Continous Control - Controlling A Double-Jointed Arm With The Unity ML-Agents Reacher Environment

Howard Hyunjin Cho

ABSTRACT

The continuous action space are widely used to solve human-like tasks such as self driving car, robot arm. Many industry want to manipulate robot as human does. The original Deep Q-Network(DQN) algorithm only worked on discrete action space with low dimensional action space which the agent decides which distinct action to perform from a finite action set. The model-free, off-policy, actor-critic methods like DDPG, A2C, A3C, D4PG, and PPO are well known algorithms that used in high dimension and continuous action spaces. In this paper, I would like to focus on DDPG and D4PG for a continuous action space.

Keywords: DDPG, D4PG, DQN, model-free, off-policy, replay buffer, soft updates, target networks

INTRODUCTION

This project is used deep reinforcement learning to control a double-jointed arm to reach target locations. The algorithms that are used in this paper were deep deterministic policy gradients(DDPG) and deep distributed distributional deterministic policy gradients(D4PG).

The deep deterministic policy gradients(DDPG) is off-policy and model-free actor-critic methods. In DDPG, it is used two deep neural networks actor and critic. In Policy-based method, agents have high variance where agent can only learn a new data or on new observations. Off-policy are used so the agent can learn from historical data which in this case, experience replay memory is used. The model-free is directly taking a data from environment, as opposed to making its own prediction about the environment. The actor-critic methods are useful in DDPG. All we are trying to do in actor-critic method is to continue to reduce the high variance commonly associated with policy-based agents. By utilizing actor-critic with experience replay is prevalent in continuous control tasks.

The deep distributed distributional deterministic policy gradients(D4PG) can improve the accuracy of DDPG with the help of distributional approach for off-policy learning. It is a combination of DDPG and distributional reinforcement learning. It adopted to train a multi-agent action.

DDPG METHODS

In DDPG, we want the believed best action every single time we query the actor network. That is a deterministic policy. As shown in Figure1, the actor is basically learning the argmax a Q(S,a), which is the best action. Actor in DDPG is used to approximate the optimal policy deterministically. Therefore, output the best believed action for any given state. The critic learns to evaluate the optimal action value function by using the actors best believed action. Therefore, we use the actor which is an approximate maximize to calculate a new target value for training the action value function much in the way DQN does.

DDPG uses replay buffer and soft updates to the target networks. In DDPG, you have two copies of network weights for each network which are a regular for the actor, an irregular for the critic and a target for the actor and a target for the critic. The target networks are updated using a oft updates strategy. As shown in Figure2, a soft update strategy consists of slowly blending the regular network weights with the target network weights. Therefore, every time step the target network be 99.99% of the target network weights and only a 0.01% of regular network weights.

In figure5 Algorithm1 is about DDPG algorithm. In the first line initialize critic network with given weight Q network(θ^Q) and actor with given weight deterministic policy function θ^μ. Second line use

