How to Train Your Dragon: Example-Guided Control of Flapping Flight

JUNGDAM WON, JONGHO PARK, KWANYU KIM, and JEHEE LEE, Seoul National University

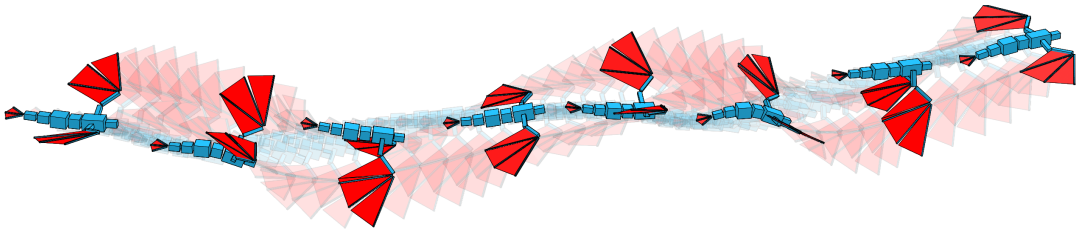

Fig. 1. The imaginary dragon learned to fly through repeated experiences. The dragon is physically simulated in realtime and interactively controllable.

Imaginary winged creatures in computer animation applications are expected to perform a variety of motor skills in a physically realistic and controllable manner. Designing physics-based controllers for a flying creature is still very challenging particularly when the dynamic model of the creatures is high-dimensional, having many degrees of freedom. In this paper, we present a control method for flying creatures, which are aerodynamically simulated, interactively controllable, and equipped with a variety of motor skills such as soaring, gliding, hovering, and diving. Each motor skill is represented as Deep Neural Networks (DNN) and learned using Deep Q-Learning (DQL). Our control method is example-guided in the sense that it provides the user with direct control over the learning process by allowing the user to specify keyframes of motor skills. Our novel learning algorithm was inspired by evolutionary strategies of Covariance Matrix Adaptation Evolution Strategy (CMA-ES) to improve the convergence rate and the final quality of the control policy. The effectiveness of our *Evolutionary DQL* method is demonstrated with imaginary winged creatures flying in a physically simulated environment and their motor skills learned automatically from user-provided keyframes.

CCS Concepts: •**Computing methodologies** → **Animation;** *Physical simulation; Reinforcement learning; Neural networks;*

Additional Key Words and Phrases: Character Animation, Physics Simulation, Physics-based Control, Reinforcement Learning, Deep Learning, Neural Network, Flapping Flight

ACM Reference format:
Jungdam Won, Jongho Park, Kwanyu Kim, and Jehee Lee. 2017. How to Train Your Dragon: Example-Guided Control of Flapping Flight. *ACM Trans. Graph.* 36, 4, Article 198 (July 2017), 13 pages.
DOI: 10.1145/3130800.3130833

1 INTRODUCTION

The beauty of flapping flight comes from the complex interaction of gravity, muscle actuation, and aerodynamics. In computer animation, winged creatures are expected to perform a variety of motor skills including soaring, gliding, hovering, taking off, and landing, though developing such a motor skill is still a daunting task. Even with an imaginary creature, its motor skills are desired to be physically realistic and controllable in a way that its wingbeats generate lift/thrust force and steer the flying direction. Balancing during flapping flight is far more challenging than fixed-wing flight.

Reinforcement Learning (RL) for generating adaptive motor skills has attracted great attention in computer animation. Given the current state of a character, the character is provided with a set of actions (or a continuous spectrum of actions) to choose from. The control policy (or controller) decides the best action to take, which results in a subsequent state and a reward associated with the state transition. The goal of reinforcement learning is to find an optimal policy, which maximizes the sum of expected future rewards.

Recently, Deep Reinforcement Learning (DRL), which combines reinforcement learning with deep neural networks, has demonstrated its potential in physics-based control and simulation. It is not obvious if deep reinforcement learning would generalize to deal with professional-quality characters with many degrees of freedom, given that we do not have any training data for the motion of imaginary creatures. The motor skill is previously defined by leveraging immediate rewards of taking actions at each state. Specifying rewards can be cumbersome from the viewpoint of animators and practitioners, who might want to have direct control over the motor skills. It is furthermore not obvious how to define a diversity of motor skills in the framework of reinforcement learning.

In this paper, we present a control method for flying creatures, which are simulated aerodynamically in three-dimensional virtual environments, controlled interactively, and equipped with a variety of motor skills. Each motor skill is represented as deep neural networks, which construct a mapping from observed states to joint actions with continuous control. The user may provide a sequence of keyframes, from which our system constructs a control policy for steering the flying direction and performing a specific motor skill. The challenge is that deep reinforcement learning often converges to local minima with high-dimensional dynamical systems. The convergence of learning is closely related to both policy update methods and exploration strategies. Our learning algorithm is inspired by Covariance Matrix Adaptation Evolution Strategy (CMA-ES), which is a stochastic, derivative-free method for numerical optimization of

DOI: 10.1145/3130800.3130833

non-linear or non-convex continuous optimization problems. Our novel algorithm combines the smart exploration strategy of CMA-ES with the generality of Deep Q-Learning (DQL), which is one of DRL methods, to improve the convergence rate and the final quality of the control policy substantially. The effectiveness of our *Evolutionary DQL* method will be demonstrated with imaginary winged creatures and their motor skills learned automatically from user-provided keyframes.

2 RELATED WORK

Computer graphics researchers have explored a variety of motion control methodologies for simulated virtual characters in the past few decades. Achieving visual realism, responsiveness to interaction, robustness to external perturbation, and adaptability to condition change are the design goals of physically based simulation and control. Although biped control have gained the most attention [de Lasa et al. 2010; Ha and Liu 2014; Lee et al. 2010, 2014; Liu et al. 2016; Sok et al. 2007; Wang et al. 2012; Yin et al. 2007], nonhuman creatures such as quadrupeds [Coros et al. 2011], swimming fishes [Grzeszczuk et al. 1998; Tan et al. 2011], flying birds [Ju et al. 2013; Wu and Popović 2003], and imaginary deformables [Coros:2012,Tan:2012] have also been studied.

2.1 Trajectory Optimization

Many optimal control methods focus mainly on generating a single trajectory that minimizes a certain energy function. The key challenge of trajectory optimization (a.k.a. spacetime optimization) is the handling of discontinuous events in the trajectory. Derivative-free, sampling-based optimization methods, such as downhill simplex and CMA-ES, are proven to be quite effective for trajectory optimization [Al Borno et al. 2013; Ha and Liu 2014; Sok et al. 2007; Tan et al. 2011]. Alternatively, contact-invariant optimization leverages fictional force at contact points to yield smooth energy landscape, for which fast, derivative-based optimization methods can be exploited [Mordatch et al. 2012]. Energy-optimal motion control is often fragile at the presence of unexpected perturbation because the effort for control is minimized. Wang et al [2010] leverages stochastic optimal control to improve the robustness of the optimized trajectory at extra energy cost.

2.2 Model-Based Controller Design

Many locomotion controllers have an underlying control model that drives the dynamics system to move on. The most popular model is a phase-based state machine, which has keyframes at state nodes and rules for phase transitioning [Yin et al. 2007]. Data-driven control leverages motion capture to improve visual realism and provide fine-level control over locomotion [Lee et al. 2010]. Alternatively, simulated controllers can be designed based on the principle of least effort, seeking optimal control policies that minimize either total joint torques or metabolic energy expenditure [Lee et al. 2014; Wang et al. 2010, 2012].

2.3 Direct Policy Learning

The controller can be viewed as a direct mapping π from state s to action a such that $a = \pi(s)$. Such a direct policy can be constructed from a collection of state-action observations (s_i, a_i) via regression. Sok et al [2007] collected state-action observations from the motion capture of human locomotion. Simply tracking motion capture data does not reproduce the original locomotion because the data may include modeling and acquisition errors. They employed trajectory optimization to rectify motion capture data and then applied locally-weighted linear regression to construct a control policy. Levine and his colleagues [2016; 2014] presented guided policy search methods that combine the flexibility of trajectory optimization and the generality of function approximation by solving them alternately. Their control policy is represented as neural networks. Trajectory optimization based on iterative LQG generates a collection of long-term, high-quality training data for neural network learning, and the learned policy thus obtained provides a new set of trajectories to be optimized for the next iteration. They applied their guided policy search methods to end-to-end visuomotor robot arm manipulation tasks and learned policies that maps raw image observations directly to torques at the robot's motors. Mordatch and Todorov [2014] presented a similar method that uses the Alternating Direction Method of Multipliers (ADMM) to couple the neural network and trajectory optimization. Tan et al [2014] developed a control system for controlling a human character to ride a bicycle in a physically simulated environment. They employed CMA-based trajectory optimization to design feedforward controllers for simulating short-term energetic motions, while a neural network evolution method is used to learn direct control policies for simulating balance-driven motions.

