# Learning to Walk

Ryan Cunningham
University of Maryland, College Park
College Park, USA
rcunning@umd.edu

**Legged walking, while seemingly simple for animals to do, is a complex problem for legged robots with many joints. This paper details the learning of a walking gait for a hexapod robot using the Deep Deterministic Policy Gradient (DDPG) algorithm described in *Continuous Control with Deep Reinforcement Learning* by Lillicrap et al., 2016. The DDPG algorithm is a reinforcement learning algorithm that was developed for environments with continuous action spaces, as in the control of the hexapod. Simulation results show that, with a constrained state space, the algorithm is successful in learning a control policy for a gait that allows the robot to walk across its environment.**

## 1 Introduction

Hexapods are a type of mobile robot that uses six mechanical legs to move its body. Each leg has three joints: one attached to the robot body (the hip joint) which controls the coxa, one attached to the coxa (the knee joint) which controls the femur, and one attached to the femur (the ankle joint) which controls the tibia and the foot at the end of the tibia. Each joint is operated by a servo motor that receives a position command and maintains that position under load.

Since hexapods have such articulated legs, they are well suited for tasks in varied environments that contain irregular surfaces, steps, and other features that would be difficult for a wheeled robot to navigate. A key disadvantage for hexapods, and legged robots in general, is that walking is significantly more difficult since they are overactuated. Commonly these robots are programmed with a gait that mimics how an animal with a similar leg configuration walks, which requires knowledge of the physical and mechanical properties of the robot. One way to develop walking gaits for hexapods without this knowledge is to have the robot learn how the movement of its legs moves its body.

Reinforcement learning is a type of machine learning that allows an agent, or robot, to learn by trial and error using feedback from its environment. Positive feedback rewards the robot's behavior while negative rewards punish it. Over time, the robot learns to maximize the reward and thus learn to choose the best action in each state. Q-learning is a popular reinforcement learning algorithm that learns a policy of behavior by considering the action at each state that provides the most future reward. The algorithm observes the agent's experience at each time step, where the experience is composed of the following features:

- $s$: the current state of the agent
- $a$: the action take from state $s$
- $r$: the reward received from taking action $a$ from state $s$
- $s'$: the next agent state that results from taking action $a$ from state $s$

Each experience is used to update the expected future reward $Q(s, a)$ for taking action $a$ from state $s$. The future reward is updated during training using the Q-function:

$$Q(s, a) = \alpha * (r + (\gamma * \max_a' Q(s', a') - Q(s, a))$$

Where $\alpha$ is the learning rate that determines how much the Q value changes, and $\gamma$ is the discount factor that controls how much of the future reward is taken into account. The learned Q values are used as a policy to select the action from any state that is expected to provide the most reward.

The goal of this paper is to use the Q-learning based Deep Deterministic Policy Gradient algorithm to have a hexapod robot walk across an environment.

## 2 Related Work

Deep Deterministic Policy Gradient (DDPG) is a recent algorithm based on Q-learning that combines features of the Deterministic Policy Gradient (DPG) and Deep Q Network (DQN) algorithms. The algorithm is a model-free actor critic algorithm, where the actor selects an action according to a policy and the critic judges the effectiveness of the action taken given the state. DDPG was developed specifically to enable learning in environments with continuous action spaces, such as those in physical control problems. The original paper showed that DDPG could successfully learn models for a variety of control tasks with the same network structure and hyperparameter selection [1].

Another paper built upon the DDPG algorithm by introducing demonstrated experiences into the learning process [2]. During the training, experiences are sampled from main and demonstration replay buffers according to the priority based on temporal difference error. The paper details success with multiple tasks using prioritized demonstration replay. While more useful for tasks with sparse rewards, using demonstrations with DDPG can be used in place of complex reward shaping that may negatively influence a learned model.

## 3 Approach

To prove a hexapod robot can learn to walk without any knowledge of its kinematics, the DDPG algorithm was used as a controller for the robot. During training, the robot collected experiences and trained the neural networks to learn a policy that would select actions that resulted in movement of the robot's body. After training, the learned model was tested and the robot's performance measured.

## 4 Implementation

### 4.1 Simulation

Training the robot to walk was done in the virtual robtoics experimentation platform (V-REP) simulation environment using a provided model of a hexapod robot. A remote interface was written in Python to communicate between the simulated robot and the code that performed the learning. This interface was responsible for running the training episodes, receiving state information from the robot, and sending servo motor commands to move the hexapod. Within this remote interface, the DDPG algorithm was implemented as a controller for the robot. For each episode of training and testing, the simulation is reset so that no compounding innaccuracies with the physics engine can affect the robot's movement. During an episode, the robot's state is observed before and after taking an action. To ensure the robot's actions are actually executed, regardless of the processing time of the controller, the simulation is run synchronously where the simulation time is manually stepped after the robot receives a new action command. This helps reduce errors from communication between the interface and V-REP and ensures the robot moves the same way every time for the same set of action commands.

