DESCRIPTION OF THE IMPLEMENTATION

On this model, we will use DDPG with Ornstein–Uhlenbeck Action Noise and Fixed Target.

DEEP DETERMINISTIC POLICY GRADIENT (DDPG)

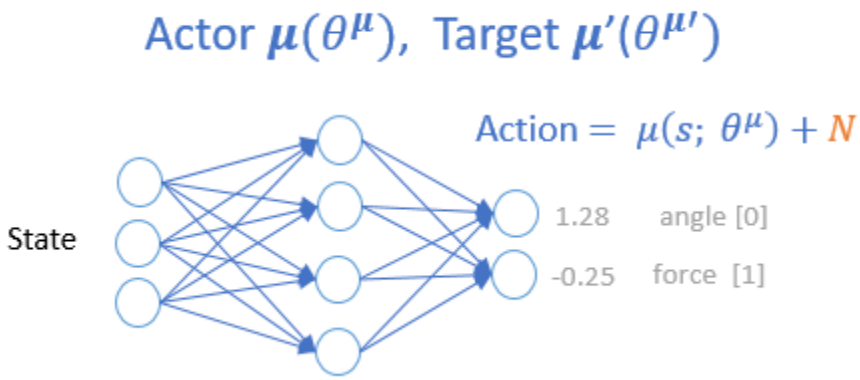
DDPG It is a good algorithm to use when we have continuous Action Space.

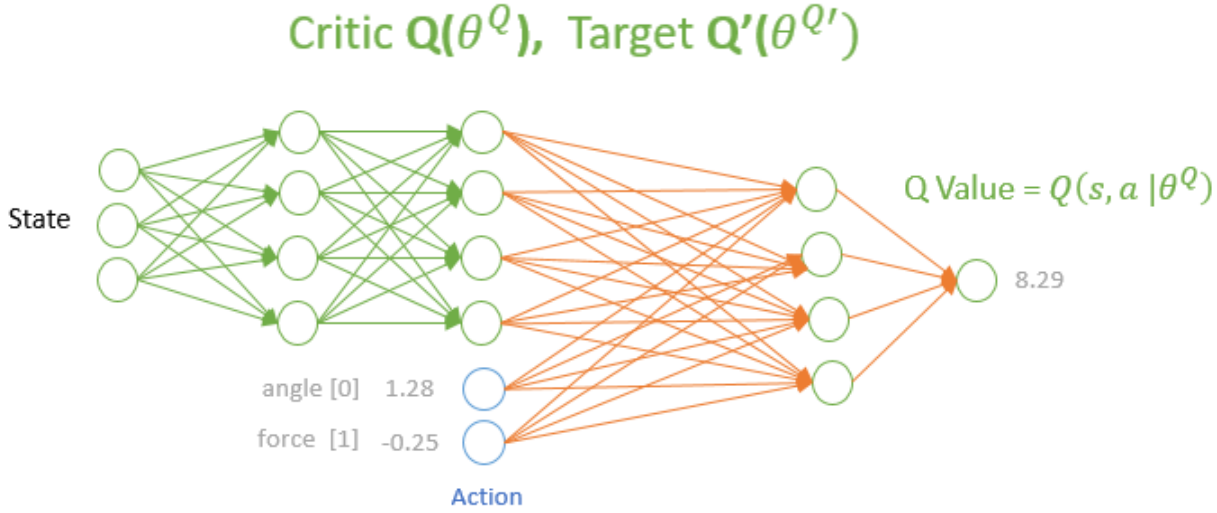
It is a Continuum Space because there are an infinite number of values for each Action

It is a Deterministic problem because for each state the policy will choose 1 action to take.

DDPG has 4 Deep Neural Networks.

- **Actor , Actor Target** : They will predict the best Action to take given a State "S"
- **Critic, Critic Target:** They will predict the Q-Value: Q(a, a) is the Total Discounted Reward that the agent can receive when he is on the state S and take the action A and follows the policy on the next steps.





- Actor: μ

  It will calculate the argmax $_a$ Q(s, a), the best action to take, given a State S

  θ^{μ} weights to be trained

  $\mu(s;\ \theta^{\mu})$ + N = Action (output of the Neural Network)

  N = noise in order to add exploration to the model

  Maximize J = $\frac{1}{M}\ \sum_i Q(s_i, \mu(s_i|\ \theta^{\mu})|\theta^{Q}) * \ \mu(s_i|\ \theta^{\mu})$

- Actor_Target: μ'

  Used to calculate a Fixed Target

  It will calculate the target value used to train the Critic Network

  (Target = True Value = y_i)

  $y_i = r_i + \ \Upsilon Q'(s_{i+1,}\ \mu'(s_{i+1}|\ \theta^{\mu\prime})|\ \theta^{Q\prime})$