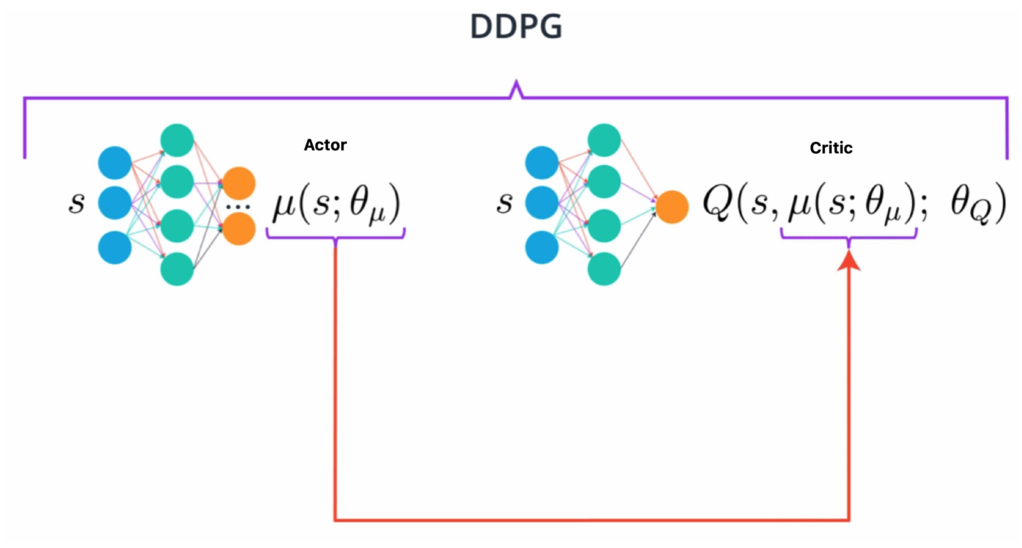

Figure 1. Actor-Critic Interaction

weight of target Q network, $\theta^{Q'}$ and weight of target policy network, $\theta^{\mu'}$ to do off-policy updates. The action-value function $Q^{\pi}(s,a)$(Q-function) can be described as recursive format by Bellman equation:

$$Q^{\pi}(s,a) = \underset{r,\ s'\sim E}{\mathbb{E}}[r(s,a) + \gamma \underset{a'\sim\pi}{\mathbb{E}}[Q^{\pi}(s',a')]]$$

If the target policy is deterministic we can describe it as a function $\mu : \mathrm{S} \leftarrow \mathrm{A}$ and avoid the inner expectation:

$$Q^{\mu}(s,a) = \underset{r,\ s'\sim E}{\mathbb{E}}[r(s,a) + \gamma Q^{\mu}(s',\mu(s'))]$$

The expectation depends only on the environment. This means that it is possible to learn Q^{μ} off-policy, using transitions which are generated from a different stochastic behavior policy β.

$$\mu(s) = argmax_a Q(s,a)$$

The first code is shown at Listing 1 Actor and Critic networks are for the deterministic policy network and the Q network. During the implementation, SELUs is used instead of ReLU which could have problem

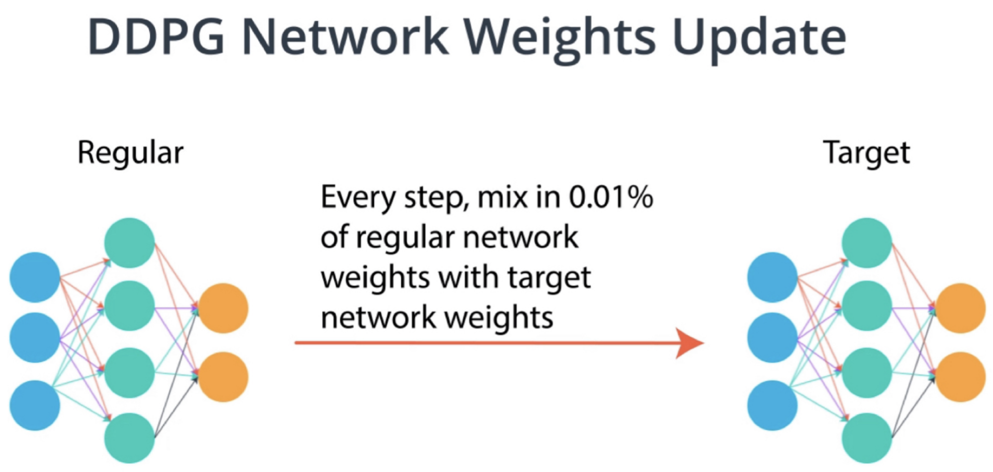
DDPG Network Weights Update
Regular
Target
Every step, mix in 0.01% of regular network weights with target network weights

Figure 2. regular target

with vanishing gradients. SELUs gives fast and better result with given same hyperparameters as shown below in figure 3 and figure 4.

$$f(a,x) = \lambda \begin{cases} \alpha(e^x - 1) & \text{for x} < 0 \\ x & \text{for x} \geq 0 \end{cases}$$

