Weight Sharing in Deep Reinforcement Learning for Continuous Control

G109 | s1784227, s1786813, s1792338

Abstract

Robot locomotion is a complex task because of non-linear dynamics and uncertainty in sensory information. Deep reinforcement learning is a good match for this problem, as neural networks have the ability to model complex non-linear functions, and reinforcement learning is designed to deal with uncertainty. Recently proposed algorithms for continuous control using deep reinforcement learning utilize fully connected networks that do not take advantage of the structurally equivalent components in robots, such as arms or legs. This work proposes a neural network architecture that incorporates locally connected sub-networks for structural components and shared weights for equal parts. Utilizing the Mujoco robot simulation environments, the two more complex robots out of the 4 investigated showed noticeable improvements in maximum attainable reward when utilizing our proposed architecture over the baseline.

1. Introduction

There is a strong desire to create autonomous robotic systems that can make their own decisions without a dependence on pre-engineered features or having human oversight. However, this is incredibly hard to achieve as most robot systems have non-linear dynamics that are difficult to model explicitly, in addition to having uncertainty in sensory information. Thus, there is a need for algorithms that can work without exact models of the system, in addition to being robust to uncertainty.

Deep reinforcement learning offers a potential solution to this problem, with promising performance in difficult tasks such as playing Atari Video games successfully (LeCun et al., 2015; Mnih et al., 2015). This ability to learn to solve complex tasks given only a reward signal and information about the environment can be applied to robotic locomotion via the use of robotic simulation. Recent algorithms for continuous control (for applications such as controlling robot joint torques) are based upon the policy gradient method (Lillicrap et al., 2015), utilizing deep neural networks as function approximators. Whilst they can be successful, they are not particularly sample efficient, meaning they require many trials to learn desirable behaviour. This would not be feasible for an expensive, real world robot, so simulated environments are invaluable.

This report presents an approach to designing the architectures of policy networks to take advantage of robots having structurally equivalent parts. Recent approaches learn different weights for all actions that control the joints. Conversely, our proposed design utilizes locally connected sub-networks with shared weights for low-level control of structurally equivalent parts. The simulated humanoid robot for example, seen in Figure 1d, in our shared-weight architecture learns one sub-network and another one for the arms. For execution, the architecture uses two copies of these networks for both legs and arms respectively, with the higher layers sending the sub-networks different inputs. This proposed architecture is investigated with the aim of identifying whether it can speed up training and/or improve the overall performance.

In the next section, we introduce the used reinforcement learning setting that we employed to test our neural network architectures and the deep deterministic policy gradient (DDPG) method that was used to train the networks. Furthermore, we describe different additions that are necessary for a successful training of the networks. In Section 3 we describe the reinforcement learning environment which is based on the Mujoco (Todorov et al., 2012) physics engine and introduce the robot-like agents that we used in our experiments. After that we propose the weight sharing policy network architecture that makes use of structurally equivalent parts. Our experiments are presented in Section 5. We show that our approach outperforms the baseline for environments with a lot of equivalent parts. Then, we compare our approach to related work and finally conclude our work in the last section and give directions for future work.

2. Methodology

The reinforcement learning setup that is used in this work is defined as follows: an agent interacts with an environment E in discrete timesteps. For each timestep the agent receives an observation x_t, takes an action a_t and receives a scalar reward r_t. The observations are sensory measurements of the robot's state such as joint angles and positions. The environment is modeled as a Markov decision process with state space $\mathcal{S}$, real valued action space $\mathcal{A} = \mathbb{R}^N$ and an initial state distribution $p(s_1)$. The transition probabilities are given by $p(s_{t+1}|s_t, a_t)$, which defines the probability of entering state s_{t+1} after beeing in state s_t and taking an action a_t. The reward is also a function of the chosen action and current state $r(s_t, a_t)$.

The agents that control the robots selects its action with a deterministic policy that maps states to actions $\pi : \mathcal{S} \rightarrow$

$\mathcal{A}$, meaning that a policy π selects an action to take upon observation of a state s_t. The expected return from a state is given by $R_t = \sum_{i=t}^{T} \gamma^{i-t} r(s_i, a_i)$, which is a sum of future rewards discounted by a discounting factor $\gamma \in [0, 1]$. The discount factor determines the importance of future reward over longer term ones, i.e. as closer it is to 1 the longer is the time horizon. The action-value function describes the expected return after taking an action a_t in state s_t after which the policy π is followed:

$$Q^{\pi}(s_t, a_t) = \mathbb{E}_{r_{i\geq t}, s_{i\geq t}\sim E, a_{i>t}\sim\pi}[R_t | s_t, a_t] \tag{1}$$

The action-value function can be recursively computed using the Bellman-Equation (Sutton & Barto, 1998), which is defined as follows if the policy is deterministic:

$$Q^{\pi}(s_t, a_t) = \mathbb{E}_{r_t, s_{t+1}}[r(s_t, a_t) + \gamma Q^{\pi}(s_{t+1}, \pi(s_{t+1}))] \tag{2}$$

It is possible to use function approximators to learn the action-value function. A function approximator parameterized by θ^Q can be optimized by minimizing the following loss:

$$L(\theta^Q) = \mathbb{E}_{s_t\sim\rho^\beta, a_t\sim\beta, r_t\sim E}\left[(Q(s_t, a_t|\theta^Q) - y_t)^2\right] \tag{3}$$

with ρ^π being the state visitation distribution for a policy π, and y_t given by

$$y_t = r(s_t, a_t) + \gamma Q(s_{t+1}, \mu(s_{t+1})|\theta^Q) \tag{4}$$

In general, this is referred to as Q-learning (Watkins & Dayan, 1992) when the greedy policy is used.

