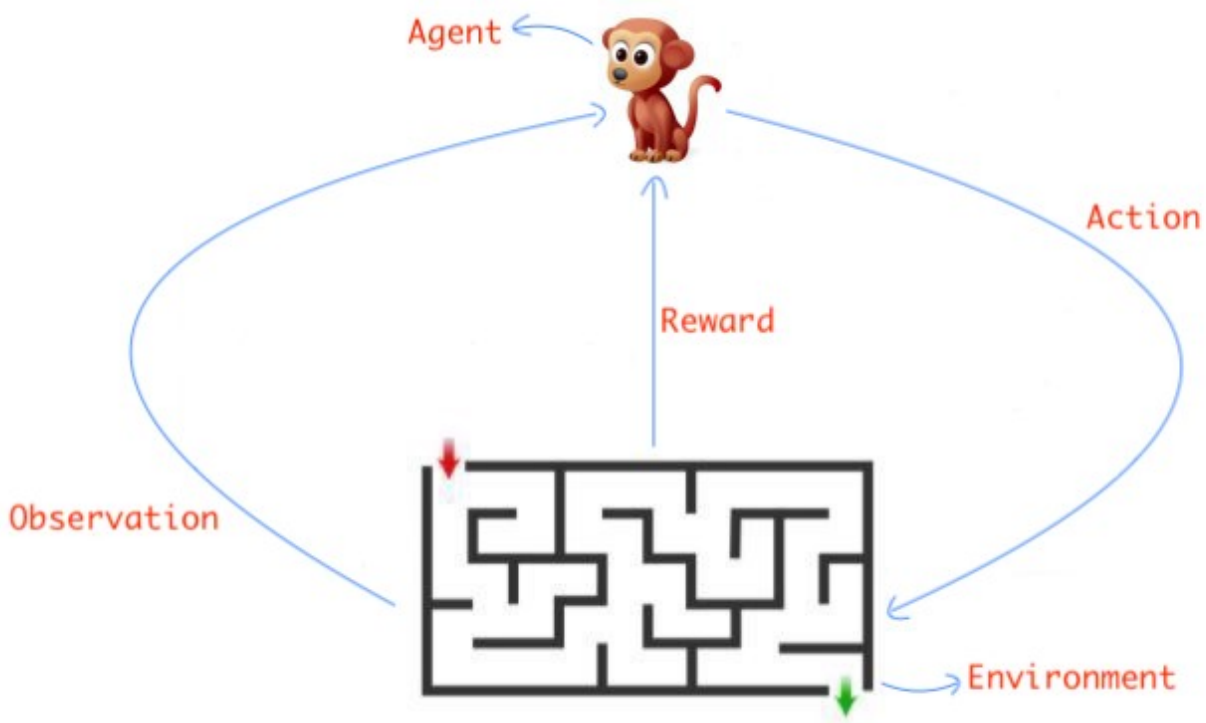
# Solving Reinforcement Learning Classic Control Problems | OpenAIGym

Shiva Verma Follow
Mar 27 · 6 min read ★


Image by Wallhere

If you are new in reinforcement learning and want to give it a try, then

Reinforcement learning is an interesting area of Machine learning. The rough Idea is that 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.