The V-REP simulation environment was chosen to train the robot because the physical model and physics of the robot are similar to real life interactions. Using this environment, the learned controller could be implemented in a real robot with the same physical dimensions with greater chance of success.

### 4.2 Algorithm and Architecture

**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
```

The DDPG algorithm was implemented with the Python Tensorflow machine learning library using [3].

The actor network consists of a neural network of input size equal to the state dimension and output size equal to the action dimension. The network is connected with two hidden layers of size 400 and 300. The hidden layers both include batch normalization of the inputs that feed into rectified linear units. The output layer is activated using the tanh function to produce continuous action values in the range [-1, 1] and then multiplied by the action bound to scale the actions to values usable by the robot. The target actor network is an exact structural copy of the actor network, but with different randomly initialized network weights.

For each time step in each episode, the algorithm passes the current state into the actor network and outputs an action. The elements of the action output are continuous values in the range of the action bound that correspond to the angle velocity of the joint. The action output is then sent to the robot to move the joints. After acting, the robot senses its next state and receives a reward for the action taken.

The critic network consists of a neural network with two inputs, a state input of size equal to the state dimension and and action input of size equal to the action dimension. The state input is connected to a hidden layer of size 400 with batch normalization and rectified linear units. The state hidden layer is merged with the action input to create a second hidden layer of size 300 with rectified linear units. The output is a single node that represents the Q value for the state and action inputs. The target critic network is an exact structural copy of the critic network, but with different randomly initialized network weights.

After recording the experience of the state, action, resulting action, and reward during a single step of the simulation episode, the experience is added to the experience replay buffer. Then a set of experiences are randomly sampled from the replay buffer and used to train the network all in one batch. The use of experience replays increases the sample efficiency and limits the error from temporally correlated experiences.

The critic network is trained by minimizing the loss between the current and predicted Q values. The current Q values are calculated using the Q function and the current state, action, and reward. The target critic network predicts Q values using the next state $s'$ and the next action $a'$ that is predicted by the target actor network. The actor network is trained by optimizing on the resulting action gradients from the critic network.

After training the networks, the actor and critic target networks are soft-updated by replacing a small part of each weight by the value of the original network weight. This makes the target networks change slowly and provides stability to the training values that are output from the target networks.

## 4.3 Hyperparameters

The following hyperparameters were used the implementation of the DDPG algorithm:

- $\alpha$

Learning rate of the network. This parameter controls how quickly the network learns by affecting how much the Q value is updated in each time step. The actor network uses a learning rate of 0.0001 and the critic network uses a learning rate of 0.001 (the values used in the DDPG paper).

- $\gamma$

Discount factor of the network. This paramter controls how much expected future rewards affect current Q values. The critic network uses a discount factor of 0.99 as the actions should be close to deterministic (the value used in the DDPG paper). This will make use of temporally connected states and actions which is very helpful in achieving a walking pattern for each leg of the robot.

- $\tau$

Soft update factor of the target network. This parameter controls how much the target networks are affected by the learned model weights of the main networks. After training, the target network weights are replaced by the result of $(1-\tau)*\theta' + \tau*\theta$ where $\theta'$ is the value of the target network weights and $\theta$ is the value of the main network weights. The soft update factor is set to 0.001 in the implementation (the value used in the DDPG paper).

- batch size

The number of experiences to sample from the replay buffer to train. This parameter controls how many experiences are used to train the networks per time step. This parameter is generally set to a power of two, with 32 and 64 being common values (the DDPG paper used a batch size of 64). Through experimentation I found a batch size of 128 to be effective for the hexapod robot.