2.1. Deep Deterministic Policy Gradient

The overall goal in reinforcement learning is to find a policy in which the expected return is maximized, starting in a state drawn from the starting distribution. The idea of policy gradient methods is to optimize a parameterized policy $\pi(a|\theta)$ directly by maximizing the expected return from the start distribution $J = \mathbb{E}_{r_i, s_i\sim E, a_i\sim\pi}[R_1]$. The advantage of learning the policy directly is that this enables infinite observation and action spaces which is intractable when using a greedy policy $\mu(s) = \arg\max_a Q(s, a)$ based on a learned action-value function.

To obtain the policy updates we get the gradient of the expected return with respect to the parameters of the policy by applying the chain rule as follows:

$$\begin{aligned}\nabla_{\theta^\pi} J &\approx \mathbb{E}_{s_t\sim\rho^\beta}[\nabla_{\theta^\pi} Q(s, a|\theta^Q)|_{s=s_t, a=\pi(s_t|\theta^\pi)}] \\ &= \mathbb{E}_{s_t\sim\rho^\beta}[\nabla_a Q(s, a|\theta^Q)|_{s=s_t, a=\pi(s_t)} \nabla_{\theta^\pi}\pi(s|\theta^\pi)|_{s=s_t}]\end{aligned} \tag{5}$$

where $Q(s, a|\theta^Q)$ is a parameterized action-value function and ρ^β is the state visitation distribution of another policy β, which means that it is possible to learn θ^π off-policy using β. It has been proven by Silver et al. (2014) that this gradient is the *policy gradient*, which is the gradient of the policy's performance.

The method of optimizing a policy based on the gradient update shown above is called *deterministic policy gradient* (Silver et al., 2014). This method was extended by Lillicrap et al. (2015) to use deep neural networks for the policy and the action-value function, which they called *deep deterministic policy gradient* (DDPG). The actor network which resembles the policy is learned with stochastic gradient ascent on the gradient defined in equation 5. The critic network which approximates the action-value function $Q(s, a|\theta^Q)$ is trained by minimizing the loss given in equation 3.

One requirement to train neural networks successfully is that the training examples are are independently and identically distributed, which is not the case when the environment is explored sequentially. This is tackled by the use of a replay buffer, as introduced by Mnih et al. (2015). The replay buffer is a finite sized cache $\mathcal{R}$ which stores tuples of previously explored transitions (s_t, a_t, r_t, s_{t+1}). When there are enough transitions in the replay buffer, makes is very likely that sampled transitions are uncorrelated.

Implementing Q-learning with neural networks based on equation 3 can be particularly unstable as the network $Q(s, a|\theta^Q)$ that is being updated is also used in calculating the target value y_t given in equation 4. The solution proposed by Lillicrap et al. (2015) is to use a separate target network and update its weights gradually. This is done by making a copy of both the actor and critic networks, $\mu'(s|\theta^{\mu'})$ and $Q'(s, a|\theta^{Q'})$ respectively. These are gradually updated with $\theta' \leftarrow \tau\theta + (1 - \tau)\theta'$, where $\tau \ll 1$. This constrains the target networks to change slowly, inducing greater stability of learning.

2.2. Parameter Noise

As mentioned in coursework 3, the reinforcement learning algorithm requires some exploratory behaviour in order to learn more about its environment. The method that was proposed in CW3 was through a noise injection $\mathcal{N}$ into the action space of the actor policy μ, like so:

$$\mu'(s_t) = \mu(s_t|\theta_t^\mu) + \mathcal{N} \tag{6}$$

The noise is temporally correlated for an efficient exploration in systems with inertia, which corresponds to using an Ornstein-Uhlenbeck process to generate $\mathcal{N}$. However, it is proposed by Plappert et al. (2017) that adding noise directly to the agent's parameters can lead to "more consistent exploration and a richer set of behaviours".

This structured exploration through parameter noise is achieved by applying additive Gaussian noise to the parameter vector of the current actor policy:

$$\tilde{\theta^\mu} = \theta^\mu + \mathcal{N}(0, \sigma^2 I) \tag{7}$$

The resulting perturbed policy is referred to as $\tilde{\pi}$, and this policy is kept for a whole rollout (simulation run that is ended by time-limit or failure condition, also referred to as an episode). However, as mentioned by Plappert et al. (2017), it is not obvious that simply adding spherical Gaussian noise to the parameters of a deep neural network with potentially millions of parameters and complicated nonlinear interactions will result in meaningful perturbations in the outputs. However, it was shown by Salimans et al. (2017) that one can reparameterize a network to achieve this very thing. This was implemented by utilizing layer normalization, introduced by Ba et al. (2016). Similar to batch normalization, it normalizes across activations within a layer, meaning the same perturbation scale σ^2 can be used across all layers and parameters.

The layer normalization statistics over H hidden units in the same layer l are calculated as follows, with μ^l the layer mean and σ^l the layer variance

$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l \qquad \sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}(a_i^l - \mu^l)^2)} \tag{8}$$

These statistics are used to fix the mean and variance of the summed inputs within each layer.

However, parameter noise introduces another issue as to the selection of the noise scale σ. As the noise is being added directly to the parameters of the neural network, this means the perturbation of the output, even with the same σ, will be massively dependent on the specific network architecture. In addition, this scale will likely change over time as learning progresses and parameters become more sensitive to noise. This issue is solved by adapting the scale of the parameter noise over time by comparing it to the variance in the action space that it induces. For every K timesteps, σ_k is updated like so:

$$\sigma_{k+1} = \begin{cases} \alpha\sigma_k & \text{if } d(\pi, \tilde{\pi}) < \delta \\ \frac{1}{\alpha}\sigma_k & \text{otherwise} \end{cases} \tag{9}$$

In this equation, a distance measure $d(\pi, \tilde{\pi})$ is utilized to determine the difference (in action space) between the perturbed and unperturbed policies. With a typical value of $\alpha = 1.01$, if this distance is too large, then the noise scale σ_{k+1} is rescaled to be smaller, and increased if the opposite is true. This method means that the magnitude of the exploration in action space is never too large or too small.

