Teach your AI how to walk | Solving BipedalWalker | OpenAIGym

Shiva Verma Follow
May 14 · 6 min read ★

Image by ChianImage

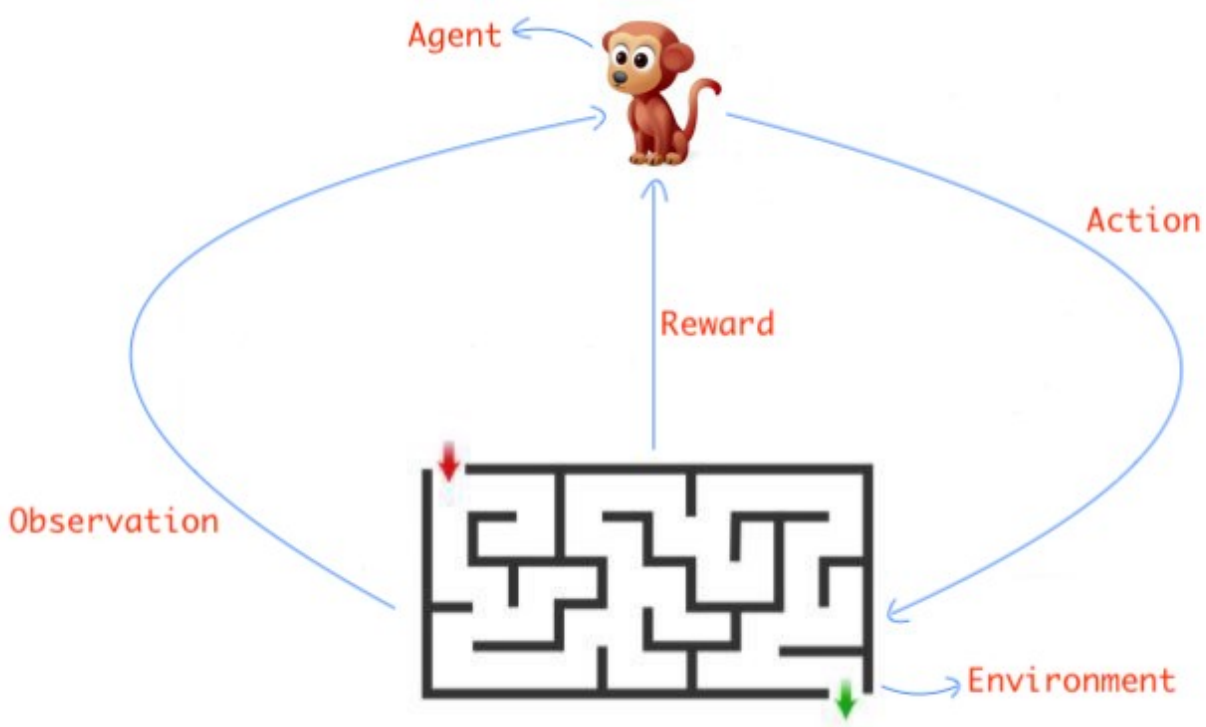
In this blog, we are going to teach a simple AI how to walk, with the help of Reinforcement Learning.

Let's first briefly understand what is Reinforcement Learning and what is this AI, which we are going to train.

. . .

Reinforcement learning, Brief Intro

Reinforcement learning is a branch of Machine learning. The Idea behind it is, you have an **agent** and an **environment**. The agent takes actions and environment gives reward based on those actions, The goal is to teach the agent optimal behaviour in order to maximize the reward received by the environment.



Reinforcement learning

For example, have a look at the diagram. This maze represents our **environment**. Our purpose would be to teach the agent an optimal policy so that it can solve this maze. The maze will provide a reward to the agent based on the goodness of each action it takes. Also, each action taken by agent leads it to the new **state** in the environment.

. . .

