DRL-Continuous-Control - DDPG Learning Algorithm - Description of the Implementation.

Formally, a Q function was used to find an optimal policy, by selecting the best action in every state, or the action with the largest Q value.

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

When dealing with complex problems and when the agent obtains the observation from the environment and needs to decide about the next action, policies are more attractive than value functions because on optimal policy can be found directly without having to worry about values at all. This is especially the case in environments with extreme number of actions or with a continuous action space. That is in part because an action chosen based on the highest value is not necessarily always optimal.

Differences Summary: Value-based vs Policy-based vs Actor-Critic

Value-based methods:

- estimate the value function
- policy is implicit (e.g. ε-greedy)

Policy-based methods:

- estimate the policy
- no value function – it directly learns the optimal policy, without having to maintain a separate value function estimate

Actor-Critic methods:

- estimate the policy
- estimate the value function

Policy Based Methods

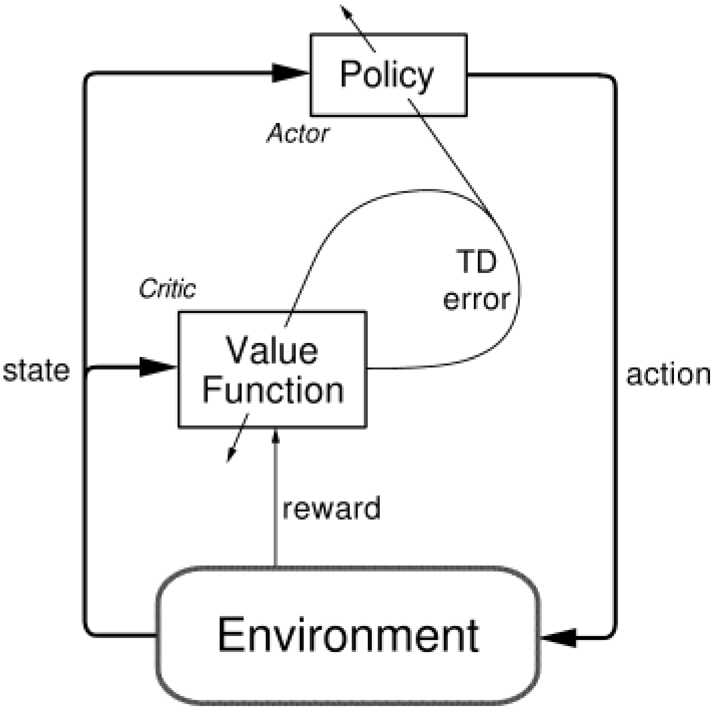
In an environment with a large action space, discrete or continuous, this optimization problem becomes hard, and a reason to consider policy-based methods.

Policy-based methods offer simplicity because they are a class of algorithms that search directly for the optimal policy, without storing additional data (i.e. action value estimates) that may not be useful. Policy-based methods can also learn both stochastic or deterministic policies, are well suited for finite, continuous and high-dimensional action spaces and have better convergence properties.

The disadvantages of these methods include typical convergence to a local rather than global optimum, and high variance policy evaluation.

Actor- Critic Methods

Actor policy function specifies an action given the current state of the environment and the critic value function produces a TD (Temporal-Difference) error signal given the state and resultant reward, to criticize the actions made by the actor.



Value-based methods use a deep neural network to estimate the value function.

A policy-based method agent uses a neural network to approximate the policy, without having to maintain a separate value function estimate. This agent parameterizes the policy and learns to optimize it directly.

A value function can be used as a baseline to reduce the variance of the policy-based agents. That is exactly what actor-critic methods are trying to use, value-based techniques to further reduce the variance of policy-based methods.

Actor- Critic methods are a hybrid between the value-based methods such as DQN and policy-based algorithms such as Reinforce and implement generalised policy iteration - alternating between a policy **evaluation** and a policy **improvement** step:

- **actor improvement** which aims at improving the current policy
- **critic evaluation,** which evaluates the current policy if the critic is modelled by a bootstrapping method, reduces the variance so the learning is more stable than pure policy gradient methods.

Estimating Expected Returns - Bias & Variance

Monte-Carlo estimate	**TD** estimate & **Dynamic Programming**
High Variance (means more samples are required to achieve the same degree of learning compared to TD)	**Low Variance** (because they are compounding a single timestep of randomness, instead of a full episode rollout)
Non-Biased (using true rewards, no estimates - given lots of data, estimates are accurate)	**Biased** (because next state estimates are not true values, adding bias into calculations)

Figure 1 Estimating Expected Returns

TD appears to learn faster, but it has trouble converging.