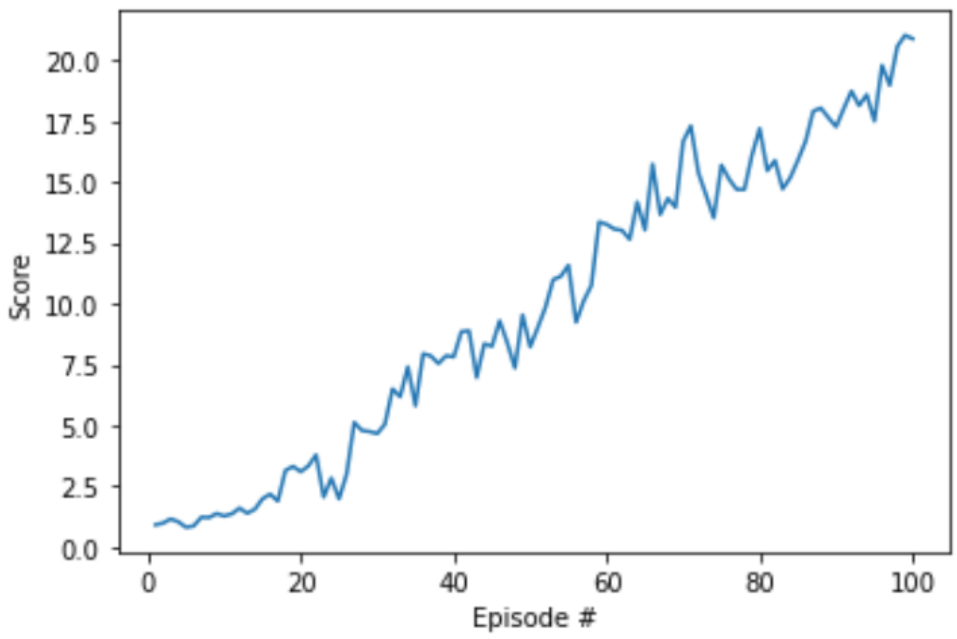

Figure 3. DDPG - ReLU Plot

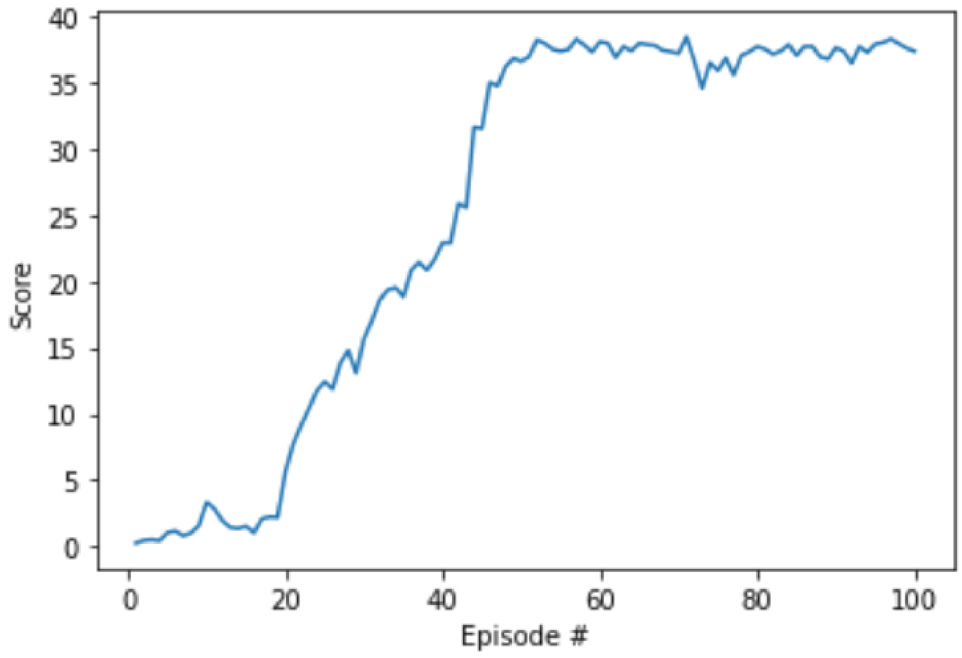

Figure 4. DDPG - SELUs Plot

We consider function approximators parameterized by θ^Q, which we optimize by minimizing the loss:

$$L(\theta^Q) = \mathop{\mathbb{E}}_{s\sim p^\beta,\ a_t\sim\beta,\ r_t\sim E}[(Q(s,a|\theta^Q) - y_t)^2]$$

where Q target,

$$y_t = r(s,a) + \gamma Q(s', \mu(s')|\theta^Q)$$

To calculate y_t it is necessary to use of a replay buffer and a separate target network. Replay buffer used in many other reinforcement learning algorithms to sample experience to update the parameters. The replay buffer contains a collection of experience tuples (S, A, R, S'). The tuples are gradually added to the buffer as we are interacting with environment. The replay buffer is used to break the correlation between immediate transitions in the episodes. The code indicates in the second code at Listing 2 Replay Buffer. DDPG optimizes the critic by minimizing the loss:

$$L(\theta^Q) = \frac{1}{N}\sum_i (y_i - (Q(s,a|\theta^Q))^2$$

Adding noise N can overcome deterministic policy gradient to explore the full state and action space:

$$\mu^{'}(s_t) = \mu(s_t|\theta_t^\mu) + N$$

Code shows at Listing 3 OUNoise. In the deterministic policy gradient, we want to maximize the rewards (Q-values) received over the sampled mini-batch where gradient is given as:

$$\nabla_{\theta^\mu} J \approx \mathop{\mathbb{E}}_{s\sim p^\beta}[\nabla_{\theta^\mu} Q(s,a|\theta^Q)|_{s=s,\ a=\mu(s|\theta^\mu)}]$$

by applying chain rule:

$$\nabla_{\theta^\mu} J \approx \mathop{\mathbb{E}}_{s\sim p^\beta}[\nabla_a Q(s,a|\theta^Q)|_{s=s,\ a=\mu(s)}\, \nabla_{\theta_\mu}\, \mu(s|\theta^\mu)|_{s=s}]$$

In the last line in Figure 5, the target networks are slowly updated(soft updates for both actor and critic. The target network values are constrained to change slowly, different from the design in DQN that the target network stays frozen some period of time. This above processes of the agent is indicated in code at the Listing 4 Agent.

Algorithm 1 DDPG algorithm

```
Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer $R$
for episode = 1, M do
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state $s_1$
  for t = 1, T do
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
    Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
    Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:
        $\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$
    Update the target networks:
        $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
        $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$
  end for
end for
```

Figure 5. DDPG Pseudocode

DDPG RESULT

Hyperparameters:

Buffer Size	1e6
Batch Size	384
GAMMA	0.99
TAU	1e-3
LR ACTOR	1e-4
LR CRITIC	1e-3
WEIGHT DECAY	0

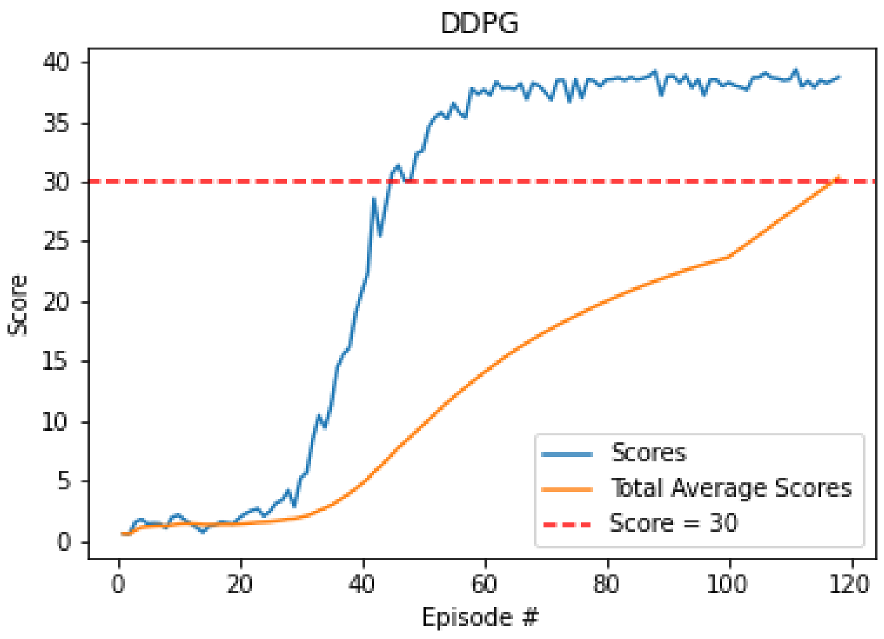

Figure 6. DDPG Plot

As the plot shown in Figure 6, the mean scores are already reaching at episode 45 and it took 14 minutes for training.