About the AI

The AI which we are going to train is an OpenAIGym environment, And its name is **BipedalWalker.**

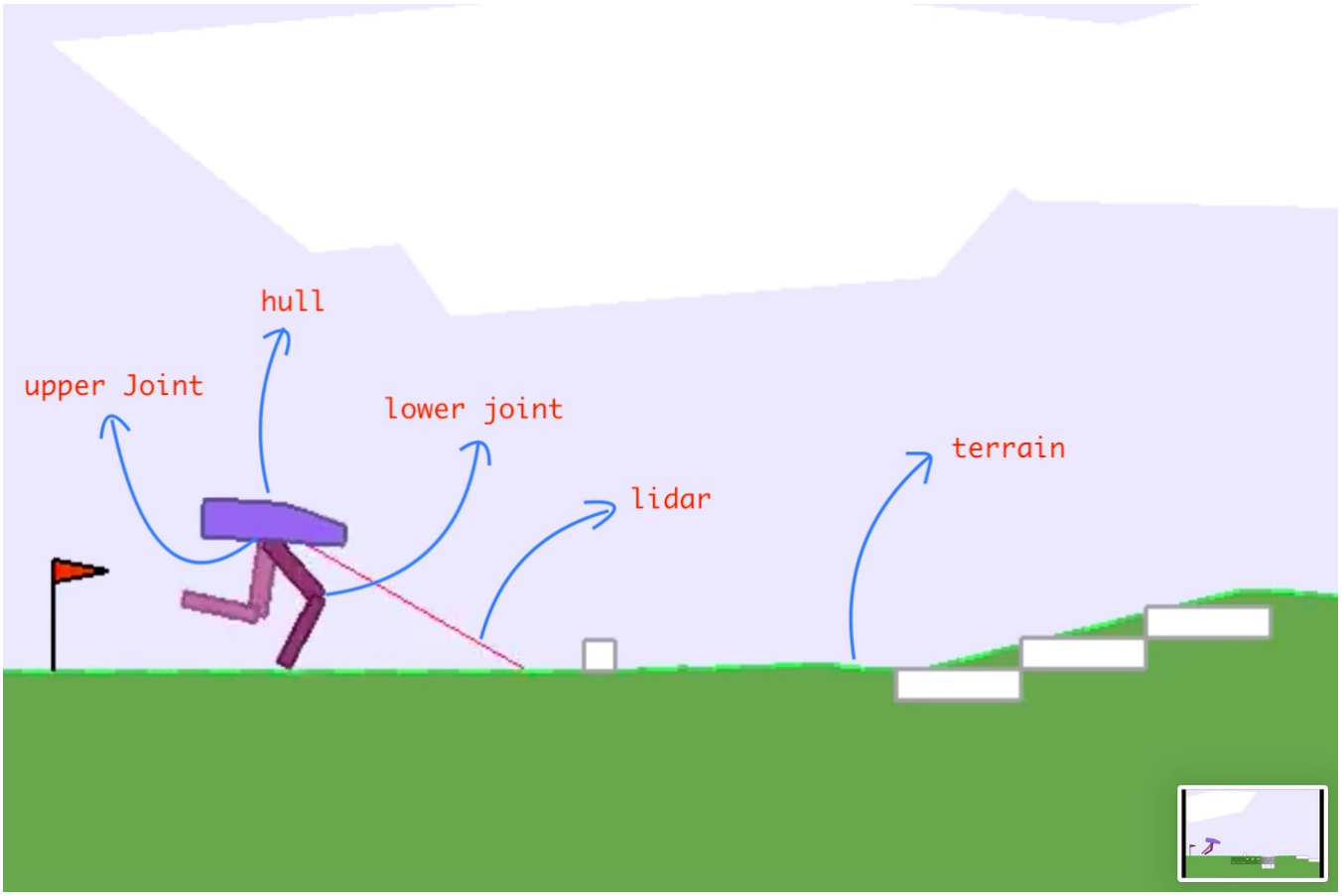
OpenAIGym is a collection of some really cool environments. You should take a look at **OpenAIGym** website to know more about reinforcement learning environments and how to use them.