In machine learning, when we are training, we want to avoid high bias and high variance. What is desired, is both low bias and low variance, which is very hard to achieve.

When we are trying to estimate the value function or policies from returns, we have to consider the bias – variance trade-off. A return is computed from an entire episode, or trajectory. Value functions, on the other hand, use the expected return, which imply estimation and therefore also bias. While training, an agent is trying to find policies that maximize the total expected reward, limited to only sampling the environment. Therefore, we can only estimate the expectations, while trying to reduce the variance and keep bias to a minimum.

The actor or policy-based approach roughly learns this way:
The agent executes a few episodes and then learns. The policy/strategy is to do more of the episodes with good outcome/rewards and less of the rest of them.

After many executions, repeating this strategy/process, the probability of actions that lead to a good outcome will have increased, while the probability of actions that lead to less/ or negative rewards will have decreased. The problem with this approach is that it is inefficient, as it needs lots of data to lean a useful policy. It is inefficient because many actions that occur within the episodes with bad outcome/negative rewards could have been good very good actions. Decreasing the probability of the good actions taken, just because of the final outcome is not the best idea. This procedure works, because, repeating it infinitely many times, a good policy would surface, but at the cost of slow learning and high variance.

The critic or the value-based approach learns differently. Even before an episode starts or as the episode unfolds, the agent, guesses the final outcome/reward. At the beginning, the guesses are wrong/off. With time and experience, guesses become better and better and the agent is able to distinguish between good and bad situations or actions. The better the distinction and action selection, the better the performance.

Overall, this approach is not perfect either, as guesses introduce bias. In the beginning they are wrong because of lack of experience, and they can over or under-estimate. On the other hand, guesses are more consistent over time, reason why TD estimates have a lower variance.

Policy-based agents learn to act, while a value-based agent learns to estimate situations and actions. Combining a policy-based ‘actor’ agent and a value-based ‘critic’ agent often yields better results. Actor-critic agents learn to adjust the probabilities of good and bad actions, using the ‘actor’ alone, but a ‘critic’ is able to tell ‘good’ from ‘bad’ actions and speed-up learning.

As a result, actor-critic agents are more stable than value-based agents and need fewer samples than policy-based ones.

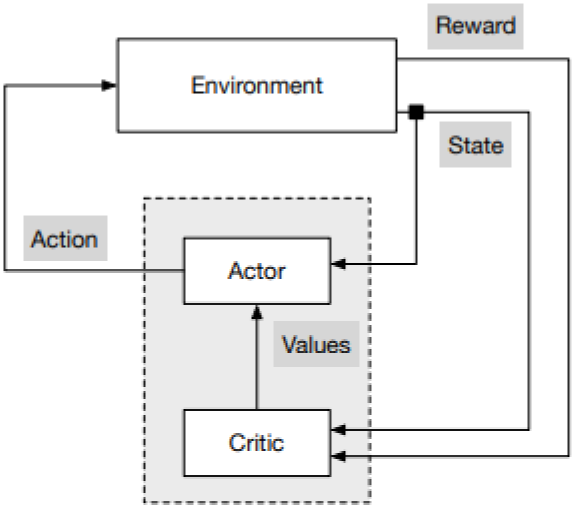


Figure 2 Actor-critic architecture diagram

If Deep Q Learning, which only works with discrete actions, is modified to work with continuous actions, the result would be DDPG.

DDPG

Deep Deterministic Policy Gradients (DDPG) is **off-policy** and **model-free.**

It is **off-policy** because it learns offline by gathering experiences collected from environment agents and sampling experiences from large Replay Memory Buffer. It is model-free because it does not learn a model of the environment, or it does not use the transition probability distribution (and the reward function) associated with the Markov decision process (MDP). It is a “trial and error” algorithm.

DDPG Algorithm

1. The Actor and the Critic use separate neural network, two copies of network weights for each network, (Actor: regular and target) and (Critic: regular and target).

2. The **Actor network** takes as input the **state s** and the network learning **weights** and produces as output the **action a**:

$$a = \mu(s|\Theta^{\mu})$$

3. The Critic network **$Q(s, a| \Theta^{Q})$** which takes an input as a **state s** and **action a** and returns the ***Q*** value where **Θ^{Q}** are the weights. The critic learns to evaluate the optimal action value function by using the actors best believed action.