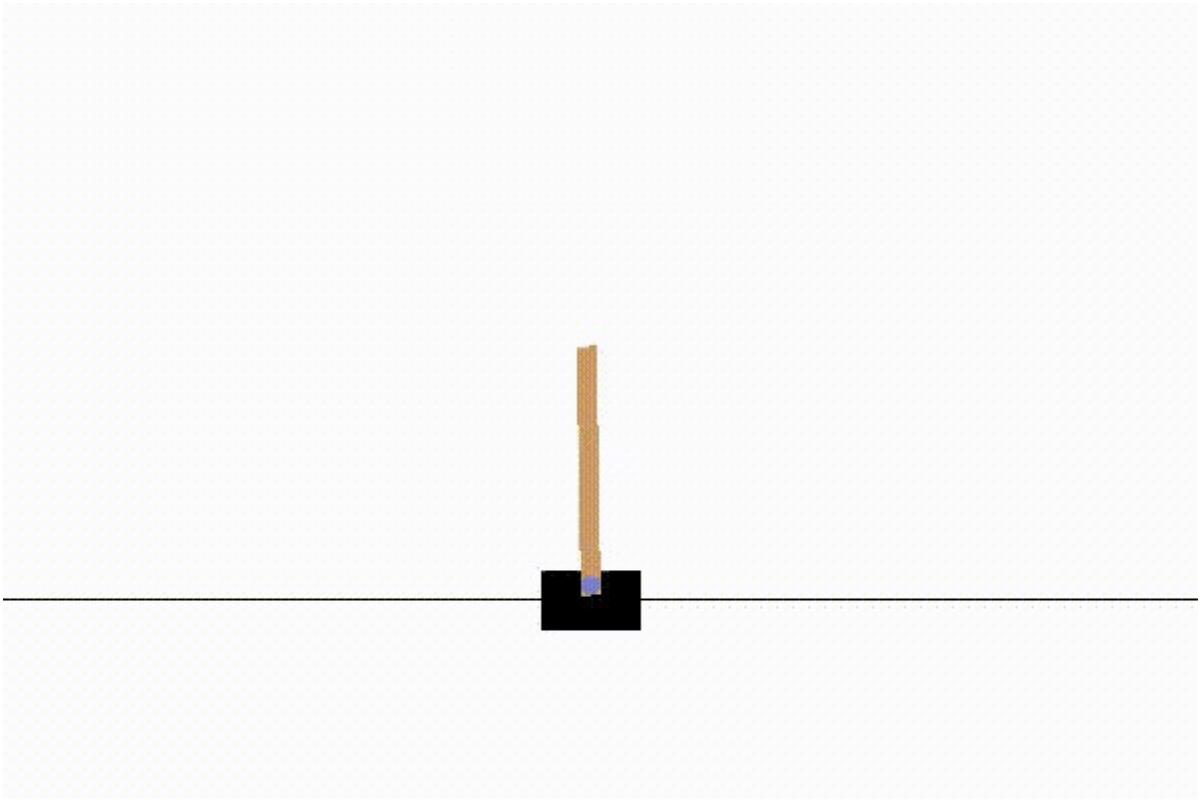

Reinforcement Learning Diagram

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.

will leave 2 environments for you to solve as an exercise. Please read **this** doc to know how to use Gym environments. Let's begin.

. . .

## CartPole-v1



Cart-Pole trained agent

### About the environment

> *A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to*

> *or the cart moves more than 2.4 units from the centre.*

## Action space (Discrete)

- `0` - Apply 1 unit of force in the left direction on the Cart
- `1` - Apply 1 unit force in the right direction on the cart

## State space (Continuous)

- `0` - Cart Position
- `1` - Cart Velocity
- `2` - Pole Angle
- `3` - Pole Velocity At Tip

In this environment, we have a discrete action space and continuous state space. In order to maximize the reward agent has to balance the pole as long as it can. Because it is getting the reward of +1 for each time step.

## Solving the environment

I am solving this problem with the DQN algorithm, which is compatible and works well when you have a discrete action space and continuous state space.

I will not be going into details of how DQN works. DQN approximate the actions using a neural network. There is much more to read about it. There

## Network Architecture for Cart-Pole

```
class DQN:

    def __init__(self, action_space, state_space):

        self.action_space = action_space
        self.state_space = state_space
        self.epsilon = 1
        self.gamma = .95
        self.batch_size = 64
        self.epsilon_min = .01
        self.epsilon_decay = .995
        self.learning_rate = 0.001
        self.memory = deque(maxlen=10000)
        self.model = self.build_model()

    def build_model(self):

        model = Sequential()
        model.add(Dense(24, input_shape=(self.state_space,), activation='relu'))
        model.add(Dense(24, activation='relu'))
        model.add(Dense(self.action_space, activation='linear'))
        model.compile(loss='mse', optimizer=adam(lr=self.learning_rate))
        return model
```