D4PG METHODS

Distributed Distributional DDPG algorithm (D4PG) is extend version of DDPG. It obtains state of the are performance accross a wide variety of control task, including hard manipulation. D4PG is the improvement of a distributional updates of the DDPG. As shown in Figure 7 D4PG Psudocode, there are some updated form DDPG algorithm. In policy gradient update from DDPG was

$$\bigtriangledown_\theta J \approx \underset{p^\mu}{\mathbb{E}}[\bigtriangledown_a Q_w(s,a) \bigtriangledown_\theta \mu_\theta(s)|_{a=\mu_\theta(s)}]$$

However, in D4PG:

$$\bigtriangledown_\theta J \approx \underset{p^\mu}{\mathbb{E}}[\mathbb{E}[\bigtriangledown_a Z_w(s,a) \bigtriangledown_\theta \mu_\theta(s)|_{a=\mu_\theta(s)}]]$$

where Z_w is a distribution parameterized by w. It is where the critic estimates the expected Q values as a random variable.

$$Q_w(s,a) = \mathbb{E}[Z_w(s,a)]$$

In the D4PG psudocode, it calculate the TD error in line six for construct the target distribution. This step computes N step TD target rather than one-step to incorporate rewards in more future steps:

$$y_i = r(s_0,a_0) + \mathbb{E}[\sum_{n=1}^{N-1} r(s_n,a_n) + \gamma^N Q(s_N, \mu_\theta(s_N))|s_0,a_0]$$

```
Algorithm 1 D4PG
Input: batch size $M$, trajectory length $N$, number of actors $K$, replay size $R$, exploration constant
       $\epsilon$, initial learning rates $\alpha_0$ and $\beta_0$
 1: Initialize network weights $(\theta, w)$ at random
 2: Initialize target weights $(\theta', w') \leftarrow (\theta, w)$
 3: Launch $K$ actors and replicate network weights $(\theta, w)$ to each actor
 4: for $t = 1, \dots, T$ do
 5:     Sample $M$ transitions $(\mathbf{x}_{i:i+N}, \mathbf{a}_{i:i+N-1}, r_{i:i+N-1})$ of length $N$ from replay with priority $p_i$
 6:     Construct the target distributions $Y_i = \left(\sum_{n=0}^{N-1} \gamma^n r_{i+n}\right) + \gamma^N Z_{w'}(\mathbf{x}_{i+N}, \pi_{\theta'}(\mathbf{x}_{i+N}))$
        Note, although not denoted the target $Y_i$ may be projected (e.g. for Categorical value distributions).
 7:     Compute the actor and critic updates
            $\delta_w = \frac{1}{M} \sum_i \nabla_w (R p_i)^{-1} d(Y_i, Z_w(\mathbf{x}_i, \mathbf{a}_i))$
            $\delta_\theta = \frac{1}{M} \sum_i \nabla_\theta \pi_\theta(\mathbf{x}_i) \, \mathbb{E}[\nabla_{\mathbf{a}} Z_w(\mathbf{x}_i, \mathbf{a})]\big|_{\mathbf{a}=\pi_\theta(\mathbf{x}_i)}$
 8:     Update network parameters $\theta \leftarrow \theta + \alpha_t \, \delta_\theta$, $w \leftarrow w + \beta_t \, \delta_w$
 9:     If $t = 0 \mod t_{\text{target}}$, update the target networks $(\theta', w') \leftarrow (\theta, w)$
10:     If $t = 0 \mod t_{\text{actors}}$, replicate network weights to the actors
11: end for
12: return policy parameters $\theta$

Actor
 1: repeat
 2:     Sample action $\mathbf{a} = \pi_\theta(\mathbf{x}) + \epsilon \mathcal{N}(0, 1)$
 3:     Execute action $\mathbf{a}$, observe reward $r$ and state $\mathbf{x}'$
 4:     Store $(\mathbf{x}, \mathbf{a}, r, \mathbf{x}')$ in replay
 5: until learner finishes
```

Figure 7. D4PG Pseudocode

D4PG RESULT

Figure 8 plot for Hyperparameters:

Buffer Size	1e6
Batch Size	384
GAMMA	0.99
TAU	1e-3
LR ACTOR	1e-4
LR CRITIC	1e-3
WEIGHT DECAY	0
REWARD STEPS	5
Max V	10
Min V	-10
N of Atoms	51

As shown in Figure 8, the scores reach 30 much faster than DDPG. It reaches to around 12 minutes. It approximately increases performance 20 percentage.