2.4 Reinforcement Learning in Computer Graphics

Reinforcement learning assumes a Markovian decision process. Control policy π decides which action to take at any state s, which then result in transitioning to a subsequent state s' and a reward r. Instead of constructing a direct mapping between states and actions, reinforcement learning generates an optimal control policy that maximizes the sum of expected future rewards, meaning that future plans are taken into account for the control policy. The expected future rewards and the control policy are often represented as function approximators, which are called value and policy functions, respectively. Reinforcement learning has been used successfully for designing character controllers with motion capture data [Lee and Lee 2004; Lee et al. 2010; Treuille et al. 2007] and for planning the motion of physically simulated bipeds [Coros et al. 2009].

2.5 Deep Reinforcement Learning for Control

Deep reinforcement learning employs deep neural networks as function approximators. Human-level Atari game play has been achieved by using state-action value approximators of each discrete action and replay buffers [Mnih et al. 2015; Schaul et al. 2015]. Having both state-value and policy functions as deep neural networks allows the policy to be updated by computing analytic gradients (i.e. policy gradient). Schulman and his colleagues [Schulman et al. 2015a,b] showed that choosing step-size carefully for policy gradient methods is crucial to ensure stable and steady improvements of the policy. Silver and his colleagues [Heess et al. 2015; Lillicrap et al. 2015; Silver et al. 2014] demonstrated standard RL benchmarks for continuous control (e.g. cart pole balancing, arm swing-up, arm

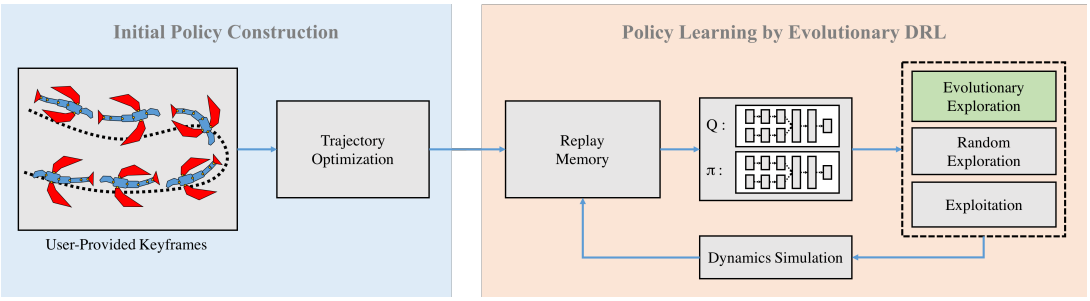

Fig. 2. System Overview.

reaching) using DRL. Peng and his colleagues [2016] learned terrain-adaptive locomotion skills of planar bipeds and quadrupeds using their dynamic states and terrain descriptions as input and parameterized steps and jumps as output actions. They also demonstrated 3D biped locomotion controller that can navigate along the target by using hierarchical deep reinforcement learning framework, where the low-level controller deals with the balance and style while the high-level controller deals with high-level goals [Peng et al. 2017].

3 OVERVIEW

An illustrative overview of our framework is shown in Figure 2. The input to our system is keyframe animation $\{\mathbf{q}_1, \cdots, \mathbf{q}_T\}$ of a creature specified by novice users or professional animators, where $\mathbf{q}_t$ is the pose at time t and T is the length of the animation. The output of the system is a control policy $\pi(\mathbf{s})$ that maneuvers the creature in a physically plausible manner. Here, $\mathbf{s}$ is the sensorimotor state of the creature, which includes its dynamic state (i.e., joint angles and its velocities) and the environment state (e.g., the location of targets and obstacles). The environment state is task-dependent, so defined differently for each individual task and control policy.

The system dynamics $p(\mathbf{s}_t, \mathbf{a}_t)$ leads to the next state $\mathbf{s}_{t+1}$, given current state $\mathbf{s}_t$ and action $\mathbf{a}_t$. Our winged creature is composed of rigid bones connected by joints and thin shells attached to the bones (see Figure 3). We assume that aerodynamic forces act only on the thin shells and are then transferred to the articulated body system of dynamics (i.e. one-way coupling). Each thin shell is shaped like an airfoil, thus aerodynamic forces can be computed by the drag and lift force equations,

$$\mathbf{f}_d = \frac{1}{2}\rho \mathbf{v}^2 A C_d \qquad \text{and} \qquad \mathbf{f}_l = \frac{1}{2}\rho \mathbf{v}^2 A C_l \tag{1}$$

where $\mathbf{v}$ is the relative velocity to the air and A is the area of the thin shell. C_l and C_d are lift and drag coefficients that vary in accordance with the *angle of attack*. We set the coefficients similarly to previous studies [Ju et al. 2013; Wu and Popović 2003]. The drag force $\mathbf{f}_d$ simulates resistance to the air and the lift force $\mathbf{f}_l$ simulates the air pressure difference between upside and downside of the airfoil, which is caused by Bernoulli's principle.

Action $\mathbf{a}$ in our system is represented as a target pose that the creature follows during the next time step. Proportional Derivative (PD) servos are used to generate torques at actuated joints to track the target pose, while the root of the skeleton and passive joints remain unactuated. Although we can represent action as torques acting at joints directly, the use of PD servos achieves better motor skills in our experiments. Similar observations can also be found in previous studies [Mordatch and Todorov 2014; Peng and van de Panne 2016].

The keyframe animation provided by the user usually includes several wingbeats that propel the creature straight ahead. In practice, the creature simulated with the keyframe animation will loose its balance and fall down immediately, because hand-crafted animation is almost always physically implausible. The trajectory optimization module in Figure 2 transforms the user-provided animation into a physically plausible one, which can be reproduced by open-loop forward dynamics simulation with aerodynamic forces. The animation thus obtained provides us with a collection of experience tuples $(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1})$, which record successful execution of actions. The experience tuples serve as initial training data for control policy learning.

Reinforcement learning makes use of both exploitation and exploration strategies. The exploitation strategy develops the estimation of future rewards by making the best use of the information currently available, while the exploration strategy probes unseen states and actions to gather more information to improve the control policy. Although our initial training data have very limited information on flying straight ahead, the exploration strategy perturbs the initial policy to learn how to turn, soar, glide, and dive. During the learning process, finding a good balance between exploitation and exploration is essential. If exploitation is dominant, the learned policy tends to be suboptimal because it could not fully observe better choices. If exploration is dominant, newly-gathered information continuously changes the estimation of future rewards before it develops the true values and thus hinders the policy from convergence.

In this paper, we utilize Deep Q-Learning (DQL) as a basis [Lillicrap et al. 2015; Mnih et al. 2015], which is a variant of reinforcement learning that represents both state-action value functions (a.k.a. Q-value functions) and policy functions as deep neural networks. In theory, DQL is able to handle control systems with high-dimensional sensory inputs and control outputs. However, achieving stable convergence is very challenging in practice because of many reasons. In this paper, we identified two key observations to improve the convergence of learning with high-dimensional dynamic systems. First, the robustness of the initial policy generated by trajectory optimization is critical to the convergence of learning. Trajectory

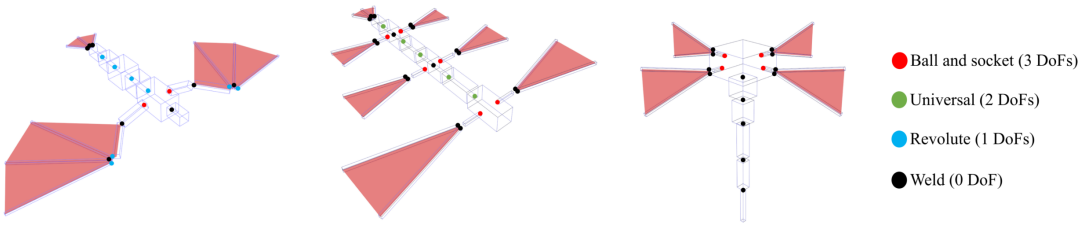

Fig. 3. The dynamic models of our creatures: (from left to right) Dragon, six-winged worm, and four-winged spore.

optimization typically defines an energy function and constructs an output trajectory that minimizes the energy function. As pointed out by Wang et al [2010], this optimal-energy trajectory could be fragile even for small perturbation since the simulated creature would put barely sufficient effort to actuate its joints, but not sufficient to resist against unexpected perturbation. This frailty of the optimal trajectory hinders exploration in RL because random perturbation of the initial policy would fail very frequently. In the next section, we will discuss how to make the optimized trajectory robust against random perturbation by employing the idea of stochastic optimal control.