cartpole.py hosted with ❤ by GitHub view raw

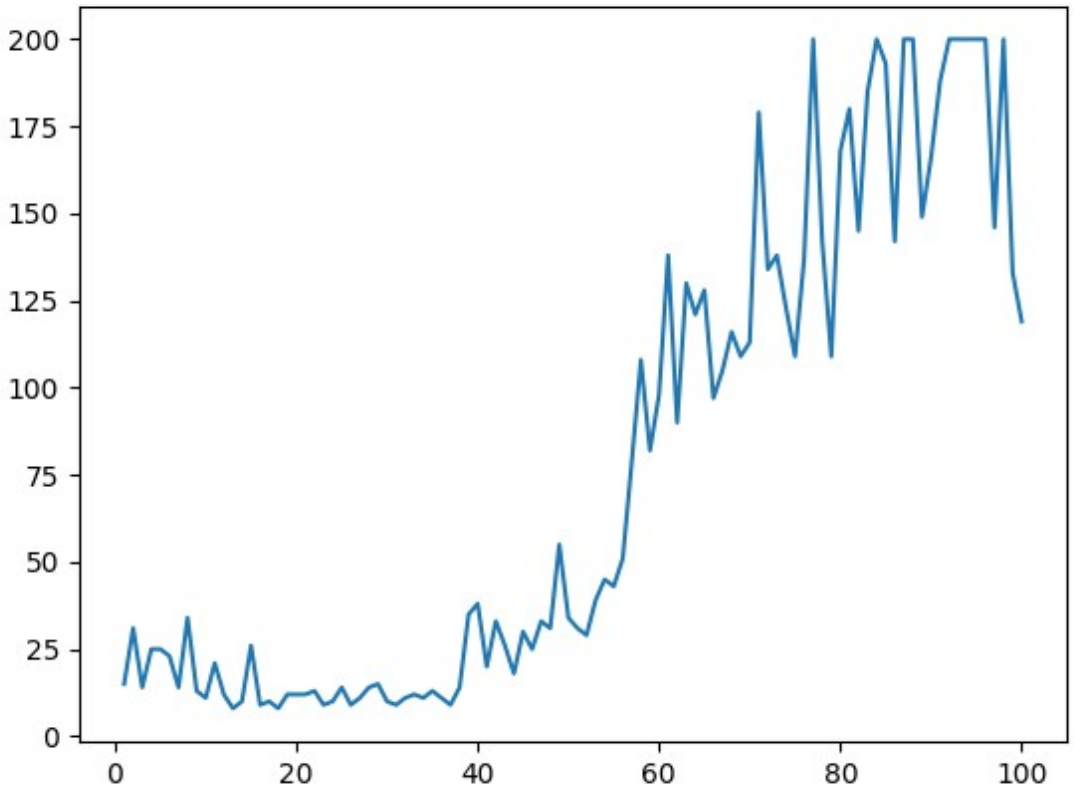
Code snippet of DQN

I have attached the snippet of my DQN algorithm which shows network architecture and hyperparameters I have used.

Input size of the network should be equal to the number of states. Output size of the network should be equal to the number of actions an agent can