Figure 9 plot for Hyperparameters:

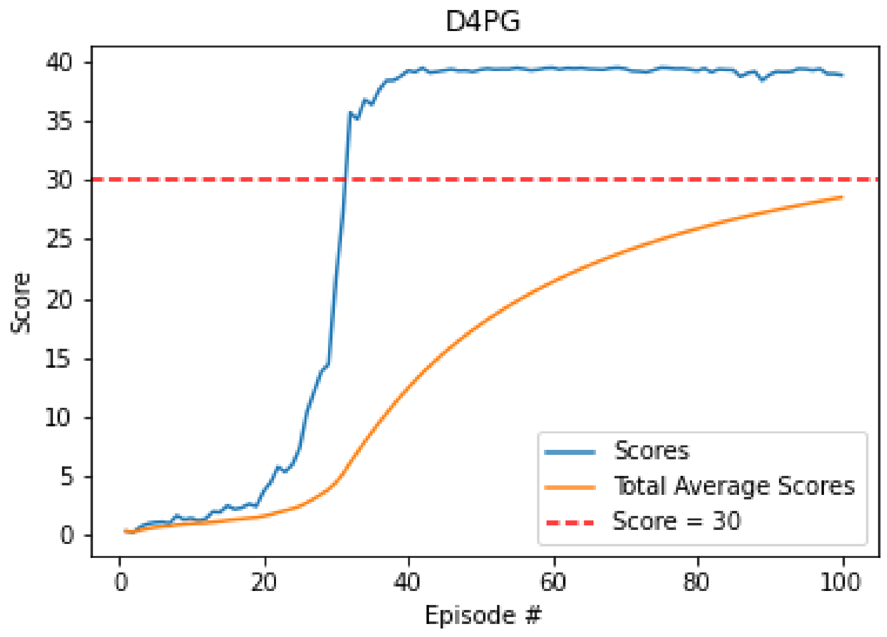

Figure 8. D4PG Plot Applying Same Hyperparameters

Buffer Size	1e6
Batch Size	384
GAMMA	0.99
TAU	1e-3
LR ACTOR	1e-3
LR CRITIC	1e-3
WEIGHT DECAY	0
REWARD STEPS	5
Max V	10
Min V	-10
N of Atoms	51

Improvement Suggestion

I would like to implement A2C which works well on CPU. Also A3C to see how the performance different compare to D4PG and DDPG.

REFERENCES

Barth-Maron, G., Hoffman, M. W., Budden, D., Dabney, W., Horgan, D., TB, D., Muldal, A., Heess, N., and Lillicrap, T. (2018). Distributed distributional deterministic policy gradients.

Böhm, T. (2018). An introduction to selus and why you should start using them as your activation functions.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2015). Continuous control with deep reinforcement learning.

Lillicrap et al. (2015) Böhm (2018) Barth-Maron et al. (2018)