Now let's look at the BipedalWalker environment in detail.

. . .

BipedalWalker

> *Reward is given for moving forward, total 300+ points up to the far end. If the robot falls, it gets -100. Applying motor torque costs a small amount of points, more optimal agent will get better score. State consists of hull angle speed, angular velocity, horizontal speed, vertical speed, position of joints and joints angular speed, legs contact with ground, and 10 lidar rangefinder measurements. There's no coordinates in the state vector.*



Action Space

BipedalWalker has 2 legs. Each leg has 2 joints. You have to teach the Bipedal-walker to walk by applying the torque on these joints. Therefore the size of our action space is 4 which is torque applied on 4 joints. You can apply the torque in the range of (-1, 1)

Reward

- The agent gets a positive reward proportional to the distance walked on the terrain. It can get a total of 300+ reward all the way up to the end.
- If agent tumbles, it gets a reward of -100.
- There is some negative reward proportional to the torque applied on the joint. So that agent learns to walk smoothly with minimal torque.

There are 2 versions of the **Bipedal** environment based on terrain type.

- Slightly uneven terrain.
- Hardcore terrain with ladders, stumps and pitfalls.

. . .

Solving the environment

I am using the DDPG algorithm to solve this problem. DDPG is suitable when we have continuous action and state space a. I am using Pytorch library.

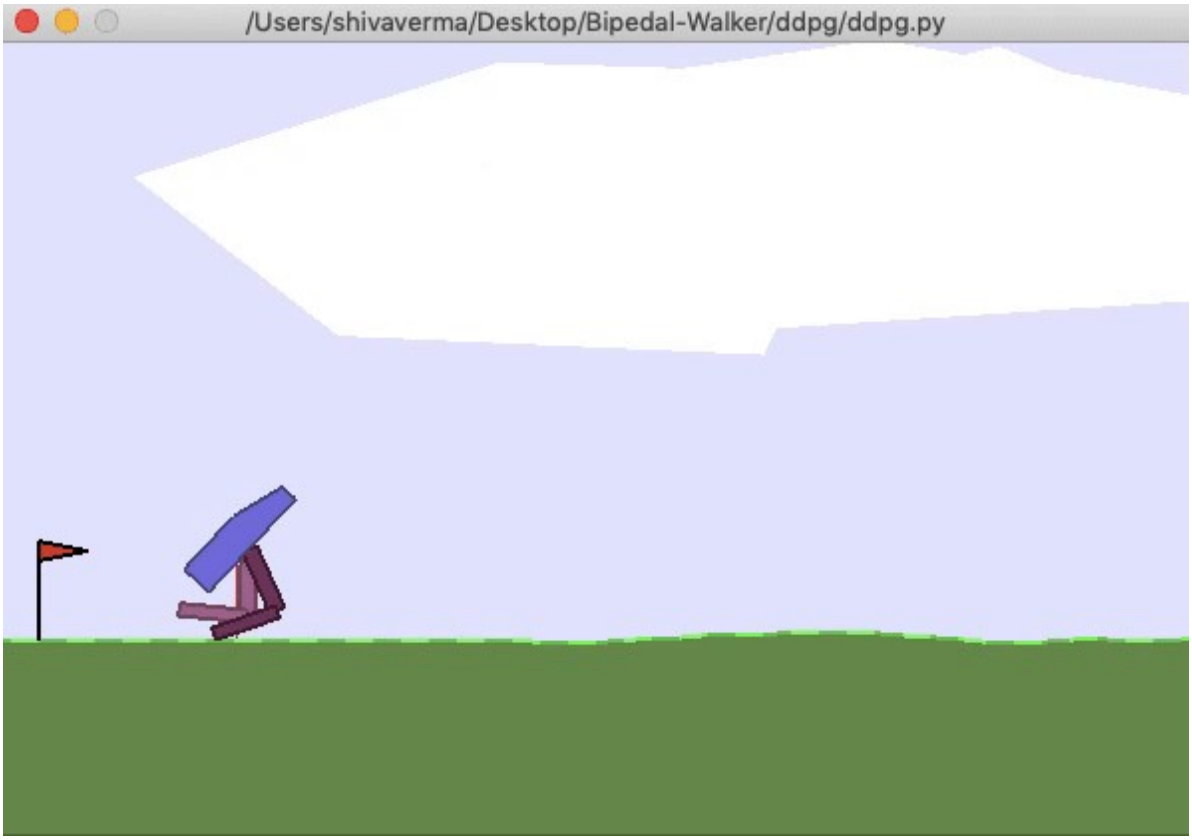
In DDPG there are two different networks called Actor and Critic. Actor-network output action value, given states to it. Critic network output the Q value (how good state-action pair is), given state and action(produces to by the actor-network) value pair. You can read about the DDPG in detail from the sources available online. I have also attached some link in the end.