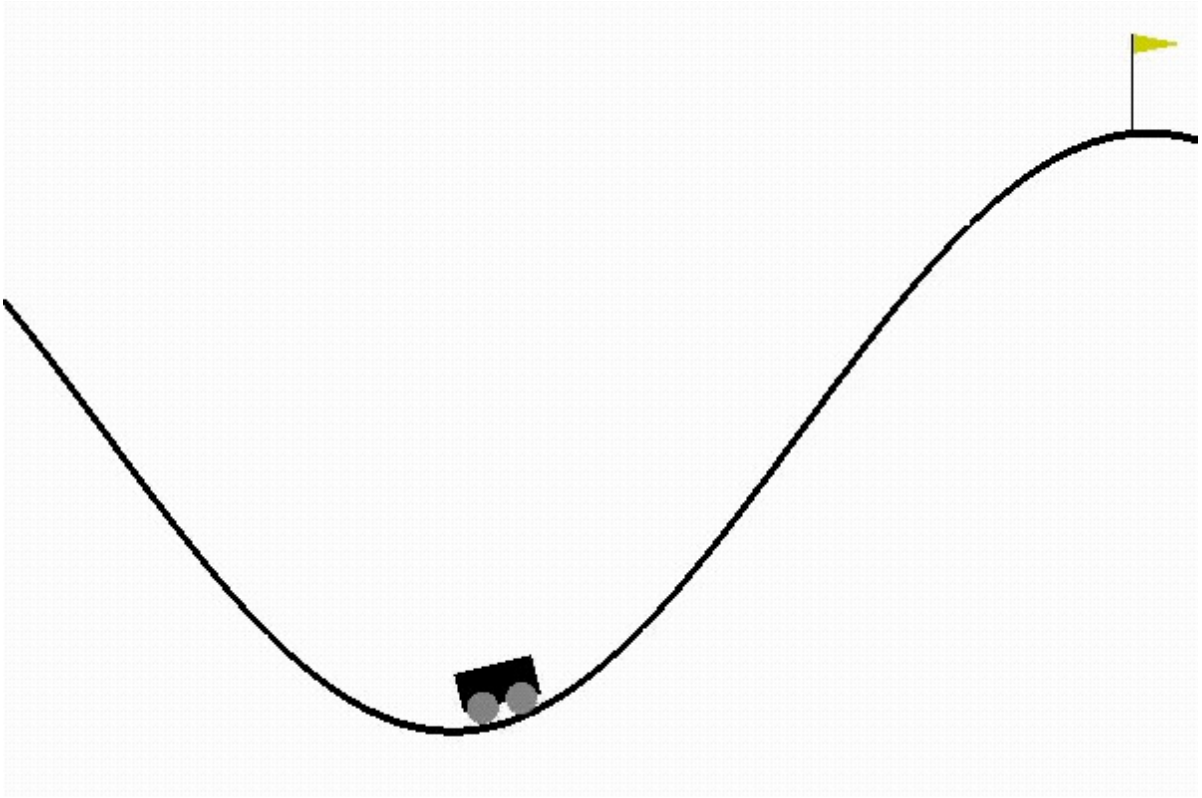
My network size is small. It consists of 2 hidden layers of size 24 each with **relu** activation. The task of balancing a pole is quite simple that is why the small network is able to solve it quite well.

Following is the plot showing rewards per episode. I was able to solve this environment in around 80 episodes.




Cartpole reward vs episode plot

. . .



Mountain-Car trained agent

## About the environment

> *A car is on a one-dimensional track, positioned between two "mountains". The goal is to drive up the mountain on the right; however, the car's engine is not strong enough to scale the mountain in a single pass. Therefore, the only way to succeed is to drive back and forth to build up momentum.*

## Action space (Discrete)

- `0` - Apply 1 unit of force in the left direction on the car
- `1` - Do nothing

- `0` - Car position
- `1` - Car velocity

In this environment, you get a reward of +100
when car reaches the goal position at the top. Now there is a trick to catch in the reward function. Until the car will not reach the goal it will not get any reward and behaviour of the car will not change. And there is very little chance that car will reach the goal just by random actions.

To solve this problem I have overwritten the reward function with my custom reward function. Here is the code snippet below.

```python
def get_reward(state):

    if state[0] >= 0.5:
        print("Car has reached the goal")
        return 10
    if state[0] > -0.4:
        return (1+state[0])**2
    return 0
```

