameters of a linear policy can be estimated using a simple update step:

```
In [29]: learning_rate = 0.05
         w_star = w_star + learning_rate*(rewards_up - rewards_down)*delta
```

Unfortunately, we face several challenges. First, we need to find a way to run many rollouts, for both shocks and for many deltas. Moreover, we need to find a way to summarize the results of these multiple rollouts. Nonetheless, lets try a single-core implementation below.

```
In [35]: def calculate_rollouts(env, weights, num_rollouts=5):
             reward = []
             for _ in range(num_rollouts):
                 reward.append(rollouts(env, 1000, weights))
             return np.mean(reward)/ (np.std(reward) + 1e-8) # Play it safe

         def obtain_new_weights(env, weights, num_rollouts=5, v=0.03,learning_rate = 0.05, num_d
             final_weights = weights
             for _ in range(num_deltas):
                 delta = v*np.random.randn(6*24)
                 w_test_up = weights + delta
                 w_test_down = weights - delta
                 rewards_up = calculate_rollouts(env, w_test_up, num_rollouts)
                 rewards_down = calculate_rollouts(env, w_test_down, num_rollouts)
                 final_weights+= (1./num_deltas)*learning_rate*(rewards_up - rewards_down)*delta
             return final_weights
```

```
        # Run the algorirthm for a few seconds...
        new_weights = w_star
        for i in range(5):
            new_weights = obtain_new_weights(env, new_weights, num_rollouts=2, v=0.05,learning_
            print("Iteration %d: Reward %d." % (i+1, rollouts(env, 1000, new_weights) ))
```

```
Iteration 1: Reward 30.
Iteration 2: Reward 39.
Iteration 3: Reward 31.
Iteration 4: Reward 30.
Iteration 5: Reward 49.
```

Clearly, the code above is not efficient. And this is where Ray and RLlib help.

Interested in seeing how the linear policy behaves after 8h of training, try:

python code/run_ars_policy.py --policy_type="mlp"

and a gif will be create to visualize the behavior of the trained neural network agent. The gif will be create in trained_policies/walker-walk/mlp folder.

If you want to train your own agent for another domain/task, for example humanoid-run, you just need to type:

python code/ars.py --policy_type="mlp" --domain_name="humanoid" --task_name="run"

The linear policy after approximately 8h of training can reach levels of 500 in the walker-walk domain. Below please find an example of the linear agent.

```
In [39]: HTML('<img src="img/linear-walker-walk.gif">')
```

```
Out[39]: <IPython.core.display.HTML object>
```

Several policies have been implemented, and many more are in my to-do list. As of today, the MLP policy with layer normalization is performing quite well in the walker-walk optimization problem, reaching levels of 700.

```
In [104]: HTML('<img src="img/mlp-walker-walk.gif">')
```

```
Out[104]: <IPython.core.display.HTML object>
```

Still interested? Then have a look at ARS.py, run_ars_policy.py, and policies.py. These files leverage Ray and RLib to estimate many type of policies leveraging all the available computing power.