For DDPG, the distance measure $d(\pi, \tilde{\pi})$ is given by the following equation:

$$d(\pi, \tilde{\pi}) = \sqrt{\frac{1}{N}\sum_{i=1}^{N} \mathbb{E}_s\left[(\pi(s)_i - \tilde{\pi}(s)_i)^2\right]} \tag{10}$$

where $\mathbb{E}_s[\cdot]$ is estimated from a batch of states in the replay buffer (batch size 32). N is the dimension of the action space (the number of outputs of the actor network). In equation 9, $\delta = \sigma = 0.1$. From this, it is clear to see how the adaptive parameter noise keeps the effective action space noise equivalent (or at least near) to some action noise with the same standard deviation $\delta = \sigma$.

3. Mujoco Environment

The motivation for our proposal of weight sharing in deep reinforcement learning for continuous control is to make the training in robotic environments more efficient by utilizing structurally equivalent parts. Since it is not feasible for us to make experiments with real world robots, we rely on physical simulations of robot-like agents. These allow us evaluate and compare different approaches, while being an approximation of the real task.

In coursework 3, we used the Roboschool environment, which contains various simulated robots such as a two dimensional walker, cheetah and a humanoid. But we found that due to the reward structure in Roboschool it seems to be very hard to train. After trying many different network architectures and hyperparameters for DDPG, we did not manage to train any of the agents to move forward at all. Roboschool is a replacement for the Mujoco physics engine (Todorov et al., 2012) (the motivation for the replacement is that Mujoco requires a licence to be used). However, most recent publication still rely on Mujoco instead of Roboschool (Tassa et al., 2018; Song & Wu, 2018), hence we decided to switch to Mujoco. Mujoco contains the same agents, but the physics simulation is different and the reward functions are not the same. With Mujoco we were able to train agents that move and eventually agents that achieve a very good performance.

Mujoco is a physics engine which is specialized on model-based control. It is able to efficiently simulate robot like agents with multi-joint dynamics and contact responses (Todorov et al., 2012). For our experiments we use four environments of Open AI Gym (Brockman et al., 2016) that are powered by Mujoco and have been introduced as reinforcement learning benchmark tasks by Duan et al. (2016): Walker2d, HalfCheetah, Ant and Humanoid (cf. Figure 1 respectively). In the case of Mujoco environments, actions are the torques that are applied to the joints of the agents and observations that consist of: the robot's position and velocity as well as its joint positions and joint angle velocities. The ant and humanoid environment include contact information of body parts with the floor.

The task in each of these environments is to learn a policy that makes the agent move forward as fast as possible while applying torques that are as small as possible. The simulation is episodic and is run for a maximum of 5000 steps. All but the cheetah environment have a stop condition if the upper part of the agent comes to close to the floor. While the reward functions of the environments differ slightly in the used penalties, all of them use the the speed of the agent in x-direction as a reward:

$$v_t = \frac{pos_{\text{after}} - pos_{\text{before}}}{dt}, \tag{11}$$

where dt represents of how long the simulation has been run since the last step and pos_{after} is the x-position after applying the actions and pos_{before} before applying them. In the following we refer to this term with v_t. In addition, all reward functions contain a penalty term for the applied actions a, which incentivizes the agent to not use too large torques. On the one hand, this is desirable for real robots because smaller torques lead to a smaller power consumption and the penalty also avoids unnecessary movements of the robot. On the other hand, we want to avoid high

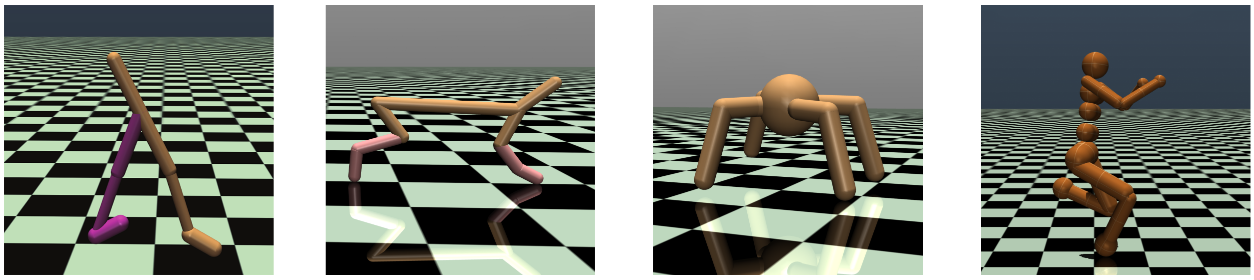
(a) Walker2d-v2 (b) HalfCheetah-v2 (c) Ant-v2 (d) Humanoid-v2

Figure 1. OpenAi Gym environmemts based on the Mujoco physics engine that where used in our experiments.

torques because they may lead to extreme movements that are not possible in reality or could damage the robot or the environment.

In the following, we discuss the structure and reward function of each of the environments that we use in more detail. For all environments we use the the version v2 in Open AI Gym.

3.1. Walker2d

The walker environment is a fairly simple agent. It consists of two controllable legs and a torso stump (cf. Figure 1a) and can only move along the x-axis of the physical environment. The agent has 6 controllable joints, three in each leg, which can be moved around the y-axis. The difficulty in this environment is that the agent must not tip over and does not has an upper torso to balance itself. The reward function of the walker is defined as follows

$$r_t = v_t - 0.001 \cdot ||a||^2 + 1, \tag{12}$$

the last constant term is a survival bonus.