Secondly, repeatedly perturbing straight ahead flight with random noise would eventually discover new motor skills such as soaring and hovering if unlimited computation resources are provided. However, exploration by such a random strategy would be extremely slow and thus policy learning can benefit from smarter exploration strategies. We found that the evolutionary strategy of CMA-ES supplements the random strategy to explore unseen states and actions rapidly and stably. Unlike previous studies that alternate between trajectory optimization and direct policy learning [Levine and Koltun 2014; Mordatch and Todorov 2014], we plug the evolutionary strategy directly into DQL as its part, so we can make use of intermediate samples in evolution as well as the final optimized result. This type of fusion is feasible only with sampling-based optimization such as CMA-ES and achieves substantial speedup and better convergence over the standard DQL. We will explain the use of CMA-ES for stochastic trajectory optimization in Section 4, followed by our *Evolutionary DQL* in Section 5 that takes the key component of CMA-ES for improving DQL.

4 INITIAL POLICY CONSTRUCTION

Given user-provided keyframe animation, the primary goal of trajectory optimization is to transform it into physically valid animation while maximizing rewards. Physically valid animation can be represented as a sequence of actions $\{\mathbf{a}_1, \cdots, \mathbf{a}_T\}$ and initial state $\mathbf{s}_0$. Forward dynamics simulation applies system dynamics recursively to generate a simulated trajectory $\{\mathbf{s}_0, \cdots, \mathbf{s}_T\}$ such that

$$\mathbf{s}_i = p(\mathbf{s}_{i-1}, \mathbf{a}_i) \tag{2}$$

for $i = 1, \cdots, T$. The secondary goal is to have the simulated trajectory resilient against random perturbation such that the deviation of the simulated trajectory is bounded within a certain range when random noises are added to the actions.

4.1 Rewards

The reward function is a cumulative sum of immediate rewards over the duration of the animation,

$$r(A, S) = \sum_{i=1}^{T} r(\mathbf{s}_{i-1}, \mathbf{a}_i, \mathbf{s}_i), \tag{3}$$

where A is a sequence of actions $\{\mathbf{a}_1, \cdots, \mathbf{a}_T\}$ and S is a sequence of the resultant states $\{\mathbf{s}_0, \cdots, \mathbf{s}_T\}$. The immediate reward function involves five terms.

$$r = r_{\text{target}} + r_{\text{collision}} + r_{\text{effort}} + r_{\text{balance}} + r_{\text{regul}} \tag{4}$$

The first term r_{target} drives the creature towards its target position.

$$r_{\text{target}} = -w_1 \|\mathbf{p}\|^2, \tag{5}$$

where $\mathbf{p}$ is the position of the target with respect to the body local coordinate system. $r_{\text{collision}}$ penalizes interpenetration through obstacles.

$$r_{\text{collision}} = \begin{cases} -w_2 d^2, & \text{if collision occurs} \\ 0, & \text{otherwise} \end{cases} \tag{6}$$

where d is the penetration depth. The third term r_{effort} minimizes the effort of flying.

$$r_{\text{effort}} = -w_3 \|\tau\|^2, \tag{7}$$

where τ is the vector of torques generated by PD servos. The fourth term r_{balance} penalizes abrupt rotational movements to keep the creature balanced during flight.

$$r_{\text{balance}} = -w_4 \|\omega\|^2 - w_5 \|1.0 - \mathbf{v} \cdot \mathbf{u}\|^2, \tag{8}$$

where $\mathbf{v}$ and $\mathbf{u}$ are the up-vectors of the skeleton root and the world, respectively, and ω is the angular velocity at the root. r_{regul} is a regularization term that prevents large deviations from the user-provided keyframe animation.

$$r_{\text{regul}} = -w_6 \sum \|\mathbf{a}_i - \mathbf{q}_i\|^2, \tag{9}$$

where $\mathbf{q}_i$ are frames of the user-provided animation.

4.2 Trajectory Optimization

The primary goal can be achieved by formulating the problem as nonlinear optimization taking actions as parameters to be optimized. We employ CMA-ES for the optimization, which is usually formulated to minimize a certain energy function. Conversely, in this paper, CMA-ES maximizes rewards according to the convention of RL and shares the same reward function with RL in the next section. CMA-ES is an iterative procedure that begins with initial action

sequence $A_0 = \{\mathbf{a}_1, \cdots, \mathbf{a}_T\}$. The whole action sequence over duration $[1, T]$ is treated as a single point in a high-dimensional space. The optimization procedure takes a collection of random action sequences over the same duration by perturbing the initial one with a multivariate normal distribution, where μ_0 is its mean and Σ_0 is the initial covariance matrix (see Algorithm 1, line 4). Each action sequence A_j over the duration generates a simulated trajectory S_j via forward dynamics simulation (line 5). The resultant trajectories are evaluated with respect to the reward function to select a subset of best trajectories (line 6-8). The mean and covariance of the multivariate normal distribution are then updated based on the subset (line 9). New action sequences for the next generation will be selected from the updated normal distribution. The algorithm repeats this procedure until the reward converges (line 10-11). The user-provided keyframe animation driven by PD servos serves as the initial action sequence, which transforms gradually as the iteration proceeds.

Algorithm 1 Covariance Matrix Adaptation Evolution Strategy

μ_0 : initial mean
Σ_0 : initial covariance
N_g : maximum # of generations
N_s : # of trajectories generated in each generation

1: **procedure** CMA-ES
2: **for** $i = 1, \cdots, N_g$ **do**
3: **for** $j = 1, \cdots, N_s$ **do**
4: $A_j \leftarrow \mathcal{N}(\mu_{i-1}, \Sigma_{i-1})$
5: $S_j \leftarrow p(s_0, A_j)$
6: $r_j \leftarrow \mathrm{r}(A_j, S_j)$
7: $R_i = \max\{r_1, \cdots, r_{N_s}\}$
8: $\Phi_i \leftarrow$ Choose N_b best actions among $\{\mathbf{A}_1, \cdots, \mathbf{A}_{N_s}\}$
9: $(\mu_i, \Sigma_i) \leftarrow$ Update the distribution by Φ_i
10: **if** $\|R_i - R_{i-1}\| < \epsilon$ **then**
11: Break

4.3 Optimization under uncertainty

Achieving the secondary goal requires further modification of the optimization procedure to account for the stability of each individual action sequence. If the action sequence is stable, its reward value would not vary significantly at the present of small perturbation [Wang et al. 2010]. Therefore, improving the stability of A_j entails maximizing the expected rewards at the neighbor of A_j. To account for the modification, line 4-6 are replaced with a procedure that we sample a few random actions $\{A'_k\}$ from $\mathcal{N}(A_j, \mathrm{diag}(\sigma))$ and then compute the expected reward $r_j = E[r(A'_k, S'_k)]$, where each reward $r(A'_k, S'_k)$ is computed from an independently simulated trajectory $S'_k = p(s_0, A'_k)$. σ is related to the robustness expected for the resulting trajectory. Larger value of σ tends to make the trajectory more robust, though excessively large values of σ would make it converge with difficulty.

5 EVOLUTIONARY DEEP Q-LEARNING

The initial control policy provides us with very little information considering the enormity of the state and action spaces. We now discuss how to generalize the information such that the learned policy can respond well at arbitrary situations with continuous control.