  $\theta^{\mu\prime}$ won't be trained. We will transfer the parameters from the Actor to the Actor_Target Network using the Soft Update:

  ${\theta^{\mu\prime}}_{Actor\_Target} = (1-\tau){\theta^{\mu\prime}}_{Actor\_Target} + \tau\ {\theta^{\mu}}_{Actor}$ τ <<1

  Used for stability, once we train the parameters, we want to move the prediction in direction to the target. If we move the target and the prediction at the same time it will be very difficult to learn

- Critic: Q

  Will calculate the Q value, given a State_Action Pair

  θ^{Q}weights to be trained

  $Q(s, \mu(s;\ \theta^{\mu}); \theta^{Q})$ = Q value (Output of the Neural Network)

  Loss MSE = $\frac{1}{M}\ \sum_i (y_i - Q(s_i, a_i|\theta^{Q}))^2$

- Critic_Target: Q'

  Used to calculate a Fixed Target

  It will calculate the target value used to train the Critic Network

  (Target = True Value = y_i)

  $\mu'(s_{i+1}|\ \theta^{\mu\prime})$

  $\theta^{Q\prime}$ won't be trained. We will transfer the parameters from the Critic to the Critic_Target Network using the Soft Update:

  ${\theta^{Q\prime}}_{Critic\_Target} = (1-\tau){\theta^{Q\prime}}_{Critic\ \_Target} + \tau\ {\theta^{Q}}_{Critic}$ τ <<1

  Used for stability, once we train the parameters, we want to move the prediction in direction to the target. If we move the target and the prediction at the same time it will be very difficult to learn

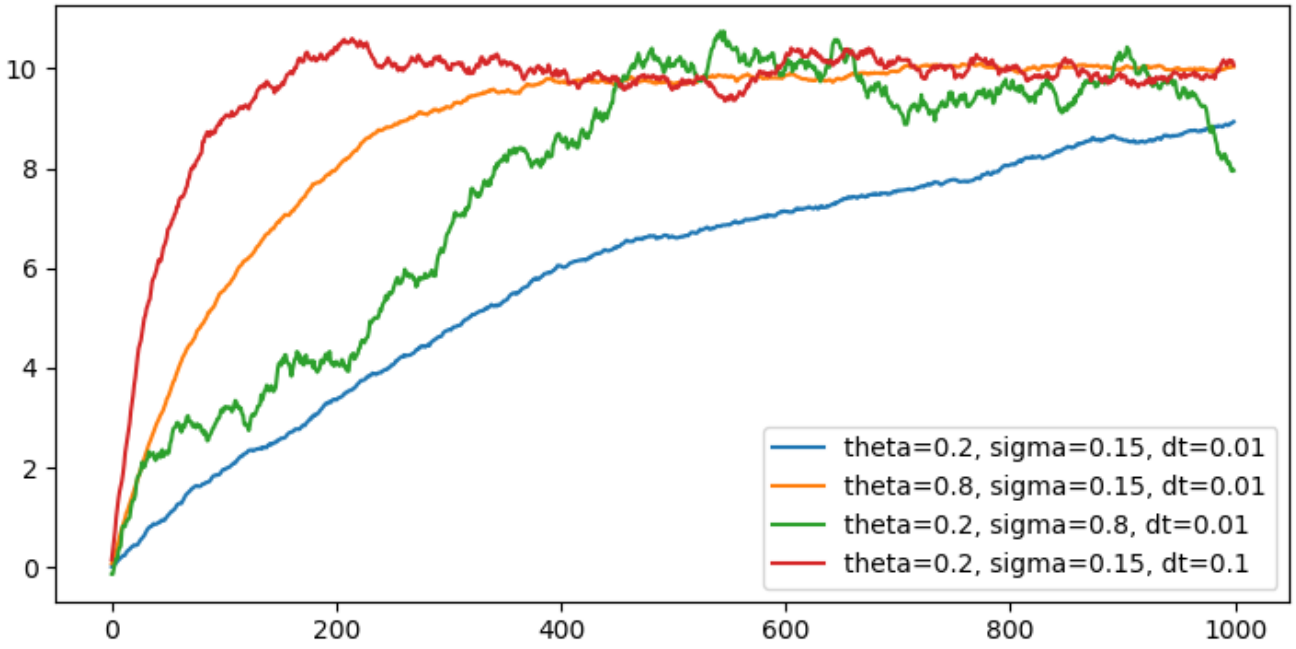
NOISE Ornstein–Uhlenbeck Action Noise

We need to add noise to the Action in order to make the agent explores the environment.

$$x = x\_prev + theta * (mu - x\_prev) * dt + sigma * \sqrt{dt} * random.Normal(size=mu.shape)$$

The noise is correlated in time. It will start with a value x0 and over time it will converge to mu.
dt will dictate how fast the noise will converge to mu. Sigma will be responsible for the volatility of the noise. On DDPG we will use: mu=0 and x0=0 for each component of the vector action [angle, force, etc.]