3.2. HalfCheetah

The half cheetah is also only able to move along the x-axis and has also 6 controllable joints. In contrast, to the walker the cheetah is not prone to fall over that easily, as it is naturally in a stable state without changing its position. The reward function of the cheetah is:

$$r_t = v_t - 0.1 \cdot ||a||^2. \tag{13}$$

There is no survival bonus in this environment. We believe the reason is that the simulation does not end when the cheetah falls over, so if there would be a survival bonus it could set $a = \mathbf{0}$ and would constantly receive a positive reward, which would lead to unintended behavior.

3.3. Ant

The ant is a 3d robot that can move in any direction and has 8 controllable joints. Each leg has two controllable joints where one axis can be controlled: the hip joint allows the leg to tilt around the z-axis and the knee moves the leg up and down. One difficulty with the ant, compared to walker and cheetah, is that the agent must not only learn to how to run but also to run in the right direction. The reward function is given by:

$$\begin{aligned} r_t &= v_t - 0.5 \cdot ||a||^2 + \text{cost}_{contact} + 1, \\ \text{with cost}_{contact} &= 0.5 \times 10^{-3} \cdot ||\text{clip}(c, -1, 1)||^2, \end{aligned} \tag{14}$$

the last constant term is again a survival bonus and the contact cost term penalizes hard impacts on the ground.

3.4. Humanoid

The humanoid is by far the most complex agent. It has 9 controllable joints of which some have three degree of freedom (DoF), thus allowing the humanoid more complex movements. There are joints with 1 DoF in each knee and elbow, the hips are universal joints with 3 DoF, the abdomen is modeled as one joint with 3 DoF and the both shoulders have 2 DoF. This makes the control of the humanoid much more complex compared to the other robots, as it is harder to coordinate the torques for the different axes of the joints. The reward function of the humanoid is defined as follows:

$$\begin{aligned} r_t &= 0.25 \cdot v_t - 0.1 \cdot ||a||^2 + \text{cost}_{contact} + 5, \\ \text{with cost}_{contact} &= \min\{0.5 \times 10^{-6} \cdot ||c||^2, 10\}, \end{aligned} \tag{15}$$

the last term is again a survival bonus and is higher compared to walker and ant. Similarly to the ant, there is a impact penalty for contacts with the ground.

4. Proposed Network Architectures

In most of the recent work for continuous control with deep reinforcement learning, the policy networks are fully connected. That means all outputs depend on the same hidden representations and every output unit has its own set of weights. We propose a novel architecture for robot control that utilizes local connectivity and weight sharing. Our network architecture is split into two parts: the *commander* maps the observations into a shared non-linear representation and produces different inputs for the following *controllers*, which are locally connected subnets that produce the outputs for a subset of all actions given the hidden representation produced by the commander. The weights of the controllers can be shared among different action subsets.

Figure 2 shows an example architecture for a humanoid robot. The commander takes the observations as inputs

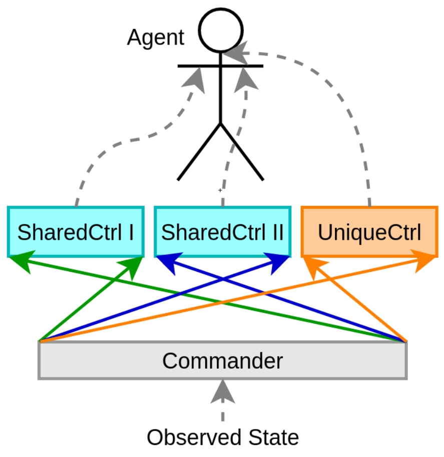

Figure 2. Two controllers with shared weights control the left and right arm of a humanoid robot. An independent third controller is in charge of its neck.

and can exist of an arbitrary number of fully connected hidden layers. After these layers, the commands for the three controllers are produced. While these commands are different for all controllers, the first two controllers share the weights of their hidden layers, because the arms of the humanoid are structurally equivalent. The controllers then output the actions for the arms and the neck, respectively.

5. Experiments and Results

With respect to obtaining our overall objective of evaluating whether or not our proposed commander-controller architecture improves learning, we compared the performance of different policy network architectures in the four Mujoco based OpenAI Gym environments that we described in Section 3. In the following, we describe the used hyperparameter settings and specifics of our implementation. Then, we present and discuss our results.

Experiment Setup While experimenting with Mujoco and in the previous work of coursework 3 with Roboschool we found that DDPG is very sensitive to different hyperparameter settings. We furthermore observed that baseline implementations from other papers, e.g. rllab (Duan et al., 2016), rllabplusplus (Gu et al., 2016) and OpenAI baselines (Plappert et al., 2017), do not exactly follow the original DDPG algorithm of Lillicrap et al. (2015). These issues have been thoroughly investigated by Henderson et al. (2017), with their findings matching what we have observed. For example, that "implementation differences which are often not reflected in publications can have dramatic impacts on performance." (Henderson et al., 2017).

While we have our own fully functional implementation of DDPG in PyTorch[1], we therefore decided to use the DDPG implementation of OpenAI baselines (Dhariwal et al., 2017)[2], to obtain results that are better comparable with results presented in other work. Furthermore, we use the following hyperparameter choices that are partly based on the suggestions of Henderson et al. (2017) for running tasks in OpenAI's Mujoco environments:

- discount factor: $\gamma = 0.995$
- target update factor: $\tau = 0.01$
- batch size: $m = 128$
- use reward and observation normalization
- 25 mini-batch Adam (Kingma & Ba, 2014) steps every 50 simulation steps
- relu for hidden layers and tanh for action outputs
- actor learning rate: 1×10^{-4} and critic learning rate 1×10^{-3}
- critic L2 regularization 1×10^{-2}

Evaluation Method We evaluate the performance of the different architectures every 1000 simulation steps, by running the policy without added noise, such that there is no exploration. The sum of all rewards is then reported as a evaluation metric. However, the results vary a lot between runs with different seeds, because the random exploration has a huge impact on the performance of the robot. In their experiments Henderson et al. (2017) have shown that using $N < 5$ seeds is not sufficient to make statistically significant conclusions. Furthermore, they state that it can be misleading if you only report the top-N trials. Therefore, we decided to run 10 random seeds for each experiment and average the results over them. In addition, we run each evaluation 5 times and average their results to get a better estimate of the performance of the starting distribution.