Let's first see the training of BipedalWalker. Later we will cover the algorithm in detail.

. . .

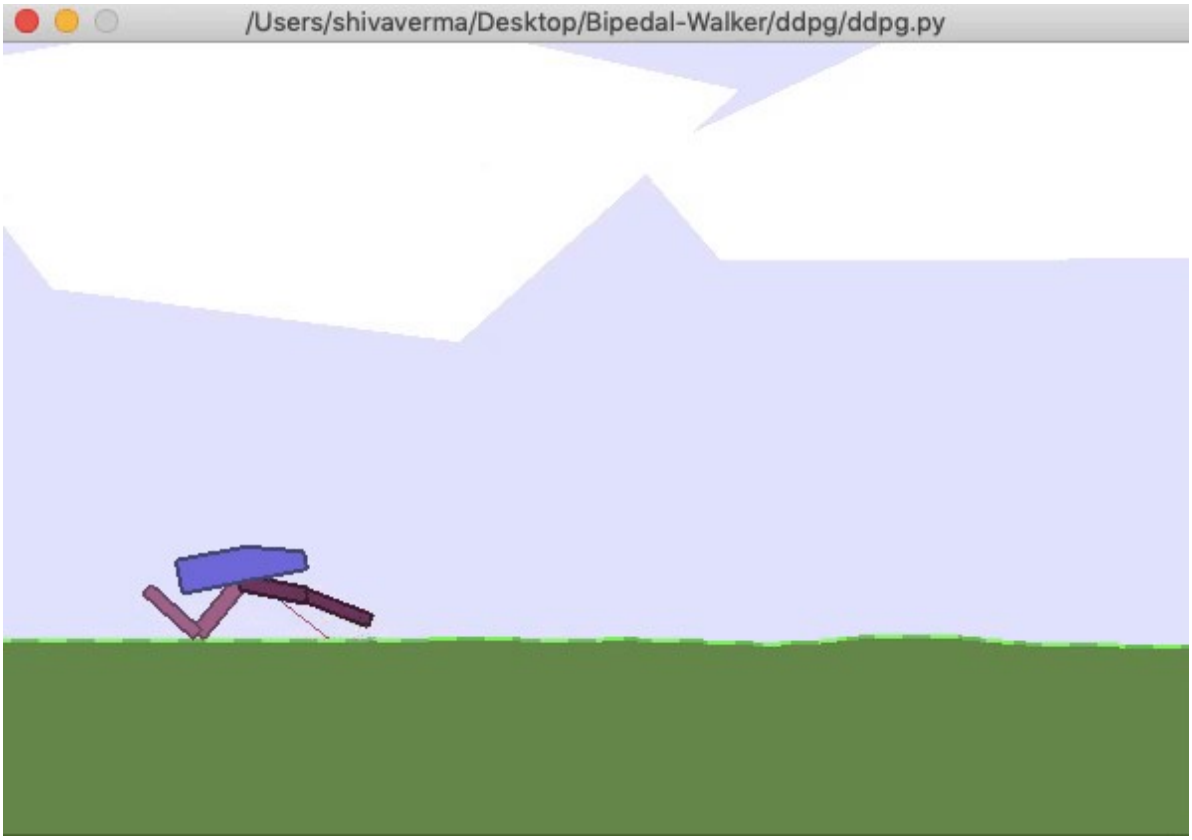
Training Visualization (Slightly uneven terrain)