mountain_car1.py hosted with ❤ by GitHub view raw

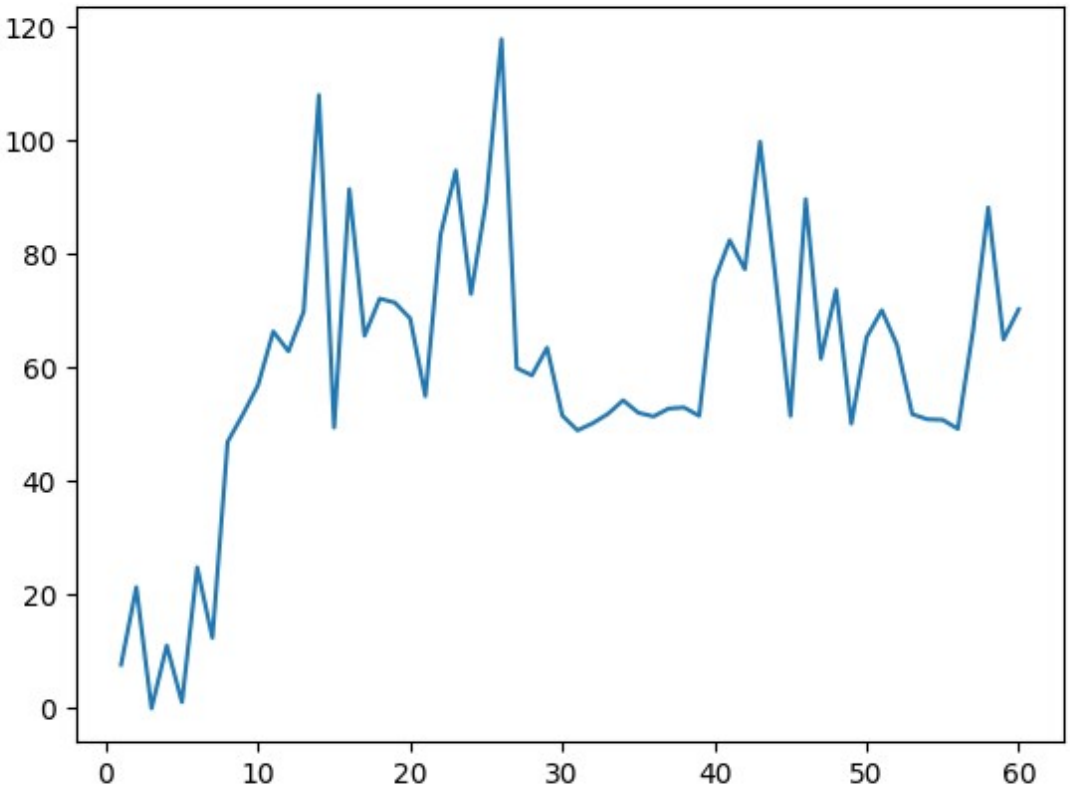
Custom reward function

I am giving reward based on the height climbed on the right side of the hill. The more height the car will climb the more reward it will get. This will encourage the car to take such actions so that it can climb more and more. I am also giving one bonus reward when the car is reached at the top. The

## Solving the environment

This environment also consists of discrete action space and continuous state space. I have used the same DQN algorithm with little change in network architecture. I have increased the size of the hidden layer and the rest is exactly the same.

Following is the plot showing rewards per episode. The car started to reach the goal position after around 10 episodes.




MountainCar reward vs episode plot

# Pendulum-v0

Pendulum trained agent

## About the environment

> *The inverted pendulum swingup problem is a classic problem in the control literature. In this version of the problem, the pendulum starts in a random position, and the goal is to swing it up so it stays upright.*