Network Architectures The baseline policy network for all environments is a fully connected network with two hidden layers and 128 hidden units each. After each hidden layer follows a Layer Normalization layer and a ReLU nonlinearity. Lastly, the output units are restricted to the action space of $(-1, 1)$ with tanh.

Our naming convention indicates the numbers of layers in the commander and the controllers respectively. We count the branching layer that branches out to the controllers as a part of the commander. This means for example, that the network architecture *Commander-2-Controller-1* has three layers: one commander hidden layer, one commander hidden branching layer, no controller hidden layer and one controller output layer.

For our shared-controller experiments, unless otherwise stated, we design the numbers, purposes and dimensionalities of the controllers for every environment as follows. The walker's policy networks have a single controller with 64 hidden units, shared among both of the walker's legs. Likewise the half cheetah's policy networks consist of a

[1]https://github.com/nicoring/RoboRL

[2]An implementation of our architectures using baselines can be found here: https://github.com/jejay/baselines

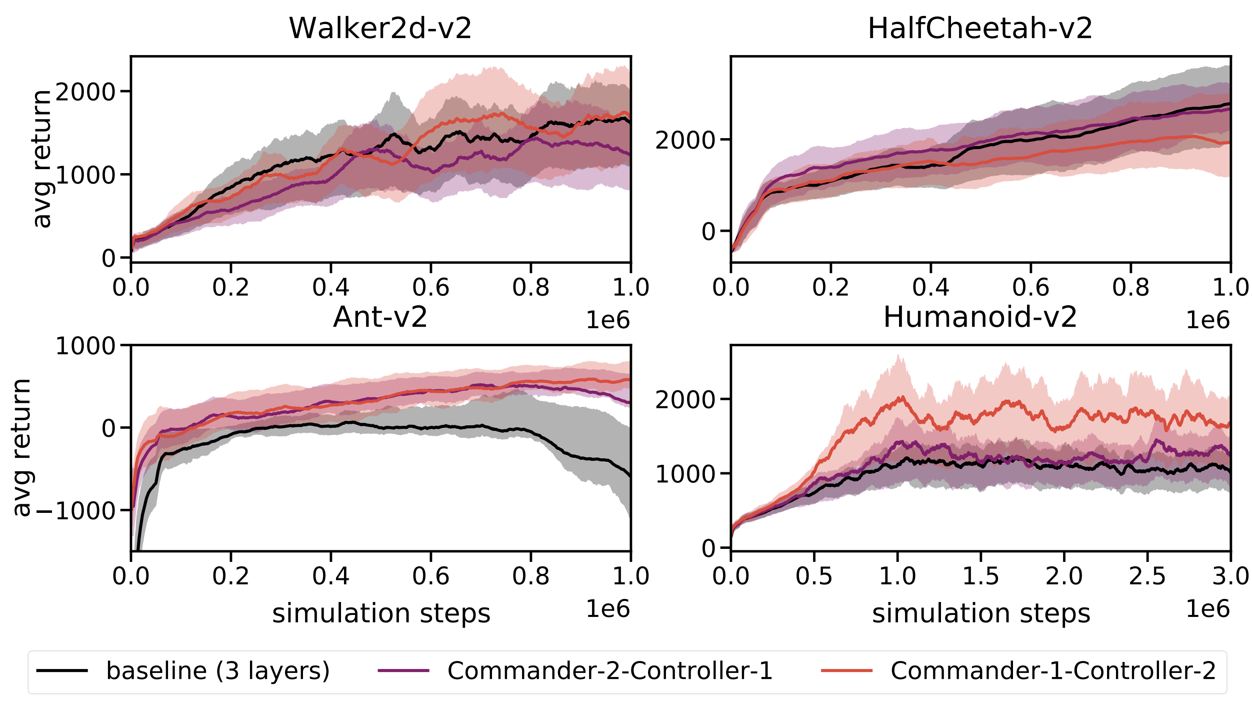

Figure 3. Average returns over 10 seeds in four environments for three network architectures. For better presentation reported values are smoothed with a rolling mean with a window of 50. The lines show the mean performance and the colored areas show the regions between the 25th and 75th percentile.

single controller with 64 hidden units, shared among both of the half cheetah's legs. The ant's shared policy networks have a single controller with 32 hidden units and which is shared among all of its four legs. The humanoid's shared policy networks have three distinct controllers. One with 32 hidden units, shared among both knees including the part of the hip attached to each knee; another controller with 24 hidden units shared among both arms with attached shoulders and a third non-shared controller for the abdomen with 16 hidden units. For all environment's shared network architectures the sum of all controller's hidden layers multiplied by the number each controller is shared equals 128, the hidden dimensionality of the fully connected baseline. This sum can be pictured as the width among all controllers when they are laid out as pictured in figure 2. For example in the case of the humanoid this is $32 \cdot 2 + 24 \cdot 2 + 16 = 128$.