- In the beginning, AI is behaving very randomly. It does not know how to control and balance the legs.



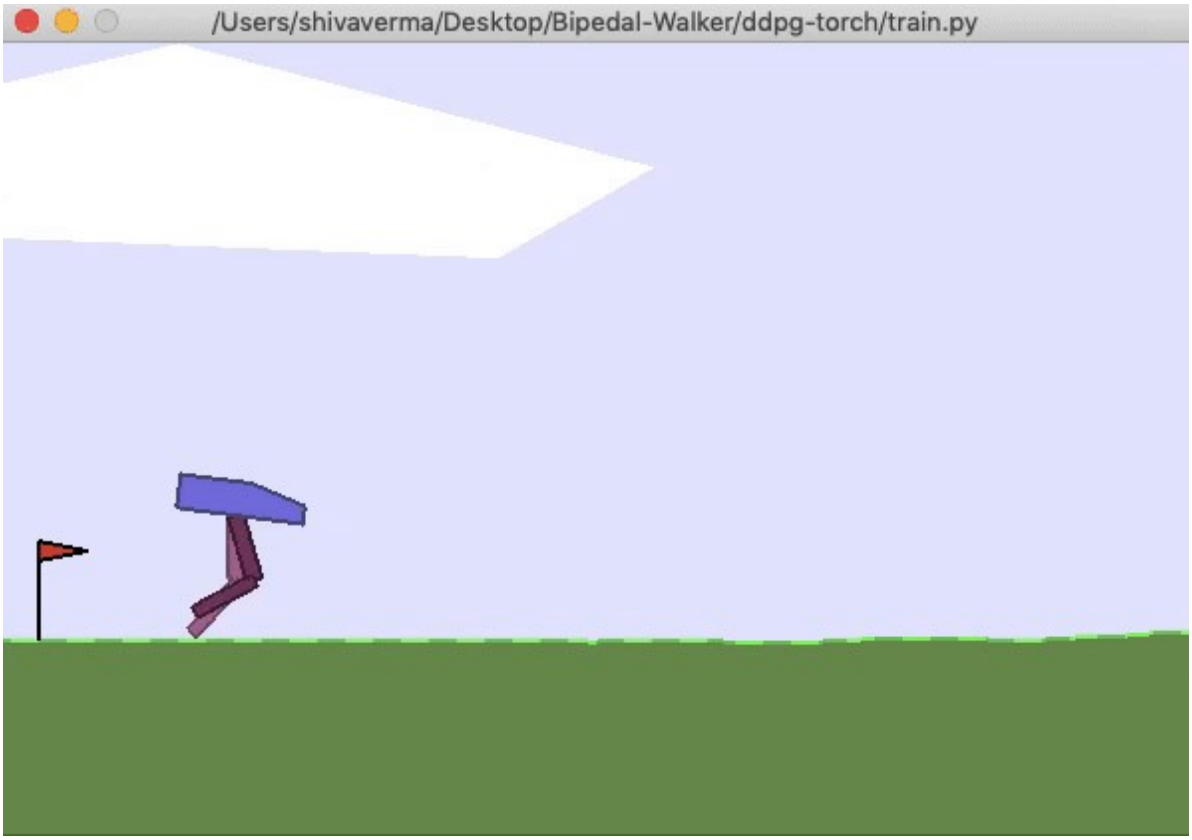
In the beginning

- After 300 episodes, it learns to crawl on one knee and one leg. This AI is playing safe now because if it tumbles then it gets -100 reward.