4. Target network for both the Actor network **μ'** and Critic network **Q'**, use the **$\Theta^{\mu'}$** and **$\Theta^{Q'}$** weights of the target Actor and Critic network respectively.

5. Exploration noise **N** is added to the action produced by the Actor since the policy us deterministic:

$$\mu(s|\Theta^{\mu}) + N$$

6. Action **a_t** in state **s_t** is used and a reward **r_t** is received moving to a new state **s_{t+1}.** This transition information is added to an experience replay buffer **R**.

7. Similarly to the **DQN** algorithm, a minibatch of **N** transitions (**s_i, a_i, r_i, s_{i+1}**) are sampled from the replay buffer **R** and used to train the network, and predict the target ***Q'*** value:

$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\Theta^{\mu'})| \Theta^{Q'})$$

8. Then, use the TD error loss **L** computed as:

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

to update the Critic networks

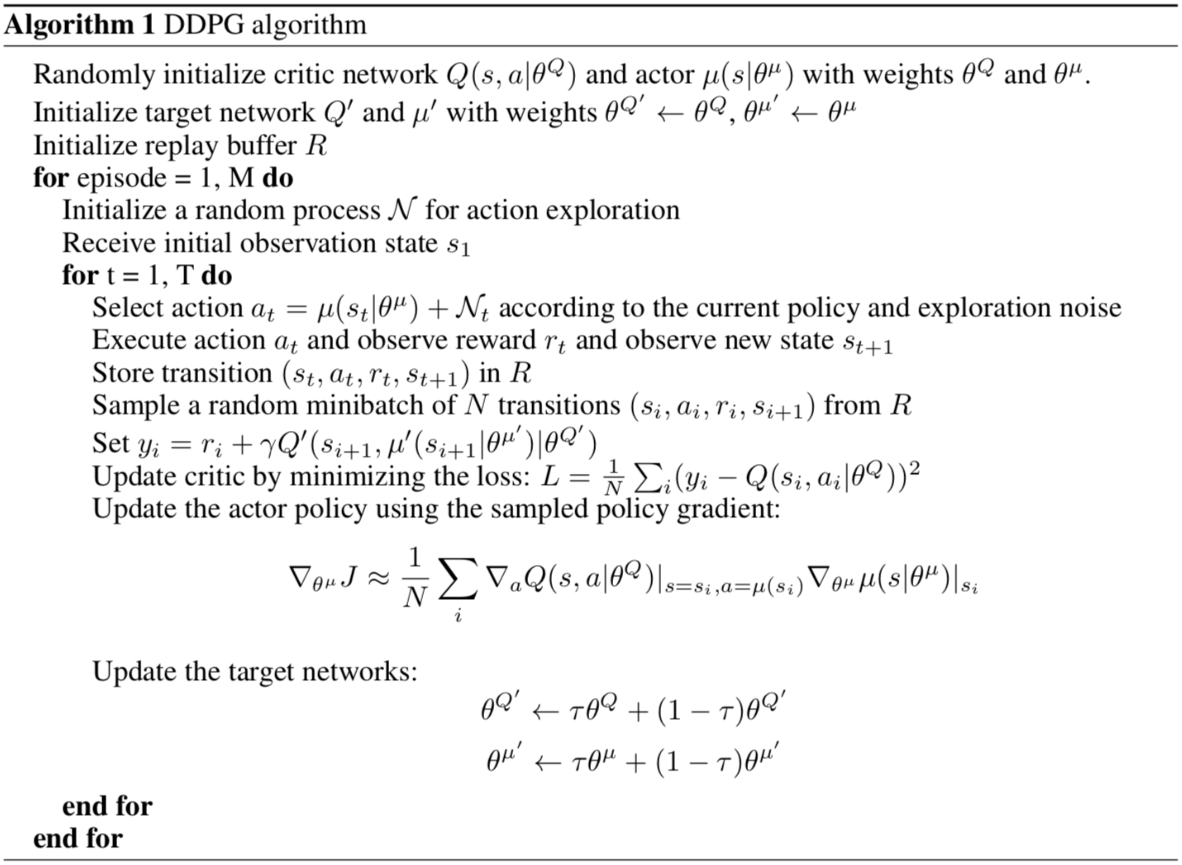
9. The Actor policy weights are updated using the sampled policy gradient.