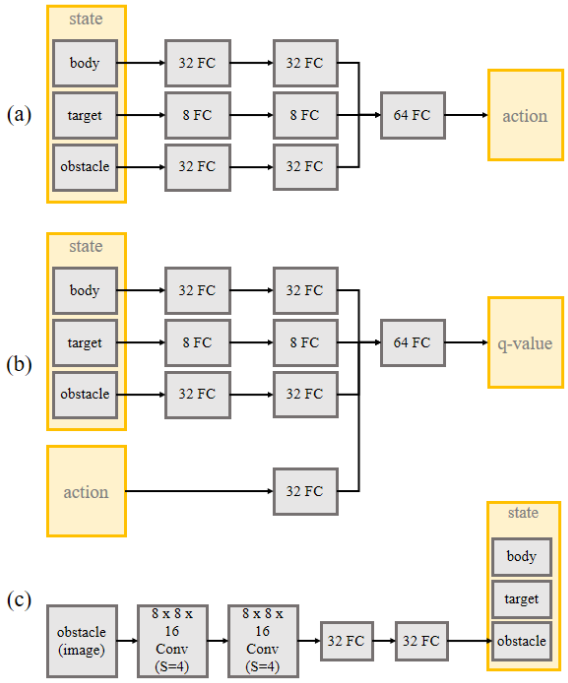

Fig. 4. The structure of Neural Networks. (a) The policy network, (b) the state-action value (Q) network, and (c) convolutional layers for visumotor control. Noth that the convolutional layers are plugged to the sensory inputs of the policy and Q networks.

The reinforcement learning is defined by an agent and an environment. Given state $\mathbf{s} \in \mathbb{S}$ of an agent, action $\mathbf{a} \in \mathbb{A}$ is chosen by its control policy $\pi : \mathbb{S} \rightarrow \mathbb{A}$. Execution of the action brings the agent to a new state $\mathbf{s}' \in \mathbb{S}$ and the agent receives a reward $r : \mathbb{S} \times \mathbb{A} \times \mathbb{S} \rightarrow \mathbb{R}$ according to the desirability of the state transitioning. The goal of reinforcement learning is to find the optimal policy that maximizes the sum of expected future rewards $\mathcal{R} = \sum_{i=0}^{\infty} \gamma^i r_i$, where discount factor $\gamma \in [0, 1)$ is introduced to prevent the infinite sum from diverging. If we can predict the future rewards at any state, then determining the optimal policy becomes a simple matter. We can simply compare the sum of the immediate reward and its future rewards for all possible actions, then select the action that maximizes the sum. This idea is called Bellman backup. An approximator of future rewards $V(\mathbf{s})$ at state $\mathbf{s}$ is called a state value function. If $V(\mathbf{s})$ stands for the true value, its recursive relation can be written as follows.

$$V(\mathbf{s}) = \max_{\mathbf{a} \in A}(r(\mathbf{s}, \mathbf{a}, \mathbf{s}') + \gamma V(\mathbf{s}')), \tag{10}$$

where A is a set of all possible actions at the state $\mathbf{s}$. We can construct the optimal state value function by iteratively replacing the left-hand side value with the right-hand side value for all states. Similarly, we can define a state-action value function that approximates the

future reward at any state-action pairs.

$$Q(\mathbf{s}, \mathbf{a}) = r(\mathbf{s}, \mathbf{a}, \mathbf{s}') + \gamma \max_{\mathbf{a}' \in A'} Q(\mathbf{s}', \mathbf{a}'), \tag{11}$$

where A' is a set of all possible actions at the subsequent state $\mathbf{s}'$. The benefit of this formulation is that we can directly determine the optimal policy as follows.

$$\pi(\mathbf{s}) = \operatorname*{argmax}_{\mathbf{a} \in A} Q(\mathbf{s}, \mathbf{a}). \tag{12}$$

Note that simulating the system dynamics for all possible actions is necessary with Equation 10, but not with Equation 12. A reinforcement learning method based on Equation 11 and 12 is called *Q-learning*.

5.1 Representation

The state and action of our system have continuous values. For continuous value problems in Equation 12, finding the optimal policy from the state-action value function requires yet another optimization in the action domain. To prevent this, we maintain a policy function $\pi(\mathbf{s}) : \mathbb{S} \rightarrow \mathbb{A}$ and a state-action value function $Q(\mathbf{s}, \mathbf{a}) : \mathbb{S} \times \mathbb{A} \rightarrow \mathbb{R}$ separately, approximating both as deep neural networks with weights θ_π, θ_Q, respectively [Lillicrap et al. 2015; Silver et al. 2014]. Both networks have the same structure of three fully connected layers with tanh units (see Figure 4). Two separate layers at the front learn the features of dynamic states and environment/task states independently. The two layers are combined and then followed by another fully connected layers. The size of output layers varies according to the dimension of $\pi(\mathbf{s})$ and $Q(\mathbf{s}, \mathbf{a})$. Each network has approximately 10k free parameters.

5.2 Evolutionary Exploration

Our learning method is episodic and trajectory-centric. In each episode, our creature is provided with a new goal (e.g. a target position) and performs a sequence of actions to achieve the goal. It may or may not achieve the goal successfully. While doing so, our creature generates a simulated trajectory from the reference starting position to the target position. We collect experience tuples from the trajectory to update the weight values of the Q and policy networks.

The key to successful learning in high-dimensional RL problems is to collect appropriate experience tuples by exploitation and exploration. The exploitation strategy performs actions according to the current policy without any perturbation, which can be understood as refining the future reward function more precisely by utilizing the known knowledge (see Algorithm 2, line 1–6). The exploration strategy performs actions with perturbation in search for better policies, which can be understood as expanding the knowledge by gathering more information (see Algorithm 2, line 7–12). In continuous action spaces, adding random noise (usually Gaussian) around the current policy is a standard method for exploration because of its simplicity. However, this could make the policy remain in poor local minima especially in high-dimensional state/action spaces due to the curse of dimensionality.

The exploitation and random exploration strategies would fail to generate good trajectories in most episodes in the early phase of learning, because the control policy has not been trained yet. The RL methods are capable of learning even from failing experiences to some extent. So the control policy can make progress with a few successful episodes and many unsuccessful episodes although the learning rate will be rather slow. We found that the evolution strategy of CMA-ES can be used seamlessly for better exploration (see Algorithm 2, line 13–21). CMA-ES is a very powerful optimization method, which performs well in most episodes regardless of the progress of policy learning. Learning from success would be much faster than learning from failure. The evolutionary strategy explores the state-action space rapidly with far-reaching episodic goals.

One might worry about the performance of the evolutionary strategy in the RL framework because CMA-ES itself is computationally demanding. In the process of evolutionary optimization, a number of episodic trajectories are generated via forward dynamics simulation at each generation and thrown away afterwards. Fortunately, all the computation is not wasted, but can be reused for policy learning. We collect experience tuples from all intermediate trajectories as well as the final optimal trajectory. Therefore, a single episode using evolutionary exploration generates many coherent experiences, which provide momentum to steer the control policy in a desired direction. We will demonstrate in the experimental results that learning with evolutionary exploration converges faster and ends up with better control policies.

Algorithm 2 Exploitation and Exploration

1: **procedure** Exploitation
2: $E = \emptyset$
3: **for** $i = 1, \cdots, T$ **do**
4: $\mathbf{a}_i \leftarrow \pi(\mathbf{s}_i|\theta_\pi)$
5: $\mathbf{s}_{i+1} \leftarrow p(\mathbf{s}_i, \mathbf{a}_i)$
6: $E \leftarrow E \cup \{(\mathbf{s}_i, \mathbf{a}_i, r_i, \mathbf{s}_{i+1})\}$
7: **procedure** RandomExploration
8: $E = \emptyset$
9: **for** $i = 1, \cdots, T$ **do**
10: $\mathbf{a}_i \leftarrow \pi(\mathbf{s}_i|\theta_\pi) + \mathcal{N}(\mathbf{0}, \mathrm{diag}(\sigma))$
11: $\mathbf{s}_{i+1} \leftarrow p(\mathbf{s}_i, \mathbf{a}_i)$
12: $E \leftarrow E \cup \{(\mathbf{s}_i, \mathbf{a}_i, r_i, \mathbf{s}_{i+1})\}$
13: **procedure** EvoExploration
14: $E = \emptyset$
15: **for** $i = 1, \cdots, N_g$ **do**
16: **for** $j = 1, \cdots, N_s$ **do**
17: $\{\mathbf{a}_1, \cdots, \mathbf{a}_{N_s}\} \leftarrow \mathcal{N}(\mu_{i-1}, \Sigma_{i-1})$
18: **for** $k = 1, \cdots, T$ **do**
19: $\mathbf{s}_{k+1} \leftarrow p(\mathbf{s}_k, \mathbf{a}_k)$
20: $E \leftarrow E \cup \{(\mathbf{s}_i, \mathbf{a}_i, r_i, \mathbf{s}_{i+1})\}$
21: $(\mu_i, \Sigma_i) \leftarrow$ Update the distribution by the elite set

5.3 Learning Algorithm

The learning algorithm begins with two separate networks $\pi(\mathbf{s})|_{\theta_\pi}$, $Q(\mathbf{s}, \mathbf{a})|_{\theta_Q}$ with weights θ_s, θ_Q (see Algorithm 3). The weights are set by Xavier initialization, which is a fully automatic method for the network initialization [Glorot and Bengio 2010]. The algorithm generates a new episode to gather experience tuples (line 2–10) and updates the networks using training data thus collected (line 11–22).