Example of different noises, x0=0 and mu=10, just to illustrate the behavior of the noise over time



FIXED TARGET

Fixed Target is important because when we update theta, if we don't have a fixed target, the network will change the value of the prediction (yhat = Q(s, a)) and the target (y = r + Q(s' ,a)). So, it will be very difficult for the model to decide how to change theta in order to move the prediction towards the target. The model won't be stable, the loss will swing a lot, and it will be very difficult to converge, find the thetas that minimize the loss.

But moving the Target very, very slowly, using the Soft Update, will avoid this problem and the model will be able to learn.

SOFT UPDATE

It will be used to update the weights of the Actor and Critic Networks
τ will be a very small number << 1

$$\theta^{\mu'}{}_{Actor\_Target} = (1-\tau)\theta^{\mu'}{}_{Actor\_Target} + \tau\,\theta^{\mu}{}_{Actor}$$
$$\theta^{Q'}{}_{Critic\_Target} = (1-\tau)\theta^{Q'}{}_{Critic\_Target} + \tau\,\theta^{Q}{}_{Critic}$$

HYPERPARAMETERS

n_actions = 4 # scalar, number of actions

action_bound = 1 # scalar, maximum absolute value of the action

state_dim = 33 # scalar, number of features on the state

fc1_dims = 400 # scalar, number of neurons on the 1st hidden layer

fc2_dims = 300 # scalar, number of neurons on the 2nd hidden layer

alfa = 0.0003 # scalar, learning rate of the Actor Network

beta = 0.0003 # scalar, learning rate of Critic Network

tau = 0.03 # scalar [0,1] Soft Update of Target Network: Actor and Critic << 1

gamma = 0.99 # scalar [0,1] Discount Rate for future rewards

batch_size = 32 # scalar, size of the mini batch

memory_size = 1e+5 # scalar, maximum number of experiences to store on Replay Buffer

episodic_task = False # Boolean, check if there is a terminal state

iteration = 1000 # number of iterations to train the model

steps = 6 # number of steps the agent will take before we train

train = 4 # number of times the agent will train before it moves to the next step

noise_decay = 0.97 # scalar, factor apply to reduce the noise at each episode

Noise: mu = 0, theta = 0.03, sigma = 0.03, dt = 0.01

FUTURE IMPLEMENTATIONS

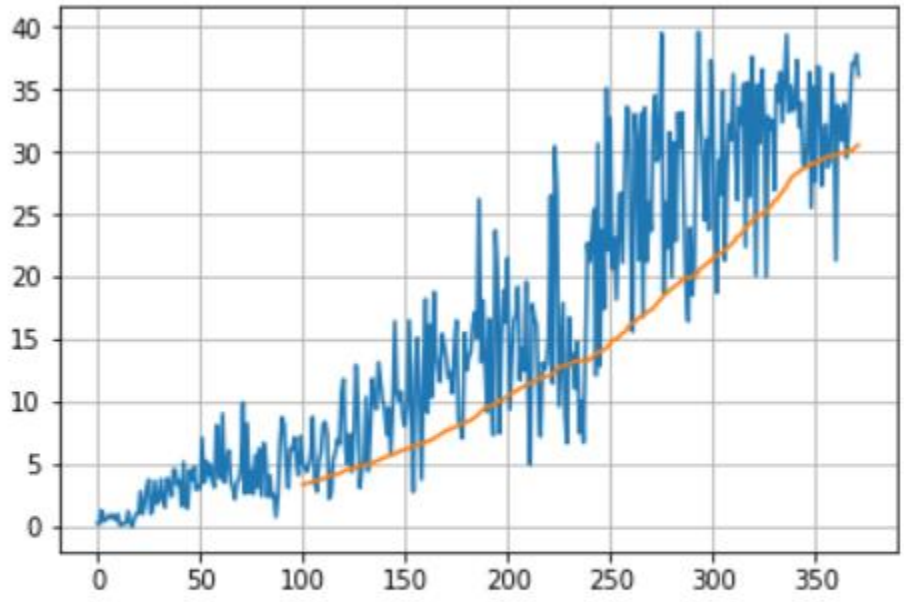
In order to make the agent learn faster and/or learn a better policy, we could use:

- Prioritize Experience Replay (PER)
  It will select the experiences that are more important than the others and it will help the agent to learn faster

- AC3 Asynchronous Advantage Actor Critic
  - In order to break the correlation, instead of using replay buffer, A3C will create multiply instances of the environment and the agent and collect experiences in parallel.

REWARDS

Rewards



Episode

```
number of episodes: 372
average reward 100 consecutive episodes:30.59
```