5.1. Commander-Controller Networks

We evaluate commander-controller networks that have fewer free parameters than the fully connected baseline. The performance of the vanilla baseline, Commander-2-Controller-1 and Commander-1-Controller-2 architectures can be seen in Figure 3, evaluated on the four OpenAI gym Mujoco environments shown in section 3. As indicated above these results are the average over 10 random seeds to get an better estimation of the performance. The values we present here are additionally smoothed with a running average with a window size of 50, we did this to make it easier to compare the results, as the performance on all environments has a very high variance (especially for Walker2d and Humanoid). An example of not smoothed results is shown in Figure 4 for the Walker2d environment.

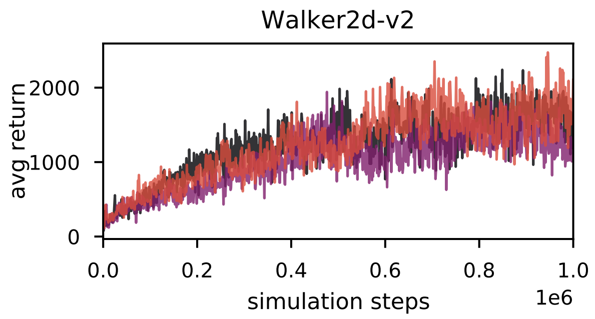

Figure 4. Not smoothed results for the Walker2d environment as a comparison to the smoothed results. The line colors are the same as in Figure 3.

Walker2d This environment appears to show no discernible differences in either the overall performance or the speed of training from the weight-sharing architectures or the baselines. We suspect this lack of improvement is due to the simplicity of the environment.

HalfCheetah This appears to show the Commander-1-Controller-2 network is outperformed by the Commander-2-Controller-1 and baseline networks in overall performance. There is however little to distinguish between the Commander-2-Controller-1 and baseline architectures, with the Commander-2-Controller-1 perhaps doing better initially, and the baseline showing signs of improving past the Commander-2-Controller-1 after the 1 million simulation steps shown. Despite this, the overlapping variances of all three architectures make it hard to claim any statistically significant superiority of one over the rest, in speed or performance.

The lack of improvement in the HalfCheetah is perhaps not surprising if indeed the weight-sharing architecture enables better learning of structurally equivalent parts. This is due to the observation that the motions of the front and back legs of the HalfCheetah are in fact extremely dissimilar (one can inspect Figure 1b to infer this). In a close to optimal performance, the forward leg makes huge leaps, while the rear leg only pushes forward.

Humanoid The humanoid appears to show some improved performance from implementing weight-sharing, with the baseline consistently performing worse than the Commander-2-Controller-1 architecture, and significantly worse than the Commander-1-Controller-2. For neither weight-sharing model however, does the speed of training appear to be improved.

We suspect that the reason for the Commander-1-Controller-2 network outperforming the Commander-2-Controller-1 network for the humanoid is that the humanoid requires more complex low-level control for its limbs and thus benefits from the additional layer in the controllers that the Commander-1-Controller-2 network provides.

Ant The Ant environment appears to show noticeable improvements from implementing our proposed architectures in terms of its overall performance. The performance benefits significantly from implementing either weight-sharing architectures over the baseline, achieving positive rewards that steadily increase over the 1 million simulation steps. The baseline architecture in contrast appears to reach its maximum potential at around 400,000 steps and never achieves an average return above 0. Again however, there is no speed benefit from introducing the weight-sharing.

It is particularly interesting that the addition of shared weight controllers improves the performance on the Ant environment the most. It could be argued it was the most theoretically promising candidate for the architecture, as it is a symmetric robot with 4 identical legs that must operate in very similar fashions in order to form a coherent walking motion. The improvement on the Humanoid environment in addition to the Ant, even evident across 10 seeds, demonstrate that our proposed architecture is a promising concept for robotic locomotion.

5.2. Restricting Command Bandwidth

In coursework 3 we argued that our architecture can be interpreted as a higher level commander calculating and sending high level commands lower level controllers which then output the low level actions. Until now the dimensionality of those commands is equal to the dimensionality of each controllers hidden dimensionality. For example, the ant's command bandwidth has a dimensionality of 32. Since high level commands should per definition not be too detailed our following experiments introduce a bottleneck between the commander and the controllers.

Before experimenting with the bottleneck, we extend our ensemble of network architectures to Commander-2-Controller-2. This is a combination of both previously proposed architectures as there is now one hidden layer in the commander and in the controllers, in addition to the branching layer where we enforce the bottleneck. As a result the network now has four layers in total. This increases the overall parameter count drastically, which in turn gets decreased by introducing a bottleneck. In our experiments the performance of the four-layer architecture without introducing a bottleneck did not differ from the three-layer architectures mentioned above.

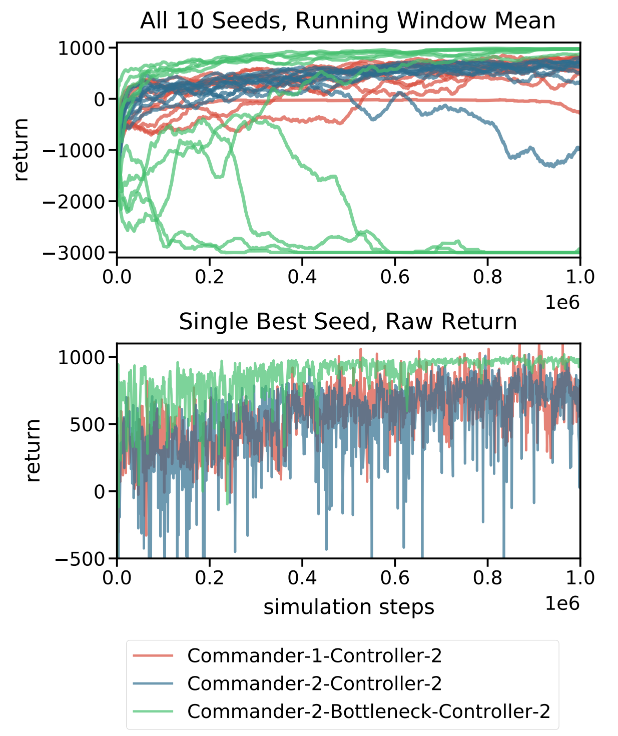

Figure 5. Runs with and without bottleneck in the Ant environment. At the top each run individually has been smoothed with the same rolling window mean as in Figure 3.