For each episode, a new task and a new environment are set randomly and then we choose a strategy probabilistically among exploitation, random exploration, and evolutionary exploration to address the task. Probabilities by ρ_0 and ρ_1 are decided such that the ratio of contribution of exploitation, random exploration, and evolutionary exploration becomes 3:1:1. This ratio was set by experimentation to prompt stable convergence of learning, while allowing conservative and rapid strategies to explore unseen states and actions in a balanced manner. Sometimes, we need to subsample experiences from evolutionary exploration, which may generate too many episodic trajectories in the evolutionary process depending on its parameter setting (e.g., the number of samples in each generation and the maximum of generations). The subsampling is a stochastic process, wherein highly-rewarded trajectories are more likely to be selected using a Boltzman distribution.

Algorithm 3 Learning

$Q|_{\theta_Q}$: state-action value network
$\pi|_{\theta_\pi}$: policy network
B : experience replay memory

1: **repeat**
2: The environment/task states are initialized randomly
3: $\rho \leftarrow U(0,1)$
4: **if** $\rho \leq \rho_0$ **then**
5: $E \leftarrow$ EvoExploration
6: **else if** $\rho \leq \rho_1$ **then**
7: $E \leftarrow$ RandomExploration
8: **else**
9: $E \leftarrow$ Exploitation
10: Put E into the replay memory B
11: $X_Q, Y_Q \leftarrow \emptyset$
12: $X_\pi, Y_\pi \leftarrow \emptyset$
13: **for** $i = 1, \cdots, N$ **do**
14: Sample an experience $\mathbf{e}_i = (\mathbf{s}_i, \mathbf{a}_i, r_i, \mathbf{s}'_i)$ from B
15: $y_i \leftarrow r_i + \gamma Q(\mathbf{s}'_i, \pi(\mathbf{s}'_i|\theta_\pi)|\theta_Q)$
16: $X_Q \leftarrow X_Q \cup \{(\mathbf{s}_i, \mathbf{a}_i)\}$
17: $Y_Q \leftarrow Y_Q \cup \{y_i\}$
18: **if** $y_i - Q(\mathbf{s}_i, \pi(\mathbf{s}_i|\theta_\pi)|\theta_Q) > 0$ **then**
19: $X_\pi \leftarrow X_\pi \cup \{\mathbf{s}_i\}$
20: $Y_\pi \leftarrow Y_\pi \cup \{\mathbf{a}_i\}$
21: Update Q by (X_Q, Y_Q)
22: Update π by (X_π, Y_π)
23: **until** no improvement on the policy

All the experiences are stored in a replay memory, which plays an important role in training the networks unbiasedly (line 10). When training the networks, a mini-batch of random samples from the replay memory are used instead of most recent experiences in order to break the temporal correlations by mixing recent and past experiences [Mnih et al. 2015]. Otherwise, the bulk of subsequent experiences makes it difficult for the networks to converge. Only actions that have positive temporal difference (TD) errors are used to update the policy network, following the implementation of CACLA [van Hasselt and Wiering 2007], while the Q network is updated regardless of the sign of their TD errors (line 15–22). Similar to [Mnih et al. 2015], separate target networks are used to compute y_i to stabilize convergence. The weights of the target networks are updated to the latest values in every 50 iterations.

Fig. 5. The synthetic vision images (64x64 pixels) from the viewpoint of the dragon in flight. The depth images were taken from the z-buffer of the OpenGL rendering pipeline.

5.4 Visuomotor Control

A visuomotor skill is the ability to synchronize visual information with physical movements. The visuomotor policy is a mapping from sensory images to actions. This is also called *End-to-End* control because it generates control outputs directly from raw observations. End-to-end visuomotor control is similar to how real animals sense the world and control their body.

We can use synthetic vision images taken from the viewpoint of the creature to simulate visuomotor skills. The creatures are assumed to be able to recognize obstacles from synthetic depth images (see Figure 5). The depth images are fed into Convolutional Neural Network (CNN) layers, which are connected to the front end of the policy/Q networks (see Figure 4). Although end-to-end learning of the deep neural networks is possible in principle using our algorithm, the actual computation of learning is substantial beyond the computing power of a single PC. The CNN has approximately 40k free parameters to learn, which is four times larger than the policy/Q networks.

A pragmatic solution is to learn visual perception separately from motor control and then end-to-end learning is used only to fine tune the whole visuomotor networks. Conceptually speaking, the creature first learns motor skills based on the geometric coordinates of the obstacles and then learns how the obstacles would look in the synthetic vision later. We used a small number of obstacles (ten in our experiment) to pre-train the policy/Q networks without CNN layers as explained so far. Each obstacle is represented by the coordinates of its center and the radius. The number of obstacles is fixed throughout the learning process because the number affects the structure of the neural networks. We then attach a virtual camera to the creature, and simulate the creature by using the pre-trained policy network in randomly initialized environments. Data (state, action, and value) are collected during the simulation, and then we use it for training the extended networks with CNN layers at the front end in a supervised learning fashion. In the learning process, only the weights of CNN layers are updated while the other parts of the networks remain intact. In our experiment, we collected 100,000 data and trained with 32 batch size.

Note that the motor policy using geometric coordinates does not generalize well if the new environment has more obstacles than

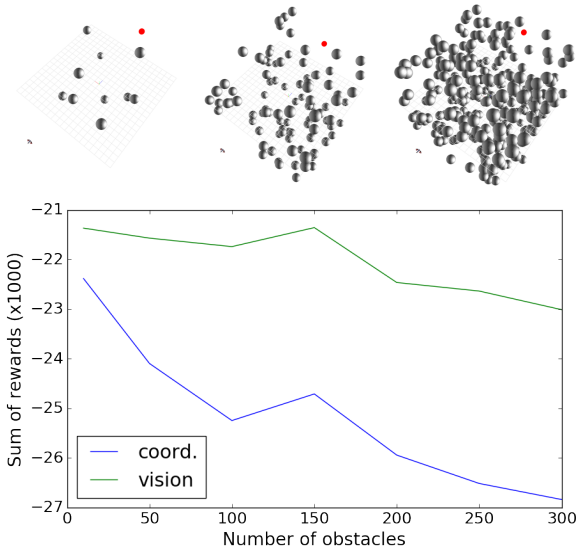

Fig. 6. Visuomotor control experiments with the density of the obstacles. A red circle is the target and gray circles are the obstacles. We select the closest ten obstacles as inputs to the network using geometric coordinates, whereas such process is not needed for the visuomotor network.

Table 1. Creature details and parameters

Creature	Dragon	Worm	Spore
DoFs (actuable)	22	34	18
Mass (kg)	104	92	24
Length (meter)	5.0	5.2	7.4
Width (meter)	9.1	5.3	7.4
Height (meter)	0.6	0.4	8.6
dt (simulation)	0.001	0.001	0.001
dt (control)	0.2	0.2	0.2
Gravity (N/kg)	9.81	9.81	9.81
Density (ρ)	1.275	1.275	1.275
PD gains (wing)	10^5	10^5	10^4
PD gains (body)	5×10^4	2×10^4	N/A
Learning rate (π)	0.001	0.001	0.001
Learning rate (Q)	0.001	0.001	0.001
Discount factor (γ)	0.99	0.99	0.99
Time horizon (s)	20	20	20
w_1 (target tracking)	0.01	0.01	0.01
w_2 (collision avoidance)	500	500	500
w_3 (effort minimization)	10^{-7}	10^{-8}	10^{-7}
w_4 (angular velocity)	0.5	0.5	0.5
w_5 (upright position)	20	20	20
w_6 (regularization)	0.05	0.03	0.05

the training environment. More importantly, the coordinate-based encoding is order-dependent, meaning that different numberings of obstacles would generate different sensory inputs regardless of its actual geometric configurations. This issue is not present with pixel-based vision inputs. Even though visuomotor learning takes the geometric coordinates as a supervisor, the visuomotor skill scales better with the generalization of the environment conditions. Figure 6 shows the performance comparison of control policies at the presence of progressively more obstacles. The visuomotor policy (green plot) deals better with dense obstacles than the motor policy with coordinate inputs (blue plot), because there is no prior assumption in visual perception as to the structure of obstacles.