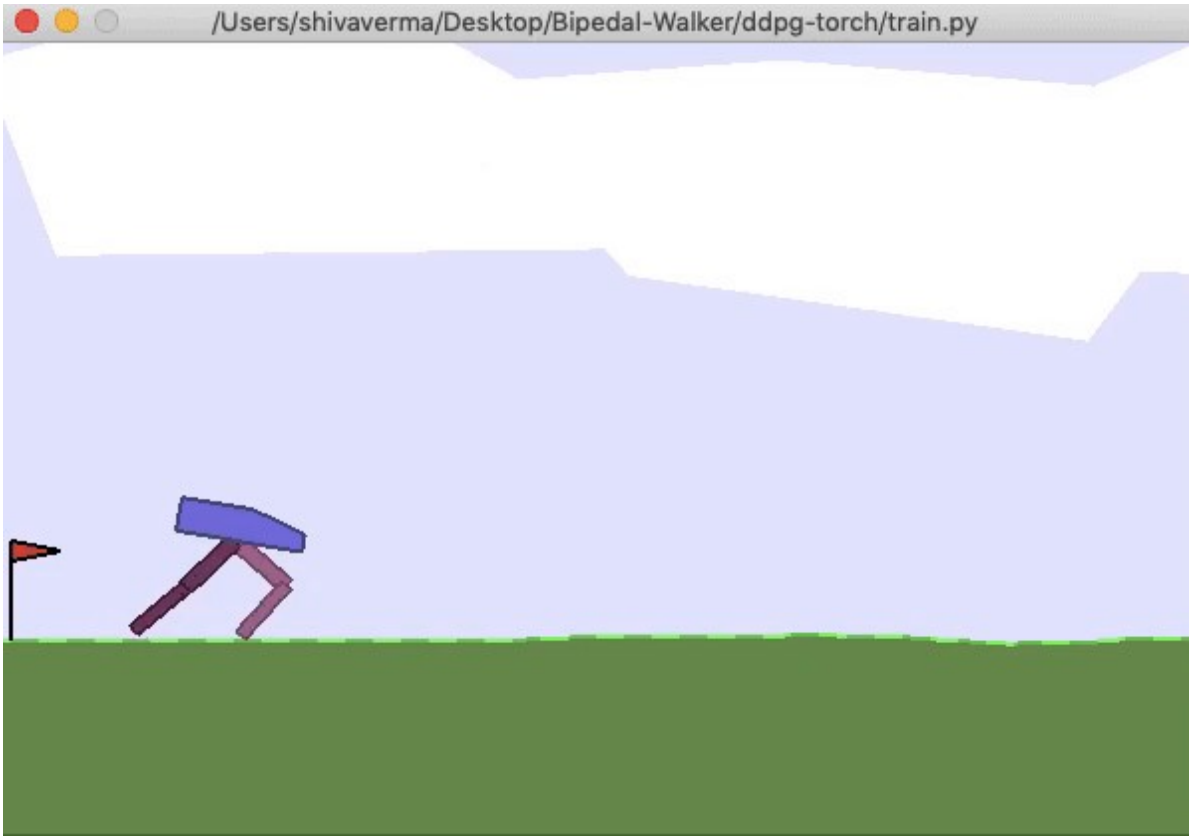
After 300 episodes

- After 500 episodes it started to balance on both of the legs. But It still needs to learn how to walk properly.



After 500 episodes

- After 600 episodes, it learns to maximize the rewards. It is walking in some different style. After all, it's an AI not a Human. This is just one of the way to walk in order to get maximum reward. If I train it again, it might learn some other optimal way to walk.