10. We update the weights of both target networks (Agent, Critic) “softly” for greater model stability, by mixing the original weights with a small fraction of the target ones:

$$\theta^{Q'} \leftarrow \tau\theta^{Q} + (1 - \tau)\theta^{Q'}$$

$$\theta^{\mu'} \leftarrow \tau\theta^{\mu} + (1 - \tau)\theta^{\mu'}$$

DDPG algorithm pseudocode:



Algorithm 1 DDPG algorithm

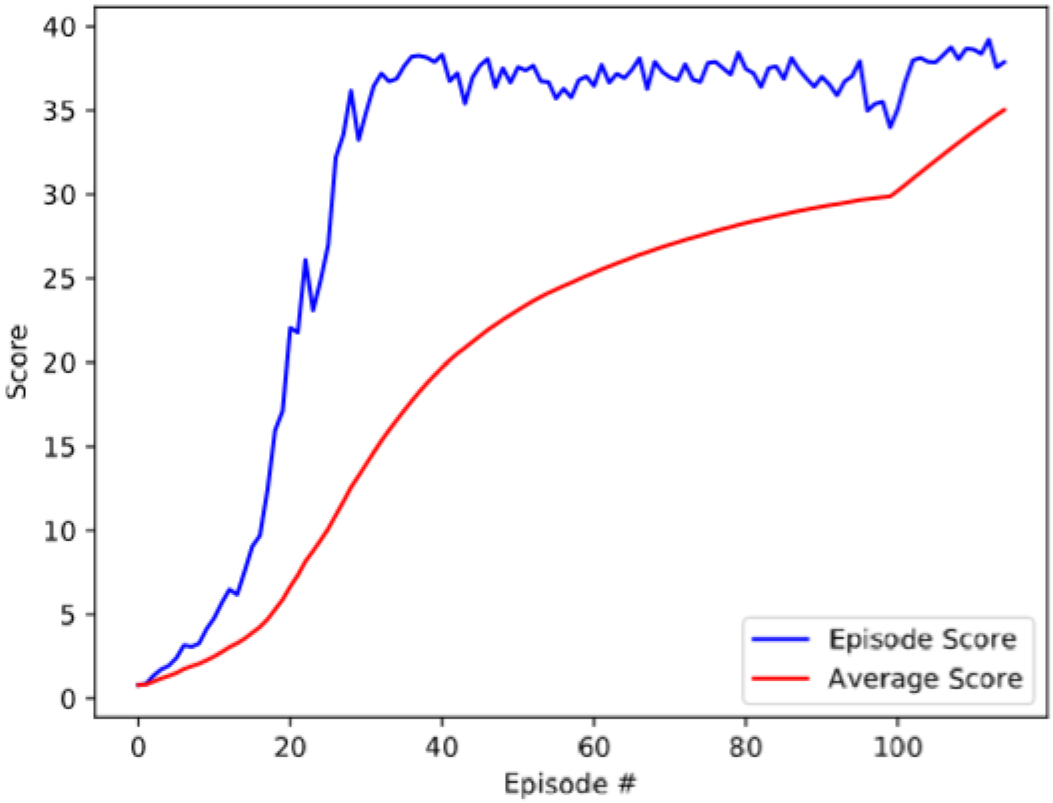
```
Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer $R$
for episode = 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
```

Hyper-Parameters

BUFFER_SIZE	1e6	replay buffer size
BATCH_SIZE	256	minibatch size
GAMMA	0.99	discount factor
TAU	1e-3	soft update of target parameters
LR_ACTOR	2e-4	learning rate of the actor
LR_CRITIC	2e-3	learning rate of the critic
UPDATE_EVERY	4	
ALT_UPDATE_EVERY	10	parameter for switching between ‘updating’ and ‘learning’ mode

NUM_UPDATES	4	number of samples to learn in ‘learning’ mode
ALPHA	0.1	
EPS_START	1.0	Epsilon start
EPS_END	0.01	Epsilon end
EPS_DECAY	2e-7	Epsilon decay per learn step