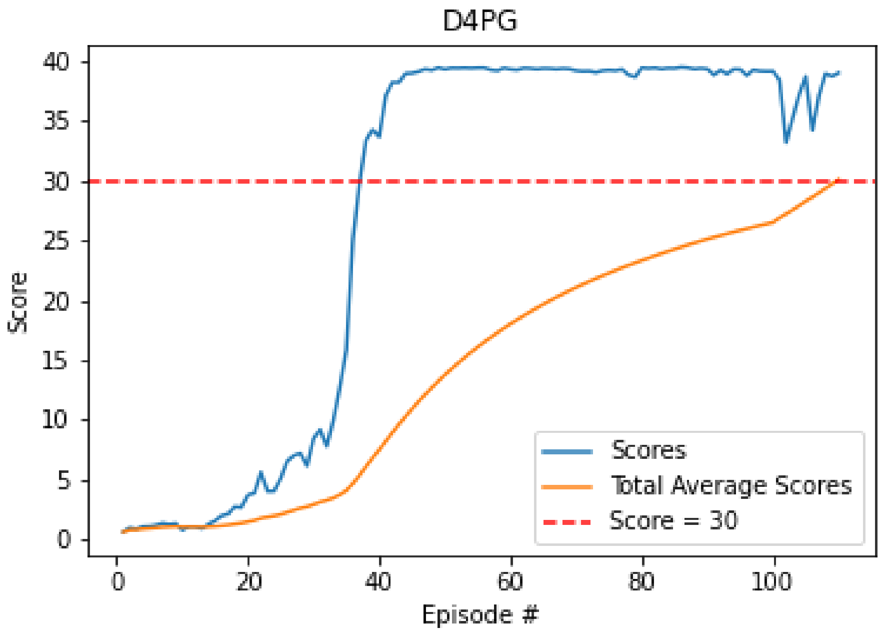

Figure 9. D4PG Plot

```python
class Actor(nn.Module):
    """Actor (Policy) Model."""

    def __init__(self, state_size, action_size, seed, fc1_units=400, fc2_units=300):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """
        super(Actor, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units).to(ddpg_agent.device)
        self.fc2 = nn.Linear(fc1_units, fc2_units).to(ddpg_agent.device)
        self.fc3 = nn.Linear(fc2_units, action_size).to(ddpg_agent.device)
        self.reset_parameters()

    def reset_parameters(self):
        self.fc1.weight.data.uniform_(*hidden_init(self.fc1))
        self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
        self.fc3.weight.data.uniform_(-3e-3, 3e-3)

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


class Critic(nn.Module):
    """Critic (Value) Model."""

    def __init__(self, state_size, action_size, seed, fc1_units=400, fc2_units=300):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in the first hidden layer
            fc2_units (int): Number of nodes in the second hidden layer
        """
        super(Critic, self).__init__()
        self.seed = torch.manual_seed(seed)

        self.fc1 = nn.Linear(state_size, fc1_units).to(ddpg_agent.
device)
        self.fc2 = nn.Linear(fc1_units+action_size, fc2_units).to(
    ddpg_agent.device)
        self.fc3 = nn.Linear(fc2_units, 1).to(ddpg_agent.device)
        self.reset_parameters()

    def reset_parameters(self):
        self.fc1.weight.data.uniform_(*hidden_init(self.fc1))
        self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
        self.fc3.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.selu(self.fc1(state))
        x = torch.cat((xs, action), dim=1)
        x = F.selu(self.fc2(x))
        return self.fc3(x)
```

Listing 1: Actor and Critic networks

```
class ReplayBuffer:
    """Fixed-size buffer to store experience tuples."""

    def __init__(self, action_size, buffer_size, batch_size, seed):
        """Initialize a ReplayBuffer object.
        Params
        ======
            buffer_size (int): maximum size of buffer
            batch_size (int): size of each training batch
        """
        self.action_size = action_size
        self.memory = deque(maxlen=buffer_size)  # internal memory
    (deque)
        self.batch_size = batch_size
        self.experience = namedtuple("Experience", field_names=["
    state", "action", "reward", "next_state", "done"])
        self.seed = random.seed(seed)

    def add(self, state, action, reward, next_state, done):
        """Add a new experience to memory."""
        e = self.experience(state, action, reward, next_state, done
    )
        self.memory.append(e)

    def sample(self):
        """Randomly sample a batch of experiences from memory."""
        experiences = random.sample(self.memory, k=self.batch_size)

        states = torch.from_numpy(np.vstack([e.state for e in
    experiences if e is not None])).float().to(device)
        actions = torch.from_numpy(np.vstack([e.action for e in
    experiences if e is not None])).float().to(device)
        rewards = torch.from_numpy(np.vstack([e.reward for e in
    experiences if e is not None])).float().to(device)
        next_states = torch.from_numpy(np.vstack([e.next_state for
    e in experiences if e is not None])).float().to(device)
```

```python
        dones = torch.from_numpy(np.vstack([e.done for e in experiences if e is not None]).astype(np.uint8)).float().to(device)

        return (states, actions, rewards, next_states, dones)

    def __len__(self):
        """Return the current size of internal memory."""
        return len(self.memory)
```

Listing 2: Replay Buffer

```python

class OUNoise:
    """Ornstein-Uhlenbeck process."""

    def __init__(self, size, seed, mu=0., theta=0.15, sigma=0.2):
        """Initialize parameters and noise process."""
        self.mu = mu * np.ones(size)
        self.theta = theta
        self.sigma = sigma
        self.seed = random.seed(seed)
        self.reset()

    def reset(self):
        """Reset the internal state (= noise) to mean (mu)."""
        self.state = copy.copy(self.mu)

    def sample(self):
        """Update internal state and return it as a noise sample."""
        x = self.state
        dx = self.theta * (self.mu - x) + self.sigma * np.array([np.random.randn() for i in range(len(x))])
        self.state = x + dx
        return self.state
```