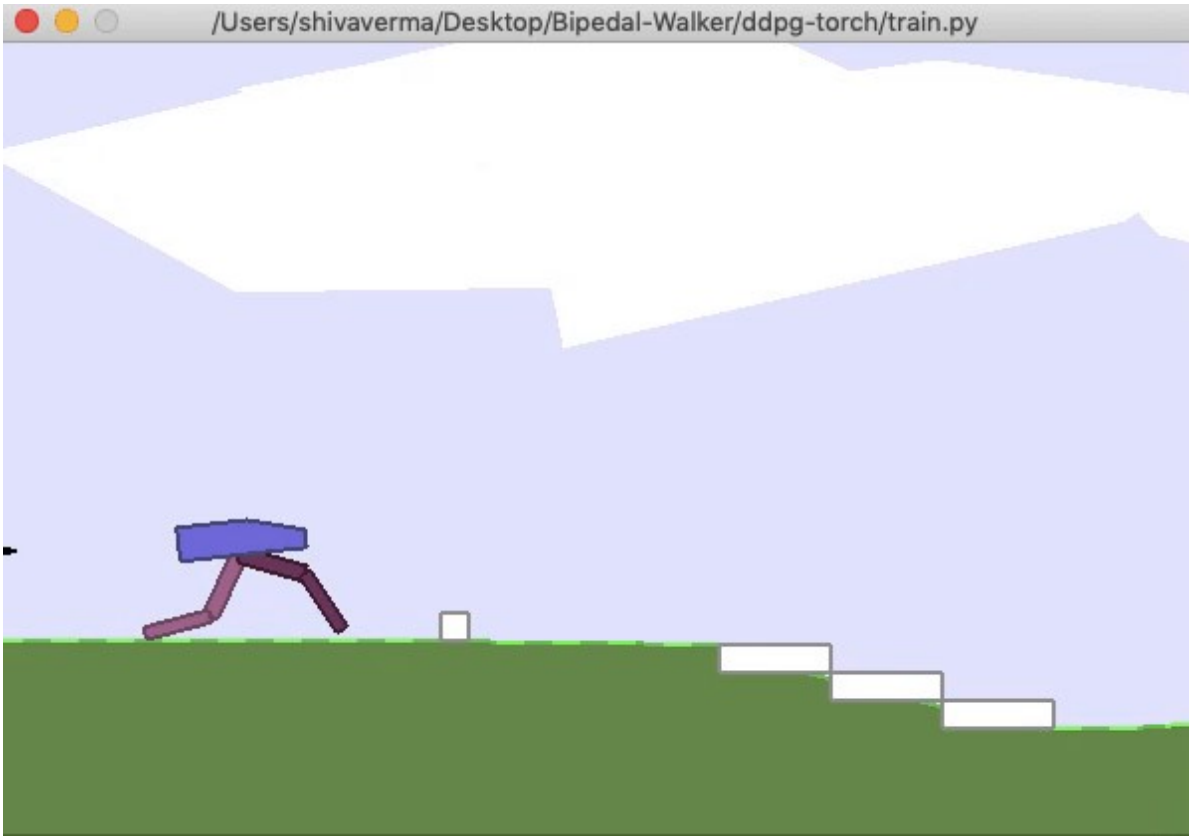


After 600 episodes

. . .

Training Visualization (Hardcore terrain)

- I saved my weight from the previous training on simple terrain and resumed my training on the hardcore terrain. I did it because the agent already knew how to walk on simple terrain and now it needs to learn how to cross obstacles while walking.



After 1000 more episodes

- After 1000 episodes of more training, the agent was able to walk on hardcore terrain. But sometimes it falls. To be able to walk completely on hardcore terrain it needs to be trained very extensively for more time.

. . .