6 RESULTS

Our algorithm was implemented in Python, using DART [Dart 2012] for the simulation of articulated body dynamics and TensorFlow [TensorFlow 2015] for the training and evaluation of deep neural networks. The dynamics and learning parameters are summarized in Table 1. The same values for simulation timestep (0.001 second), control timestep (0.2 second), gravity (9.81 N/kg), density (1.275), learning rate of π/Q (0.001), and discount factor (0.99) are used for all examples. The reward weight values for target tracking (w_1), collision avoidance (w_2), and upright position (w_5) are also fixed, whereas the weights for effort minimization (w_3), angular velocity (w_4), and regularization (w_6) depend on the choice of model and physics parameters. All processes were run on a PC with Intel Core i7 6700K (4 cores, 4.0GHz). We did not utilize the computing power of GPUs because the computational cost is dominated by dynamics simulation rather than deep neural network operations.

6.1 Creature Models

We created three creature models. The dragon has a long trunk, two side wings, and a tail wing (see Figure 10). The trunk is composed of five body segments connected in a row. The side wings are attached at each side of the trunk, and the tail wing is attached at the end of the trunk. The side wings play a major role of propelling the body forward, while the tail wing supplements the function of side wings for fine maneuvers. The dragon is 5.0 meters long, 9.1 meters wide, and 0.6 meters tall. We tested the dragon while varying its weights from 25 kg to 104 kg. The physical properties are inspired by an extinct species called *Pelagornis Sandersi*, which is known as the largest flying bird ever discovered. Our dragon of weight 104 kg is twice as big and three times heavier than the largest ever flying bird. The six-winged gigantic worm has three rows of wings and a body longer than the dragon. The four-winged spore is a light-weight flapping creature that can float easily in the air. The body density (mass per volume) of the spore is approximately three and six times lower than the dragon and the worm, respectively.

6.1.1 Dragon. Our dragon is provided with four sets of keyframes for energetic wingbeats of a large span, gentle wingbeats of a narrower span, gliding, and diving (see Figure 7 (a)-(d)). Interestingly, motor skills learned from different initial trajectories use different strategies to carry out a given task depending on the power per weight. The control policy of a light (25 kg) dragon using energetic wingbeats allows the dragon to maneuver rapidly and soar up easily towards the target, while the control policy of a heavy (104 kg) dragon using gentle wingbeats tends to make circular flying patterns and takes spiral paths to move upwards. The dragon is circling

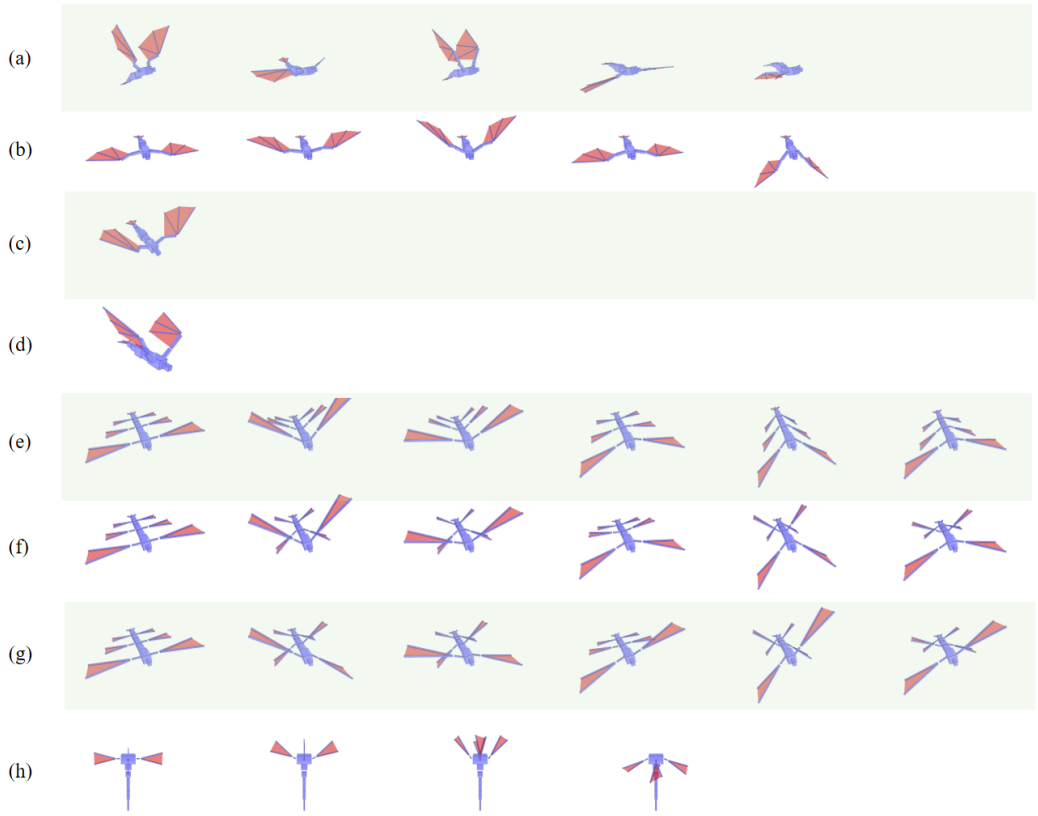

Fig. 7. User-provided keyframes.

around the target rather than going to the target directly, which is mainly due to the long time-horizon of our experiments. In such a setting, circling around the target is preferred to minimize the long-term sum of distance because it is hard for the dragon to float at a fixed location. The behavior could be modified by setting the training environment being terminated when the dragon reaches the target or by using sharper reward function with narrow support. Although the gliding and diving policies were learned using only one keyframe, the policies are also capable of steering its direction. Switching between motor skills is immediate and effortless. The dragon can change its motor skill without any delay or preparation steps, because the neural network policies map any state to an action desired at the moment.

6.1.2 Gigantic Worm. The worm learned three control policies from the user-provided keyframes (see Figure 7 (e)-(g)). With the first policy, three rows of the wings flap synchronously in the same direction. With the second policy, the wings at even and odd rows flap in opposite directions. Alternatively, the worm twists its body to propel forwards rather than flapping its wings with the third policy. The worm learned all three motor skills successfully to stylize its locomotion patterns.

6.1.3 Flapping Spore. Since all of its wings align radially facing upwards, moving up and down is easy for the spore, but it has to tilt the entire body to move horizontally. The locomotion style learned from the user-provided keyframes makes it spin around along the vertical axis for each wingbeat, resulting in a screw-like motion.

6.2 Learning

Constructing an initial control policy uses CMA-ES with its maximum generation of 100 and the population size of 8 at each generation. We sampled 10 random neighbors to estimate the expected rewards at each sample. Therefore, we run at most $100 \times 8 \times 10$ simulations, which is computationally demanding. Fortunately, individual episodic simulations are independent of each other and thus can be computed in parallel on CPU multi-cores. It takes about 10 to 20 minutes with four cores. Figure 8 shows the impact of stochastic optimization that minimizes expected rewards under uncertainty. Initial control policies were tested repeatedly at the presence random perturbation. Scattering of the simulated trajectories indicates the robustness of the control policies. The optimized policy using the standard CMA-ES easily looses its balance and falls down even for small perturbation (Figure 8 (Bottom)). The optimization with stochasticity makes the wings stroke more energetically and thus resilient against perturbation (Figure 8 (Top)).

The evolutionary strategy can be paired with many existing policy update methods. Our algorithm in the previous section utilized the CACLA-style TD update because it worked well with flying creatures. Learning with evolutionary exploration takes about 25 hours to achieve the stable controller. Figure 9 shows the comparison of our evolutionary DQL with the base DQL. The plots indicate the convergence rates while the dragon learns its first control policy. The base DQL fell into a local minimum and failed to escape. Our evolutionary DQL not only converged faster but also discovered better control policy at the convergence. The difference is apparent in the supplementary video, wherein the dragon learned by our method is equipped with various motor skills that generate agile maneuvers. This agility cannot be achieved with the base DQL.