Listing 3: OUNoise

```python

class Agent():
    """Interacts with and learns from the environment."""

    def __init__(self, state_size, action_size, random_seed):
        """Initialize an Agent object.

        Params
        ======
            state_size (int): dimension of each state
            action_size (int): dimension of each action
            random_seed (int): random seed
        """
        self.state_size = state_size
        self.action_size = action_size
        self.seed = random.seed(random_seed)

        # Actor Network (w/ Target Network)
        self.actor_local = Actor(state_size, action_size, random_seed).to(device)
        self.actor_target = Actor(state_size, action_size, random_seed).to(device)
        self.actor_optimizer = optim.Adam(self.actor_local.parameters(), lr=LR_ACTOR)

        # Critic Network (w/ Target Network)
        self.critic_local = Critic(state_size, action_size, random_seed).to(device)
        self.critic_target = Critic(state_size, action_size, random_seed).to(device)
        self.critic_optimizer = optim.Adam(self.critic_local.parameters(), lr=LR_CRITIC, weight_decay=WEIGHT_DECAY)

        # Noise process
        self.noise = OUNoise(action_size, random_seed)

        # Replay memory
        self.memory = ReplayBuffer(action_size, BUFFER_SIZE, BATCH_SIZE, random_seed)


    def step(self, states, actions, rewards, next_states, dones, t, num_learn):
        """Save experience in replay memory, and use random sample from buffer to learn."""
        # Save experience / reward
        # collect multiple agent to learn
        for state, action, reward, next_state, done in zip(states, actions, rewards, next_states, dones):
                self.memory.add(state, action, reward, next_state, done)

        # Learn, if enough samples are available in memory
        if len(self.memory) > BATCH_SIZE and t%num_learn == 0:
            experiences = self.memory.sample()
            for _ in range(num_learn):
                experiences = self.memory.sample()
                self.learn(experiences, GAMMA)

    def act(self, state, add_noise=True):
        """Returns actions for given state as per current policy."""
        state = torch.from_numpy(state).float().to(device)
        self.actor_local.eval()
        with torch.no_grad():
            action = self.actor_local(state).cpu().data.numpy()
        self.actor_local.train()
        if add_noise:
            action += self.noise.sample()
        return np.clip(action, -1, 1)

    def reset(self):
        self.noise.reset()

    def learn(self, experiences, gamma):
        """Update policy and value parameters using given batch of experience tuples.
        Q_targets = r +    * critic_target(next_state, actor_target(next_state))
        where:
            actor_target(state) -> action
            critic_target(state, action) -> Q-value
        Params
        ======
            experiences (Tuple[torch.Tensor]): tuple of (s, a, r, s', done) tuples
            gamma (float): discount factor
        """
        states, actions, rewards, next_states, dones = experiences

        # ---------------------------- update critic ---------------------------- #
        # Get predicted next-state actions and Q values from target models
        next_actions = self.actor_target(next_states)
        Q_targets_next = self.critic_target(next_states, next_actions)

        # Compute Q targets for current states (y_i)
        Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))

        # Compute critic loss
        Q_expected = self.critic_local(states, actions)
        critic_loss = F.mse_loss(Q_expected, Q_targets)

        # Minimize the loss
        self.critic_optimizer.zero_grad()
        critic_loss.backward()
        self.critic_optimizer.step()

        # ---------------------------- update actor ---------------------------- #
        # Compute actor loss
        pred_actions = self.actor_local(states)
        actor_loss = -self.critic_local(states, pred_actions).mean() #make sure use negative

        # Minimize the loss
        self.actor_optimizer.zero_grad()
        actor_loss.backward()
        self.actor_optimizer.step()

        # ----------------------- update target networks ----------------------- #
        self.soft_update(self.critic_local, self.critic_target, TAU)
        self.soft_update(self.actor_local, self.actor_target, TAU)

    def soft_update(self, local_model, target_model, tau):
        """Soft update model parameters.
          _target  =   * _local  + (1 -   )* _target
        Params
        ======
            local_model: PyTorch model (weights will be copied from)
            target_model: PyTorch model (weights will be copied to)
            tau (float): interpolation parameter
        """
        for target_param, local_param in zip(target_model.parameters(), local_model.parameters()):
            target_param.data.copy_(tau*local_param.data + (1.0-tau)*target_param.data)
```
Listing 4: Agent