## State space (Continuous)

- `0` - Pendulum angle
- `1` - Pendulum speed

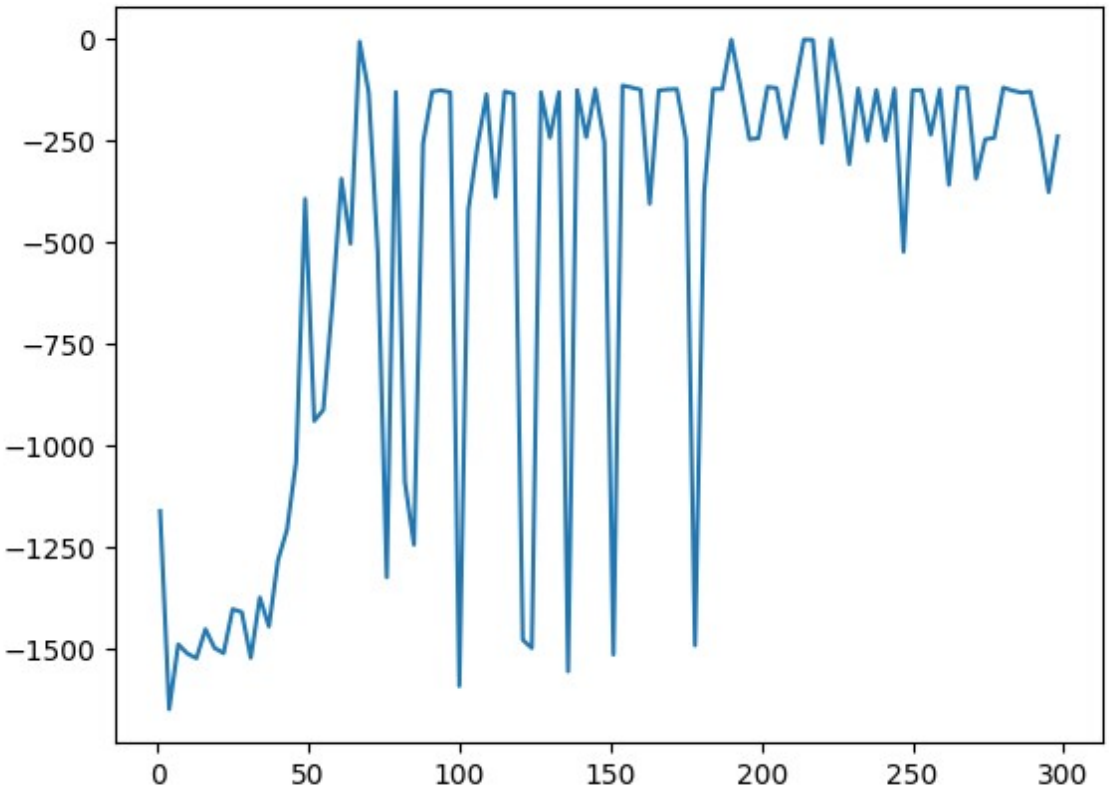
The default reward function depends on the angle of the pendulum. If the pendulum is upright, it will give maximum rewards. We do not need to change the default reward function here.

## Solving the environment

This problem is slightly different from the above two. Action space is continuous here. I am using the DDPG algorithm to solve this problem. DDPG works quite well when we have continuous state and state space.

In DDPG there are two 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.

Following is the plot showing rewards per episode. I was able to solve this environment in around 70 episodes.




Pendulum reward vs episode plot

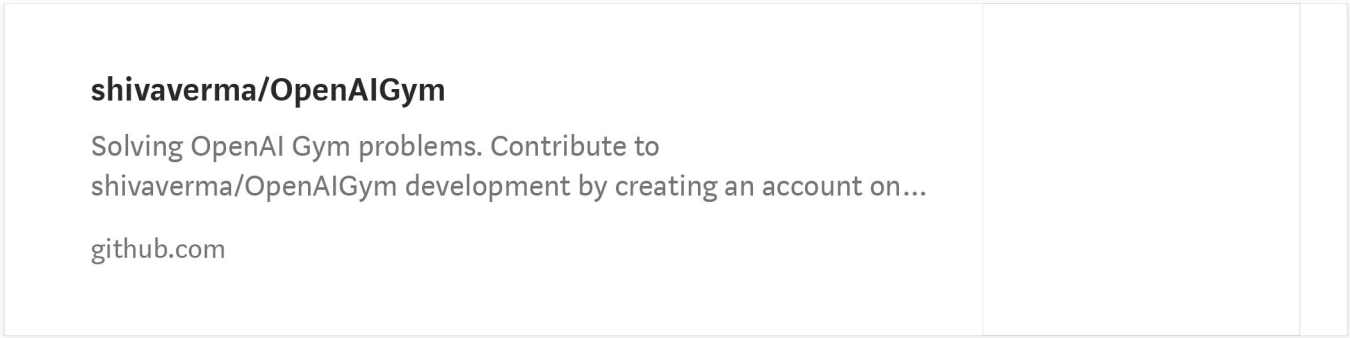
. . .

There are two more environments in classic control problems. I would highly recommend you to solve it and begin your journey in reinforcement learning. Following are the two environments.

- **Acrobot-v1**
- **MountainCarContinuous-v0**

## Resources

- This is an awesome introductory **blog** on Reinforcement Learning.
- I will highly recommend you to read the **paper** on DQN by Deepmind.
- Here is the **paper** on DDPG.
- This is a very helpful **blog** on DDPG.
- Read **this** doc to know how to use Gym environments.
- Check out other cool **environments** on OpenAIGym.
- Below is the link to my GitHub repository.




Reinforcement Learning Artificial Intelligence Machine Learning Deep Learning Openai Gym