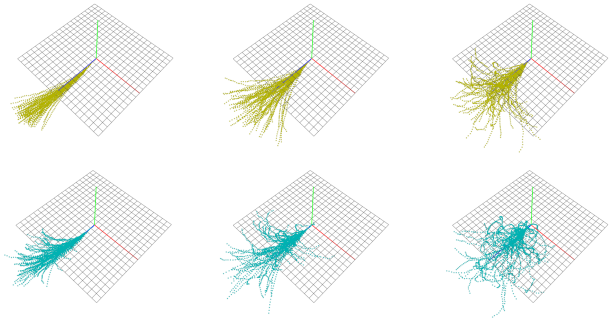

Fig. 8. Robustness comparison. The magnitude of perturbation increases from left to right. (Top) The initial policy optimized with stochasticity. (Bottom) The initial policy optimized with standard CMA-ES.

6.3 Benchmarks

To evaluate the effectiveness of our evolutionary strategy, we compared well-known DRL methods for continuous control with their evolutionary versions in OpenAI Gym [Brockman et al. 2016]. The benchmark tests used three environments: *Swimmer-v1*, *HalfCheetah-v1*, and *HumanoidStandup-v1*. *Swimmer* is a two-dimensional snake-like creature with 8 observation dimensions representing its physical states and 2 action dimensions representing instantaneous joint torques. *HalfCheetah* is a two-dimensional two-legged creature with 17 observation dimensions and 6 action dimensions. The most challenging benchmark is *HumanoidStandup* with 376 observation dimensions and 17 actions dimensions. Its goal is to make a two-legged humanoid to stand up from the decubitus (lying) position.

Three base methods, CACLA [van Hasselt and Wiering 2007], DDPG [Silver et al. 2014], and TRPO [Schulman et al. 2015a], were selected for benchmarking. The performance of the RL methods for continuous control depends on the body structure, degrees of freedom, and actuation types of the creatures. TRPO is the current state-of-the-art in OpenAI Gym environments. We used the implementation of DDPG and TRPO by Duan et al [Duan et al. 2016]. We also implemented variants of CACLA and DDPG by plugging the evolutionary strategy, called Evo-CACLA and Evo-DDPG. Incorporating a new exploration strategy into TRPO is not straightforward because TRPO does not exploit replay buffers.

Figure 11 depicts the plots of average rewards for each environment. In our benchmark tests, both Evo-CACLA and Evo-DDPG consistently converged faster with better policies than their base methods for all environments. The advantage of exploiting the evolutionary strategy varies depending on the level of difficulty of learning. The simplest creature, the swimmer with two joints, benefits marginally from the smarter exploration strategy and thus the convergence depends largely on the choice of the policy update methods. The creatures with many joints benefit more to make substantial improvements. Evo-DDPG performs comparablely or better than TRPO in HalfCheetah and HumanoidStandup, though the convergence of Evo-DDPG is not as steady as the semi-monotonic convergence of TRPO. The convergence graphs of evolutionary methods fluctuate comparably to the graphs of their base methods. It means that our rapid evolution strategy does not cause instability beyond the characteristics of the base methods.

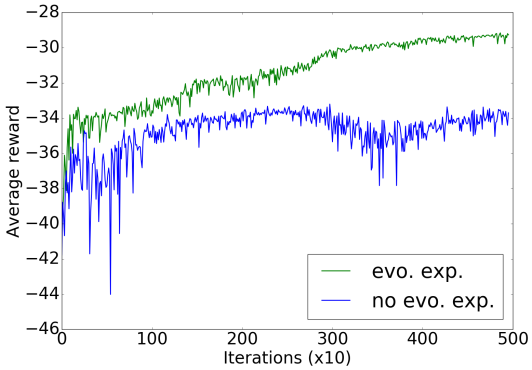

Fig. 9. The convergence of DQL with and without evolutionary exploration.

7 DISCUSSION

Our physically simulated creatures learned how to fly using deep reinforcement learning. The key challenge was to have the policy and Q networks converge stably, while exploring unseen states and actions to generalize the capability of the initial motor skills. Achieving stable convergence and prompting rapid exploration are often contradictory goals. There are several sources of risks that may hinder stable convergence of the learning algorithm. First, theoretically, many reinforcement learning algorithms with general function approximators are not guaranteed to converge or their convergence condition is not known yet. Therefore, careful parameter tuning is necessary in practice. Secondly, the forward dynamics simulation also has a notorious stability issue. The time integration would proceed stably only when the time step is sufficiently small. Using small time steps requires more computation for learning. Lastly, the balance recovery capability of the control policy is yet another source of instability that may impede exploration by perturbation. These seemingly-orthogonal issues are actually related with each other in practice. Assuming that the simulation time step is fixed, simulation and balance stability depend largely on the exploration rate, which is proportional to the magnitude of the random noise. Abrupt movements by jerky noise can make the simulation unstable

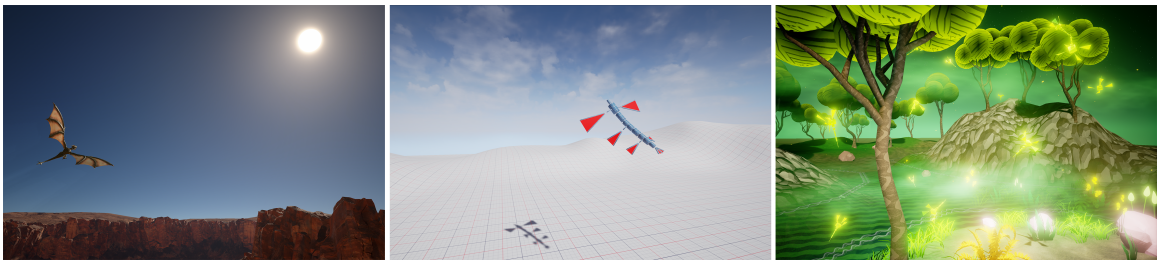

Fig. 10. Screenshots of flying creatures.

and cause the creature to lose its balance. Conversely, conservative settings for simulation would slow down the rate of exploration. Our evolutionary strategy addresses both issues, rapid exploration and simulation/balance stability, simultaneously.

Some of previous studies on continuous control make use of parametrized action models, which represents a family of cyclic actions with a small number of parameters [Ju et al. 2013; Peng et al. 2016]. The unit of action is a cycle of locomotion. Unlike those previous studies, we learn a mapping from state to action on a per-frame basis. Given a state, the control policy returns the desired actuation (joint angles) at the immediate moment. This nonparametric action model has advantages over the parametrized approaches. Most importantly, new motor skills may emerge, while parameterized approaches search policies within bounded action space. Since the parametrized action defines the phase of locomotion cycles, transitioning between control policies is usually allowed only at the beginning of a new cycle. In contrast, the nonparametric approach allows immediate transitioning between control policies regardless of the phase of locomotion.

The quality of the initial reference trajectory substantially influences the quality of the simulated motor skills. Currently, we used only five keyframes to design the initial trajectory, which may be not sufficient to describe natural looking motions. A remedy to the problem is the use of a realistic (either mocap or handcrafted by a professional artist) reference trajectory. Several recent RL studies have demonstrated that a successful control policy can be learned with random initialization. Although they showed interesting and emergent behaviors, the resultant motions are lacking in controllability, which is important especially in graphics applications. The users or animators want to have control over the style of resultant motion. Given an arbitrary body structure, many flapping patterns are feasible. We rather want to give the animators control over the resulting patterns with guidance, which may narrow down the scope of emergent behaviors to some extent to a given pattern.

A state-action value function worked better than a state value function in our experiment, although the two functions could work similarly with the use of CACLA-style update. We think that the explicit update for the actions (especially for the samples by the Evo-Exp) in Equation 11 made a difference in practice. Understanding the reason theoretically or testing other update methods would help us to look into the deeper aspects of the problem.

Even though our focus has been on flapping flight in this paper, our approach can be extended to deal with other types of locomotion, such as swimming, legged locomotion, and even limbless crawling. The strength of reinforcement learning lies mainly on its generalization capability that can deal with arbitrary body structures, sensors, actuators, motor skills, and environmental conditions.