Based on the four layer network, we introduce three different bottleneck sizes by reducing the hidden units of the branching layer to decrease the dimensionality of the command bandwidth from 32 to eight, four and two. On ten runs, significant differences in performance are not visible for the eight and four dimensional bottlenecks. The two dimensional bottleneck, when evaluated with the same metrics as used in figure 3, shows an average performance worse by a large magnitude and its 25-75-percentile area spans nearly the whole reward space (plot not included). We investigate this by displaying the individual seeded runs in figure 5 (top). It shows that the bottleneck runs can be categorized into failing and succeeding runs. The failing runs keep deteriorating over time until they land at a reward of -3000, which is two or three times worse than the worst baseline result. In contrast, nearly every single succeeding run shows an improved performance over the Commander-1-Controller-2 and Commander-2-Controller-2 runs. This means that the two dimensional bottleneck Commander-2-Bottleneck-Controller-2 is the best performing network on the ant, if the evaluation metric is a best case average. In figure 5 (bottom) the raw return of the single best runs of each network confirms this impression. Even while both non-bottleneck networks' runs peak slightly higher, the bottleneck network is the only one able to visibly reduce its evaluation variance within a single run. For all other of the best six bottleneck runs, the plot looks similar.

This shows that commander and controller can communicate with low dimensional high level commands; and if a communication can be learned, the overall performance is even better than without the imposed bottleneck. We suspect that by enforcing a useful regime, we reduce the space of suitable policies, effectively reducing the chance of finding any sufficient policy. But if a useful policy can be found, the regime seems have a lock-in effect and therefore reduces the variance within that run.

6. Related Work

Sharma & Kitani (2018) observed that robot locomotion is often of cyclic nature. Thus, they introduced phase parametric policy and value networks in order to explicitly enforce a cyclic structure within the policy and value spaces available to an agent, thus encouraging learning to conform to this successful cyclic regime. They found their implementation of phase-DDPG improved upon the evaluated reward of their baseline by up to ~50% on the Mujoco Walker2d task. A natural walking motion stems mostly from the phased co-ordination of similar motions in each leg, which our shared network structure may adopt even without the enforced cyclic behaviour in Sharma & Kitani (2018). This would occur by the controllers learning low-level control with the fully connected commander governing the cyclic co-ordination. Combining these approaches in some way would certainly be an interesting endeavour for future work, for example by enforcing the phase parameterization on the controller alone.

Heess et al. (2016) introduce a hierarchical network structure with high level and low level controllers embedded in a recurrent neural network such that the communication is constrained by time slots. In contrast to our work, the low level controllers are not designed to fit specific structural parts of a robot. Instead they focus on transfer learning between tasks for the same robot. They assume that the learned low level controllers are invariant to the high level task like walking, searching or escaping a maze. They show that by transferring low level controllers to a new task for the same robot , the performance can be accelerated and improved on that task. The performance gap is higher the more complex the new task is; some even can not be learned without transfer learning. Similarly, our approach, especially with the bottleneck, can be interpreted as a high-level-low-level communication with instantaneous transfer learning between shared low level controllers that are responsible for equivalent structures in a robot: what is true for one leg, should also be learned for all legs. Also, we come to the same conclusion, namely that especially complex tasks profit from this transfer learning.

Our proposal of the commander-controller network architecture was inspired by the use of controllers such as PID or MPC in control theory. While we learn the controllers as part of out network it is also possible to combine deep reinforcement learning with these controllers. Wang et al. (2007) proposed an adaptive PID controller design, where the parameters of the PID controller are tuned on-line by an actor-critic reinforcement learning system. In contrast to our approach, they only tune the controller's parameters and the reference trajectory is given. An interesting approach could be to also generate the trajectory with a RL system. We could for example, replace our low-level controllers with PID controllers, which then receive the reference joint angles and output torques based on the current joint angles.

7. Conclusion and Future Work

In this report, we proposed a novel policy network architecture based on local connectivity and weight sharing for robot locomotion tasks. Furthermore, we introduced the the reinforcement learning setting in which we evaluate our algorithms. Since we cannot use real robots we use OpenAI gyms that use the Mujoco physics engine to simulate robot-like agents. We gave a detailed description of the Mujoco environment and the reward function of all four used agents.

The motivation for our proposed policy architecture is to make use of structural equivalent parts to speed up training of reinforcement learning agents and to get better overall performance. Our experiments with Walker2d and HalfCheetah didn't show an improvement in training speed and overall performance. We believe that the reasons are that the control of the legs in 1d is not hard to learn and the overall coordination is more important. However, we get a better performance with networks that employ weight sharing for low-level controllers for the Ant and Humanoid environments. The reason might be, that the low-level control is more complex for these environments as the movements are not restricted to one dimension. Furthermore, we showed that introducing a bottleneck between the commander and the controllers can reduce the variability in the peformance.

Our experiments have shown that our proposed architecture is able to achieve better performances than the baseline in more complex environments. Therefore, a next step would be to evaluate it on even more complex environments such as the new robotics environments in OpenAI gym (Plappert et al., 2018), which feature, for example, a simulated robot hand. In these environments, it would be interesting to experiment with even deeper and wider networks, which is not really necessary for the Mujoco environments because they can be solved almost optimally with quite small networks as has been shown by Henderson et al. (2017). Another part of future work is to test our policy architectures with other state-of-the-art deep RL algorithms, such as PPO (Schulman et al., 2017).