- replay buffer size

The maximum size of the replay buffer. This parameter controls how long the model stores experiences in the replay buffer. This affects how long the model will remember and be able to use past experiences. Through experimentation I found a replay buffer size of $10^5$ worked well (the DDPG paper used a replay buffer size of $10^6$).

## 4.4 Demonstrations

Training the hexapod to learn to walk was very time consuming, so demonstrations were used to speed up the training process. The hexapod model included with V-REP has a programmed gait controller that can control the robot to perform a specific gait. The controller sets the robot's step height, step length, and other parameters to produce a gait that is stable. The DDPG algorithm was modified to simply observe the state transitions while the robot was under the control of the programmed gait; no network training was performed. The required action for each state transition was calculated and the states and action were recorded in a replay buffer separate from the main replay buffer. Subsequent training started by filling the separate demonstration repay buffer with experiences. The rest of the training process was done exactly as before, except that the batch of experiences was sampled from the main replay buffer and the initialized demonstration replay buffer.

The paper on DDPG from Demonstration uses priority to sample from the demonstration buffer, but for simplicity my implementation samples a number of experiences from the main buffer and a number of experiences from the domstration buffer. The sum of the two sample sizes should equal the batch size parameter. After experimenting, I found that sampling 120 experiences from the main replay buffer and 8 experiences from the demonstration replay buffer (with a maximum size of 5000) worked well. By sampling some experiences from the collected demonstration experiences, the training can learn from the expert gait while still learning its own gait from training experiences.

### 4.5 Reward Shaping

The reward used to have the robot learn to walk is the change in position, or velocity, of the robot body at each time step. When the robot body moves forward it receives a positive reward, while when it moves backward it receives a negative reward. Since the reward is contrained to the movement of the robot's body, the legs are free to move forward or backward separately from the reward received. This allows the development of a cyclic motion where each leg steps forward and pulls backward to push the body forward. The reward for body movement was scaled by a factor of 10 through experimentation with different scales.

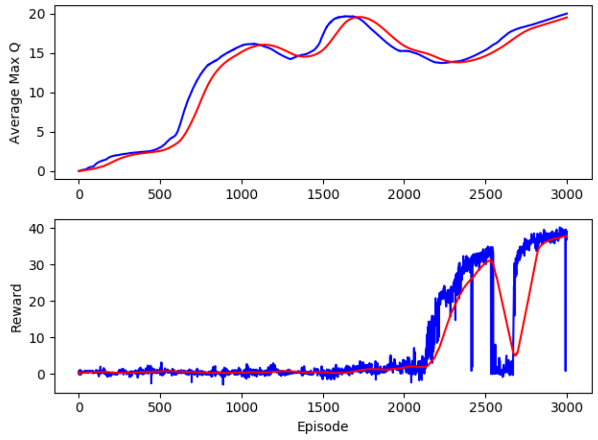
There is also a reward of -0.001 applied at each time step so that when the body is not moving the reward will accumulate a negative value. This should encourage the robot to walk by keeping it from taking actions that keep the body still.

## 5 Results

The hexapod robot was trained in three different experimental setups with different starting orienations and with or without demonstrations. The first two experiments show results with a limited random robot starting orientation. These experiments were done to limit the complexity of the task and show incremental progress.

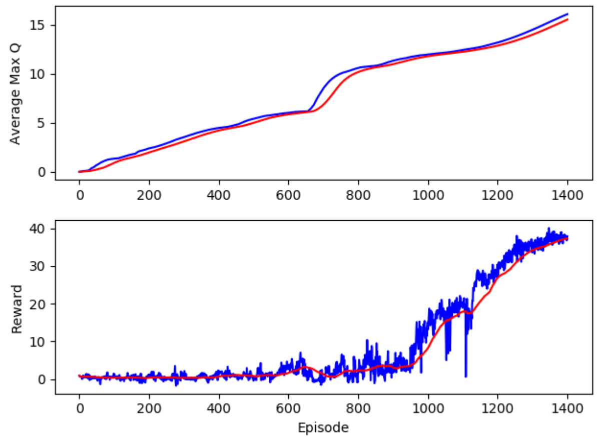
### 5.1 Limited Orientation without Demonstrations

The first experiment used a random initial orientation in range $[-30°, 30°]$ and did not use demonstration experiences. At the end of training the robot was successfully able to walk 3.5 meters to the edge of the environment. The reward graph shows a significant drop around episode 2500; this is most likely due to the robot continuing to learn after falling off the environment.



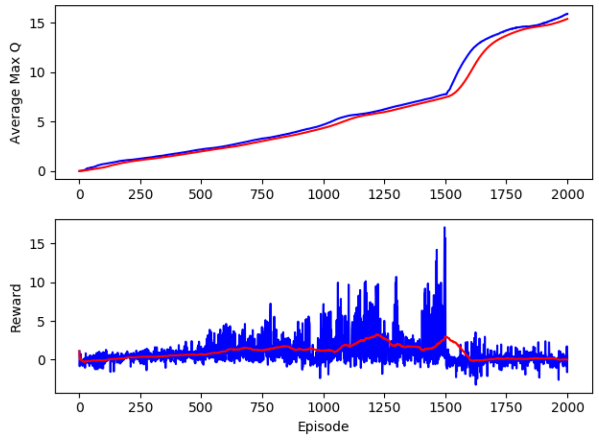
### 5.2 Limited Orientation with Demonstrations

The second experiment used a random initial orientation in range $[-30°, 30°]$ and did use demonstration experiences. At the end of training the robot was also successfully able to walk 3.5 meters to the edge of the environment. The learned robot gait is very similar to the first experiment, but the model was learned in half the time.



### 5.3 Full Orientation with Demonstrations

The last experiment used an unlimited random initial orientation in the range $[-180°, 180°]$. This is the ideal experiment since hexapods are able to move in any direction without reorienting their bodies. At the end of training the robot was unsuccessful in even walking forward consistently; it mostly took steps back and forth. The graph of episode rewards shows that the model starts to learn but diverges and is unable to learn a successful policy.