There are many other exciting directions for future research. Learning the sensorimotor skill of a skeletal model with musculotendon units requires a mapping from sensory inputs to muscle excitation signals [Lee et al. 2014; Wang et al. 2012]. Deep learning would be a promising solution for muscle-based control considering the complexity of sensorimotor connection and the multiplicity of control layers. We are also interested in improving the simulation environment with more accurate aerodynamics simulation [Tan et al. 2011]. Turbulent air flow around the wings allows interesting maneuvers in bird flight, such as rapid break at large angle of attack. Such maneuvers cannot be learned without accurate simulation of turbulent wake and vortices. Learning other motor skills for flying creatures (take-off, landing, and flocking behavior) are also a challenging problem that has not been addressed by reinforcement learning yet.

ACKNOWLEDGMENTS

This research was supported by the MSIT(Ministry of Science and ICT), Korea, under the SW Starlab support program(IITP-2017-0536-20170040) supervised by the IITP(Institute for Information & communications Technology Promotion)

REFERENCES

Mazen Al Borno, Martin de Lasa, and Aaron Hertzmann. 2013. Trajectory Optimization for Full-Body Movements with Complex Contacts. *IEEE Transactions on Visualization and Computer Graphics* 19, 8 (2013).

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. OpenAI Gym. (2016).

Stelian Coros, Philippe Beaudoin, and Michiel van de Panne. 2009. Robust Task-based Control Policies for Physics-based Characters. *ACM Trans. Graph. (SIGGRPAH 2009)* 28, 5 (2009).

Stelian Coros, Andrej Karpathy, Ben Jones, Lionel Reveret, and Michiel van de Panne. 2011. Locomotion Skills for Simulated Quadrupeds. *ACM Trans. Graph. (SIGGRPAH 2011)* 30, 4 (2011).

Dart. 2012. Dart: Dynamic Animation and Robotics Toolkit. https://dartsim.github.io/. (2012).

Martin de Lasa, Igor Mordatch, and Aaron Hertzmann. 2010. Feature-based locomotion controllers. *ACM Trans. Graph. (SIGGRAPH 2010)* 29, 4 (2010).

Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. 2016. Benchmarking Deep Reinforcement Learning for Continuous Control. *CoRR* abs/1604.06778 (2016).

Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATSfi10).*

Radek Grzeszczuk, Demetri Terzopoulos, and Geoffrey E. Hinton. 1998. NeuroAnimator: Fast Neural Network Emulation and Control of Physics-based Models. In *Proceedings of International Conference on Computer Graphics and Interactive Techniques*

Fig. 11. Benchmark results of Swimmer-v1 (Top), HalfCheetah-v1 (Middle), and HumanoidStandup-v1 (Bottom).

(SIGGRAPH 1998). 9–20.

Sehoon Ha and C. Karen Liu. 2014. Iterative Training of Dynamic Skills Inspired by Human Coaching Techniques. *ACM Trans. Graph.* 34, 1 (2014).

Nicolas Heess, Gregory Wayne, David Silver, Timothy P. Lillicrap, Tom Erez, and Yuval Tassa. 2015. Learning Continuous Control Policies by Stochastic Value Gradients. In *Annual Conference on Neural Information Processing Systems (NIPS 2015)*. 2944–2952.

Eunjung Ju, Jungdam Won, Jehee Lee, Byungkuk Choi, Junyong Noh, and Min Gyu Choi. 2013. Data-driven Control of Flapping Flight. *ACM Trans. Graph.* 32 (2013), 151:1–151:12.

Jehee Lee and Kang Hoon Lee. 2004. Precomputing Avatar Behavior from Human Motion Data. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. 79–87.

Yoonsang Lee, Sungeun Kim, and Jehee Lee. 2010. Data-driven biped control. *ACM Trans. Graph. (SIGGRAPH 2010)* 29, 4 (2010).

Yoonsang Lee, Moon Seok Park, Taesoo Kwon, and Jehee Lee. 2014. Locomotion Control for Many-muscle Humanoids. *ACM Trans. Graph. (SIGGRAPH Asia 2014)* 33, 6 (2014).

Yongjoon Lee, Kevin Wampler, Gilbert Bernstein, Jovan Popović, and Zoran Popović. 2010. Motion Fields for Interactive Character Locomotion. *ACM Trans. Graph. (SIGGRPAH Asia 2010)* 29, 6 (2010).

Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. 2016. End-to-end Training of Deep Visuomotor Policies. *J. Mach. Learn. Res.* 17 (2016), 1334–1373.

Sergey Levine and Vladlen Koltun. 2014. Learning Complex Neural Network Policies with Trajectory Optimization. In *Proceedings of the 31st International Conference on Machine Learning (ICML 2014)*.

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

Libin Liu, Michiel Van De Panne, and Kangkang Yin. 2016. Guided Learning of Control Graphs for Physics-Based Characters. *ACM Trans. Graph.* 35, 3 (2016).

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-level control through deep reinforcement learning. *Nature* 518 (2015), 529–533.

Igor Mordatch and Emanuel Todorov. 2014. Combining the benefits of function approximation and trajectory optimization. In *In Robotics: Science and Systems (RSS 2014)*.

Igor Mordatch, Emanuel Todorov, and Zoran Popović. 2012. Discovery of complex behaviors through contact-invariant optimization. *ACM Trans. Graph. (SIGGRAPH 2012)* 29, 4 (2012).

Xue Bin Peng, Glen Berseth, and Michiel van de Panne. 2016. Terrain-adaptive Locomotion Skills Using Deep Reinforcement Learning. *ACM Trans. Graph. (SIGGRPAH 2016)* 35, 4 (2016).

Xue Bin Peng, Glen Berseth, KangKang Yin, and Michiel van de Panne. 2017. DeepLoco: Dynamic Locomotion Skills Using Hierarchical Deep Reinforcement Learning. *ACM Trans. Grap. (SIGGRAPH 2017)* 36, 4 (2017).

Xue Bin Peng and Michiel van de Panne. 2016. Learning Locomotion Skills Using DeepRL: Does the Choice of Action Space Matter? *CoRR* abs/1611.01055 (2016).

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. 2015. Prioritized Experience Replay. *CoRR* abs/1511.05952 (2015).

John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. 2015a. Trust Region Policy Optimization. *CoRR* abs/1502.05477 (2015).

John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan, and Pieter Abbeel. 2015b. High-Dimensional Continuous Control Using Generalized Advantage Estimation. *CoRR* abs/1506.02438 (2015).

David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. 2014. Deterministic Policy Gradient Algorithms. In *Proceedings of the 31st International Conference on Machine Learning (ICML 2014)*. 387–395.

Kwang Won Sok, Manmyung Kim, and Jehee Lee. 2007. Simulating biped behaviors from human motion data. *ACM Trans. Graph. (SIGGRAPH 2007)* 26, 3 (2007).

Jie Tan, Yuting Gu, C. Karen Liu, and Greg Turk. 2014. Learning Bicycle Stunts. *ACM Trans. Graph. (SIGGRAPH 2014)* 33 (2014), 50:1–50:12.

Jie Tan, Yuting Gu, Greg Turk, and C. Karen Liu. 2011. Articulated swimming creatures. *ACM Trans. Graph. (SIGGRAPH 2011)* 30, 4 (2011).

TensorFlow. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. (2015). http://tensorflow.org/ Software available from tensorflow.org.

Adrien Treuille, Yongjoon Lee, and Zoran Popović. 2007. Near-optimal Character Animation with Continuous Control. *ACM Trans. Graph. (SIGGRAPH 2007)* 26, 3 (2007).

Hado van Hasselt and Marco A. Wiering. 2007. Reinforcement Learning in Continuous Action Spaces. In *Proceedings of the 2007 IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning (ADPRL 2007)*. 272–279.

Jack M. Wang, David J. Fleet, and Aaron Hertzmann. 2010. Optimizing Walking Controllers for Uncertain Inputs and Environments. *ACM Trans. Graph. (SIGGRAPH 2010)* 29, 4 (2010).

Jack M. Wang, Samuel. R. Hamner, Scott. L. Delp, and Vladlen. Koltun. 2012. Optimizing Locomotion Controllers Using Biologically-Based Actuators and Objectives. *ACM Transactions on Graphics (SIGGRAPH 2012)* 31, 4 (2012).

Jia-chi Wu and Zoran Popović. 2003. Realistic modeling of bird flight animations. *ACM Trans. Graph. (SIGGRAPH 2003)* 22, 3 (2003).

Kangkang Yin, Kevin Loken, and Michiel van de Panne. 2007. SIMBICON: Simple Biped Locomotion Control. *ACM Trans. Graph. (SIGGRAPH 2007)* 26, 3 (2007).