Training Results

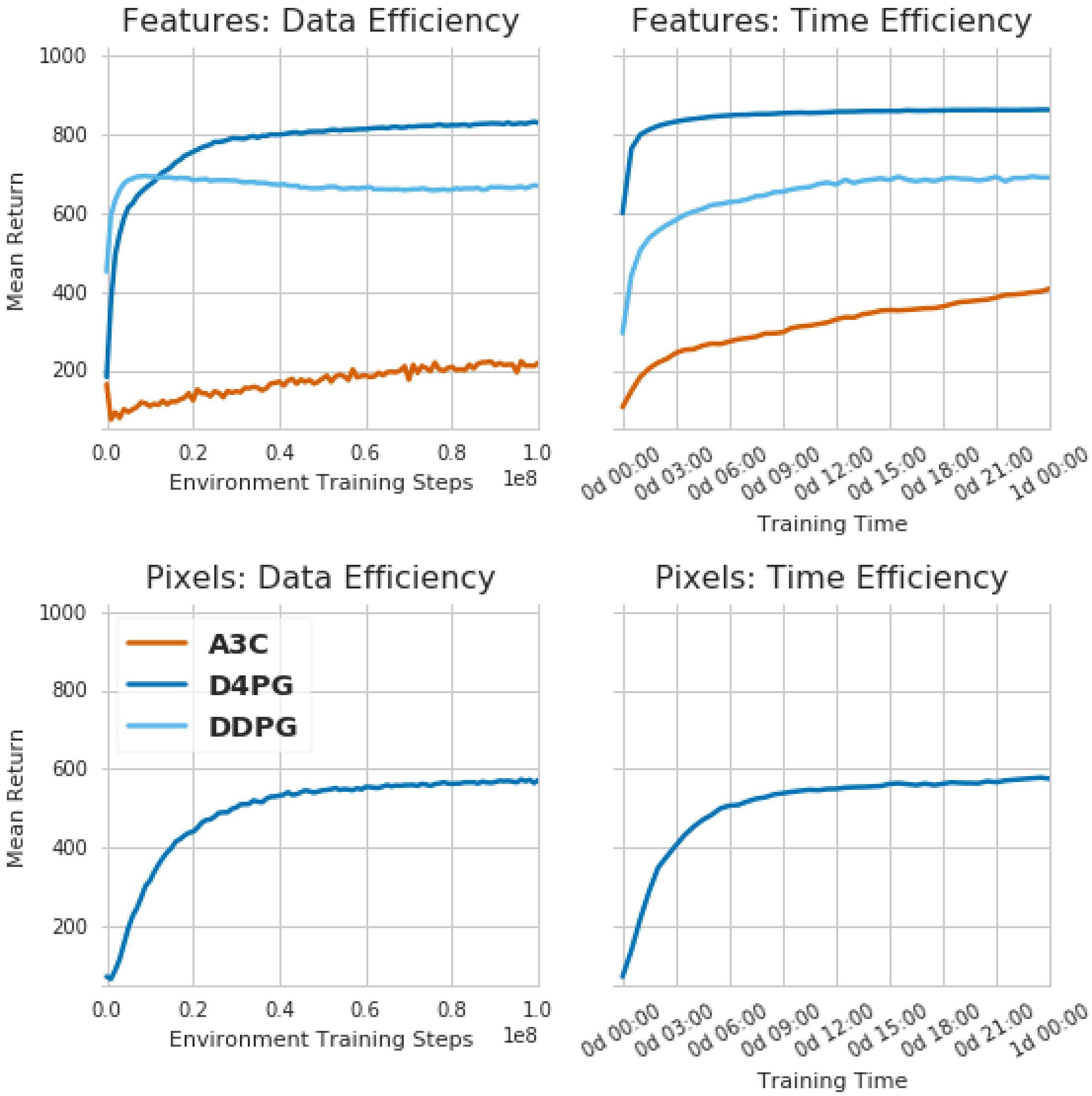


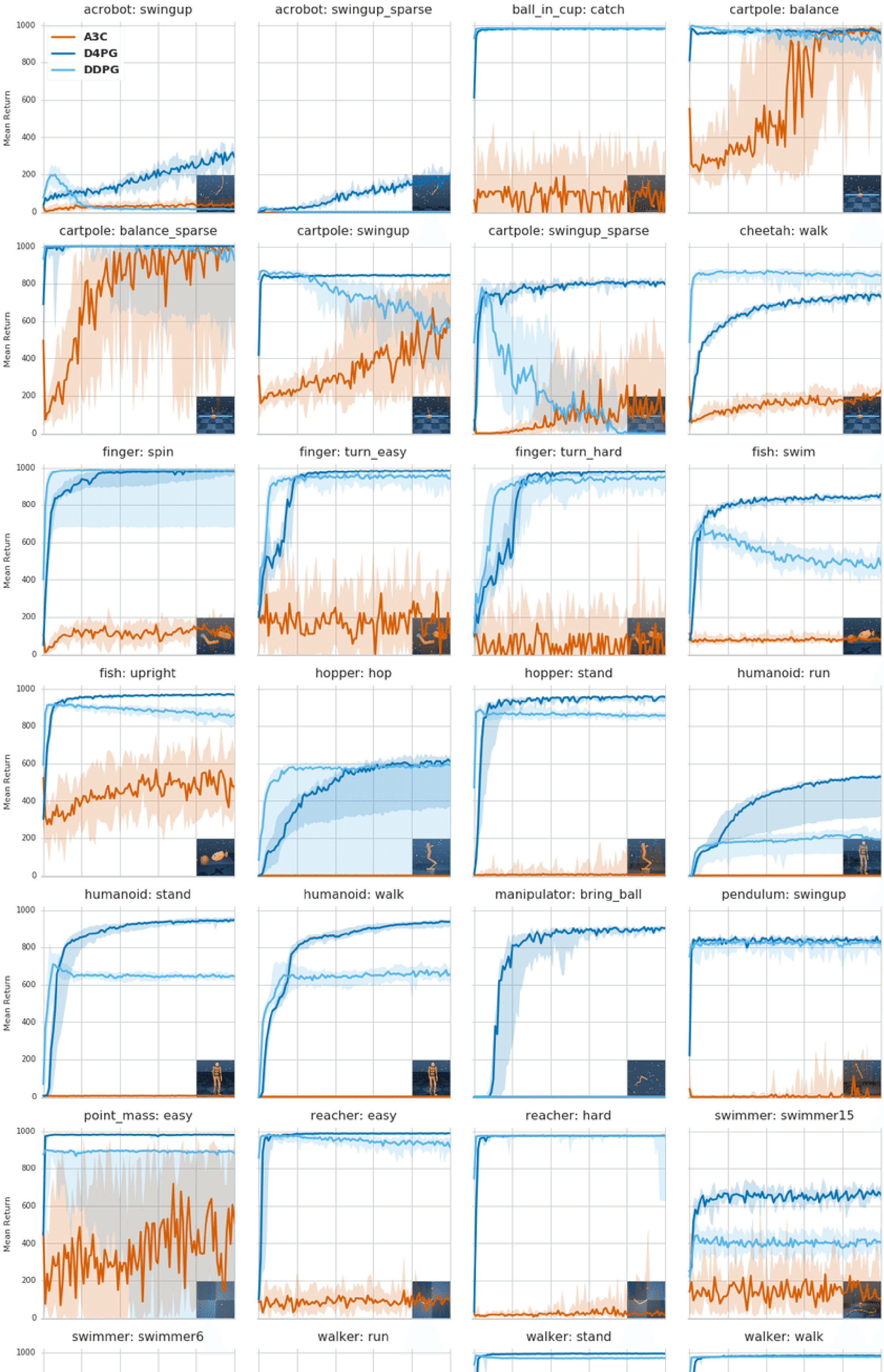
Future Improvements

Using **DDPG** with 20 agents worked well, but trying to use a Prioritized Experience Replay to learn from the experiences with high error would be a good idea, based on the “A novel DDPG method with prioritized experience replay” paper. Experimental results show to reduce training time, make training more stable and less sensitive to change in some hyperparameters

(i.e. size of replay buffer, minibatch and the updating rate of the target network)

Also, based on DeepMind Control Suite's performance benchmarks on MuJoCo physics engine, **D4PG** ,overall, seemed to have a better performer than **DDPG** and **A3C**.




acrobot: swingup
acrobot: swingup_sparse
ball_in_cup: catch
cartpole: balance
cartpole: balance_sparse
cartpole: swingup
cartpole: swingup_sparse
cheetah: walk
finger: spin
finger: turn_easy
finger: turn_hard
fish: swim
fish: upright
hopper: hop
hopper: stand
humanoid: run
humanoid: stand
humanoid: walk
manipulator: bring_ball
pendulum: swingup
point_mass: easy
reacher: easy
reacher: hard
swimmer: swimmer15
swimmer: swimmer6
walker: run
walker: stand
walker: walk
A3C
D4PG
DDPG
Mean Return
0
200
400
600
800
1000