DDPG Network Architecture

I have chosen the following hyperparameters for my network.

```
BUFFER_SIZE = 1000000      # replay buffer size
BATCH_SIZE = 100           # minibatch size
GAMMA = 0.99               # discount factor
TAU = 0.001                # for soft update of target parameters
LR_ACTOR = 0.0001          # learning rate of the actor
LR_CRITIC = 0.001          # learning rate of the critic
WEIGHT_DECAY = 0.001       # L2 weight decay
```

bipedal3.py hosted with ❤ by **GitHub** **view raw**

Hyperparameters

Following is the code snippet of my actor-network. Actor network consists of 2 hidden layers with **batch-norm** and **relu** activation. Size of the hidden layers is 600 and 300. I am using **tanh** activation on final layer because the action is bound in the range (-1, 1). Instead of using default weights, I initialized my weights which improved the performance.

```python
class Actor(nn.Module):

    def __init__(self, state_size, action_size, seed, fc_units=600, fc1_units=300):

        super(Actor, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc_units)
        self.fc2 = nn.Linear(fc_units, fc1_units)
        self.fc3 = nn.Linear(fc1_units, action_size)

        self.bn1 = nn.BatchNorm1d(fc_units)
        self.bn2 = nn.BatchNorm1d(fc1_units)
        self.reset_parameters()

    def reset_parameters(self):

        self.fc2.weight.data.uniform_(-1.5e-3, 1.5e-3)
        self.fc3.weight.data.uniform_(-3e-3, 3e-3)

    def forward(self, state):
        """Build an actor (policy) network that maps states -> actions."""
        x = F.relu((self.bn1(self.fc1(state))))
        x = F.relu((self.bn2(self.fc2(x))))
        return F.torch.tanh(self.fc3(x))
```

bipedal1.py hosted with ❤ by **GitHub** **view raw**

Actor Network

Following is the code snippet of my critic-network. There are 2 hidden layers for state and 1 hidden state for action. I am using **batch-norm** and **relu** activation here. Hidden layers of action and space are added together with **relu** activation on it.

```python
class Critic(nn.Module):

    def __init__(self, state_size, action_size, seed, fcs1_units=600, fcs2_units=300, fca1_units

        super(Critic, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fcs1 = nn.Linear(state_size, fcs1_units)
        self.fcs2 = nn.Linear(fcs1_units, fcs2_units)
        self.fca1 = nn.Linear(action_size, fca1_units)
        self.fc1 = nn.Linear(fcs2_units, 1)
        self.bn1 = nn.BatchNorm1d(fcs1_units)
        self.reset_parameters()

    def reset_parameters(self):

        self.fcs2.weight.data.uniform_(-1.5e-3, 1.5e-3)
        self.fc1.weight.data.uniform_(-3e-3, 3e-3)

    def forward(self, state, action):
        """Build a critic (value) network that maps (state, action) pairs -> Q-values."""
        xs = F.relu((self.bn1(self.fcs1(state))))
        xs = self.fcs2(xs)
        xa = self.fca1(action)
        x = F.relu(torch.add(xs, xa))
        return self.fc1(x)
```

bipedal2.py hosted with ❤ by **GitHub** **view raw**

Critic Network

After a lot of experimentation with the network architecture, I have chosen this one. I have saved the weights so that you can pick it up and load if you need. I am including a link to my Git repo which contains all the stuff.

shivaverma/OpenAIGym

Solving OpenAI Gym problems. Contribute to shivaverma/OpenAIGym development by creating an account on...

github.com

. . .

Resources

- This is an awesome introductory **blog** on Reinforcement Learning.
- I will highly recommend you to read the DDPG **paper**.
- Read **this** doc to know how to use Gym environments.
- Check out other cool **environments** on OpenAIGym.
- Following are my previous blogs on reinforcement learning.

Solving Reinforcement Learning Classic Control Problems | OpenAIGym.

If you are new to reinforcement learning and want to give it a try, then OpenAIGym is the right place to begin from.

towardsdatascience.com

- This is my previous **blog** on OpenAIGym Classic control problems.
- This is my previous **blog** on solving Lunar-Lander environment.

Machine Learning Reinforcement Learning Artificial Intelligence Deep Learning Openai Gym