## 6 Analysis

Initially the experiments did not lead to successful results. With limited episodes of training the robot would either take a big step forward and fall or take many small steps, but not in a way that enabled it to walk far. Only after training the model with several thousand episodes did the robot learn a successful walking gait. This makes sense as the reward is small but dense. At each time step it will either receive a positive reward for moving forward or a negative reward for moving backward, moving sideways, or not moving at all. With the temporal difference learning, it would eventually learn which actions led to forward movement of the body.

The learned walking gait initially seemed unexpected, but looking back actually makes sense. For the two successful experiments, the robot learned a dynamic gait that only uses four of its legs. When compared to the static gait that the demonstration robot uses, with all six legs, the learned gait is much faster. This is most likely due to the fact that the reward function is based on the robot's velocity. Training on the reward, the robot learned that it could move faster on four legs using a gait where it was not statically balanced.

It is somewhat surprising, given the fact that the learned gait is so much different than the demonstration gait, that the demonstration experiences helped so much. Even though the robot is learning a different final set of state and action mappings, the demonstration experiences helped the robot learn twice as fast. The demonstration experiences were likely very important for exploration from the starting configuration. It seems that starting with a suboptimal set of actions was enough to get the robot moving, while still allowing it to learn the optimal walking gait.

For the experiment with unlimited random starting orientation range, the results may be unsuccessful because of a number of reasons. The model divergence, shown in the graph of episode reward over time, may recover after a number of episodes or may not even happen in another test. Another reason it failed to learn may be because of the increase in state space. In this case, a larger demonstration buffer or a larger demonstration batch size could be used to include more sample experiences in the full orientation range.

## 7 Conclusions

For this project I implemented the Deep Deterministic Policy Gradient (DDPG) algorithm as a controller for a hexapod robot to have it learn how to walk without any knowledge of its physical model. The results show that the robot was able to successfully learn to walk within a constrained state space, but was unable to learn a successful model with the full state space.

## 8 Future Work

There are many ways to continue this project to have it succeed on the original goal and acheive further goals.

### 8.1 Full Movement

To accomplish the goal of learned walking from any starting orientation, I would fully implement the DDPG from Demonstrations paper with the prioritized replay. Picking the experiences to train on that have the most temporal difference error would allow the model to learn the most it could from each batch of experiences.

### 8.2 Alternate Gaits

Experimenting with reward shaping could be done to acheive specific gaits instead of acheiving an overall goal. For example, adding a reward penalty for each time step that the robot does not have three feet on the ground could be help produce a walking gait that is static. This would be useful in applications where the terrain is unknown and the robot's safety is more important than its speed. Also, added a reward penalty for excessive rotation of the body in the x and y axes could produce a gait where the robot could carry something on it's body without it falling off. There are many different applications for hexapod robots and reward shaping can be used to adapt the gait to these applications.

### 8.3 Crossing the Reality Gap

The ultimate goal is the use the learned model on a real robot. Assuming the use of a real robot with identical physical dimensions to the V-REP model, the model learned on simulation may work in the real world. If there are major discrepancies between the simulated and real robot, the algorithm could even be used to learn in the real world so that the model can take into account factors that are not present in the simulation.

## 9 Bibliography

1. T. Lillicrap, J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. In Proc. of ICLR, 2016.

2. M. Vecerik, T. Hester, J. Scholz, F. Wang, O. Pietquin, B. Piot, N. Heess, T. Rothrl, T. Lampe, and M. A. Riedmiller. Leveraging demonstrations for deep reinforcement learning on robotics problems with sparse rewards. CoRR, abs/1707.08817, 2017.

3. Emami, P. (2016, August 21). Deep Deterministic Policy Gradients in TensorFlow [Blog post]. Retrieved from https://pemami4911.github.io/blog/2016/08/21/ddpg-rl.html