References

Ba, Jimmy Lei, Kiros, Jamie Ryan, and Hinton, Geoffrey E. Layer Normalization. 2016. ISSN 1607.06450. URL https://arxiv.org/pdf/1607.06450.pdf.

Brockman, Greg, Cheung, Vicki, Pettersson, Ludwig, Schneider, Jonas, Schulman, John, Tang, Jie, and Zaremba, Wojciech. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

Dhariwal, Prafulla, Hesse, Christopher, Klimov, Oleg, Nichol, Alex, Plappert, Matthias, Radford, Alec, Schulman, John, Sidor, Szymon, and Wu, Yuhuai. Openai baselines. https://github.com/openai/baselines, 2017.

Duan, Yan, Chen, Xi, Houthooft, Rein, Schulman, John, and Abbeel, Pieter. Benchmarking deep reinforcement learning for continuous control. In *International Conference on Machine Learning*, pp. 1329–1338, 2016.

Gu, Shixiang, Lillicrap, Timothy, Ghahramani, Zoubin, Turner, Richard E, and Levine, Sergey. Q-prop: Sample-efficient policy gradient with an off-policy critic. *arXiv preprint arXiv:1611.02247*, 2016.

Heess, Nicolas, Wayne, Gregory, Tassa, Yuval, Lillicrap, Timothy P., Riedmiller, Martin A., and Silver, David. Learning and transfer of modulated locomotor controllers. *CoRR*, abs/1610.05182, 2016.

Henderson, Peter, Islam, Riashat, Bachman, Philip, Pineau, Joelle, Precup, Doina, and Meger, David. Deep reinforcement learning that matters. *arXiv preprint arXiv:1709.06560*, 2017.

Kingma, Diederik P and Ba, Jimmy. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

LeCun, Yann, Bengio, Yoshua, and Hinton, Geoffrey. Deep learning. *nature*, 521(7553):436, 2015.

Lillicrap, Timothy P., Hunt, Jonathan J., Pritzel, Alexander, Heess, Nicolas, Erez, Tom, Tassa, Yuval, Silver, David, and Wierstra, Daan. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015. URL http://arxiv.org/abs/1509.02971.

Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Rusu, Andrei A., Veness, Joel, Bellemare, Marc G., Graves, Alex, Riedmiller, Martin, Fidjeland, Andreas K., Ostrovski, Georg, Petersen, Stig, Beattie, Charles, Sadik, Amir, Antonoglou, Ioannis, King, Helen, Kumaran, Dharshan, Wierstra, Daan, Legg, Shane, and Hassabis, Demis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, feb 2015. ISSN 0028-0836. doi: 10.1038/nature14236. URL http://www.nature.com/doifinder/10.1038/nature14236.

Plappert, Matthias, Houthooft, Rein, Dhariwal, Prafulla, Sidor, Szymon, Chen, Richard Y, Chen, Xi, Asfour, Tamim, Abbeel, Pieter, and Andrychowicz, Marcin. Parameter space noise for exploration. *arXiv preprint arXiv:1706.01905*, 2017.

Plappert, Matthias, Andrychowicz, Marcin, Ray, Alex, McGrew, Bob, Baker, Bowen, Powell, Glenn, Schneider, Jonas, Tobin, Josh, Chociej, Maciek, Welinder, Peter, et al. Multi-goal reinforcement learning: Challenging robotics environments and request for research. *arXiv preprint arXiv:1802.09464*, 2018.

Salimans, Tim, Ho, Jonathan, Chen, Xi, Sidor, Szymon, and Sutskever, Ilya. Evolution Strategies as a Scalable Alternative to Reinforcement Learning. 2017. ISSN 1744-4292. doi: 10.1.1.51.6328. URL https://arxiv.org/pdf/1703.03864.pdf.

Schulman, John, Wolski, Filip, Dhariwal, Prafulla, Radford, Alec, and Klimov, Oleg. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Sharma, Arjun and Kitani, Kris M. Phase-parametric policies for reinforcement learning in cyclic environments. In *AAAI Conference on Artificial Intelligence*, February 2018.

Silver, David, Lever, Guy, Heess, Nicolas, Degris, Thomas, Wierstra, Daan, and Riedmiller, Martin. Deterministic Policy Gradient Algorithms. *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pp. 387–395, 2014. ISSN 1938-7228. URL http://proceedings.mlr.press/v32/silver14.pdf.

Song, Jiaming and Wu, Yuhuai. An Empirical Analysis of Proximal Policy Optimization with Kronecker-factored Natural Gradients. *arXiv preprint arXiv:1801.05566*, 2018. URL https://arxiv.org/pdf/1801.05566.pdf.

Sutton, Richard S and Barto, Andrew G. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

Tassa, Yuval, Doron, Yotam, Muldal, Alistair, Erez, Tom, Li, Yazhe, Casas, Diego de Las, Budden, David, Abdolmaleki, Abbas, Merel, Josh, Lefrancq, Andrew, Lillicrap, Timothy, and Riedmiller, Martin. DeepMind Control Suite. *arXiv preprint arXiv:1801.00690*, 2018. URL https://arxiv.org/pdf/1801.00690.pdf.

Todorov, Emanuel, Erez, Tom, and Tassa, Yuval. Mujoco: A physics engine for model-based control. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pp. 5026–5033. IEEE, 2012.

Wang, Xue-Song, Cheng, Yu-Hu, and Wei, Sun. A proposal of adaptive pid controller based on reinforcement learning. *Journal of China University of Mining and Technology*, 17(1):40–44, 2007.

Watkins, Christopher JCH and Dayan, Peter. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.