The Thirty-Second AAAI Conference
on Artificial Intelligence (AAAI-18)

Phase-Parametric Policies for Reinforcement Learning in Cyclic Environments

Arjun Sharma, Kris M. Kitani
Robotics Institute, Carnegie Mellon University
Pittsburgh, PA 15213
{arjuns2, kkitani}@cs.cmu.edu

Abstract

In many reinforcement learning problems, parameters of the model may vary with its *phase* while the agent attempts to learn through its interaction with the environment. For example, an autonomous car's reward on selecting a path may depend on traffic conditions at the time of the day or the transition dynamics of a drone may depend on the current wind direction. Many such processes exhibit a cyclic phase-structure and could be represented with a control policy parameterized over a circular or cyclic phase space. Attempting to model such phase variations with a standard data-driven approach (e.g. deep networks) without explicitly modeling the phase of the model can be challenging. Ambiguities may arise as the optimal action for a given state can vary depending on the phase. To better model cyclic environments, we propose phase-parameterized policies and value function approximators that explicitly enforce a cyclic structure to the policy or value space. We apply our phase-parameterized reinforcement learning approach to both feed-forward and recurrent deep networks in the context of trajectory optimization and locomotion problems. Our experiments show that our proposed approach has superior modeling performance than traditional function approximators in cyclic environments.

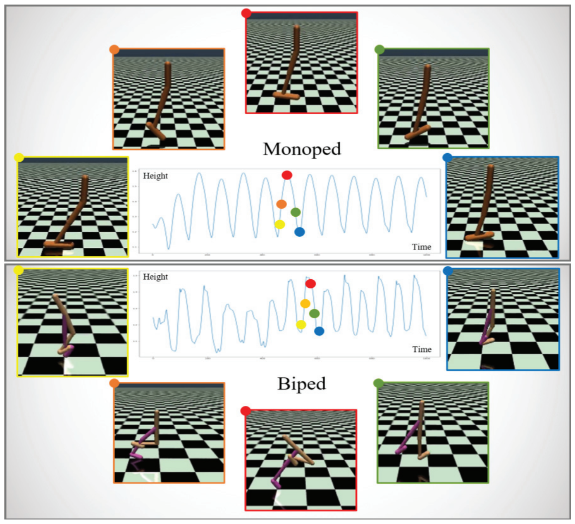


Figure 1: Many processes such as monoped (top) and biped (bottom) motion exhibit a cyclic nature. Plots in the figure depict the motion of the centre of mass of the body.

Introduction

In reinforcement learning, an agent learns by interacting with the environment. The agent takes an action and receives a reward from the environment which transitions to a new state. In many real world problems, model parameters such as reward and transition dynamics may vary while the agent is still learning through its interactions. For example, a drone may experience winds from different directions as it learns to navigate, requiring different strategies to counteract movement caused by the wind. An autonomous car may receive a high reward on choosing a particular path to its destination at one time of the day but low reward during another as traffic conditions vary. Such variations make learning with standard data-driven approaches challenging as these methods must learn to model such variations in feedback without explicitly modeling the *phase* structure of the model (e.g., wind direction, traffic conditions).

While deep learning based methods have been successfully applied to various reinforcement learning problems (Mnih et al. 2015; Schulman et al. 2015a; Mnih et al. 2016; Levine et al. 2016; Schulman et al. 2015b), most current techniques do not explicitly account for phase changes that may occur in the environment. When information regarding the phase of the environment is available (e.g., a wind sensor, clock, pedometer), a simple strategy may be to append that phase information to the state representation when learning the policy or value function. However, for high-dimensional state spaces, which are common in many real-world problems, it may be difficult for the underlying estimator (e.g., a deep neural network) to vary its policy drastically based on a single phase input variable. Another strategy may be to learn a set of independent estimators for different phases or modes of the environment. However, this approach does not scale as the number of phases increases and would also be highly sample inefficient as experience gained to learn a policy in one phase would not be used to learn the policy for another.

In this work, we look at the class of problems where the phase-structure is cyclic. Phase cycles need not follow a periodic pattern and their evolution may even be stochastic.

To solve such problems, we propose to explicitly parameterize the weights of a deep neural network conditioned on the phase and constrain the weights to lie on a cyclic manifold. Inspired by recent progress in computer graphics for generating smooth and natural human motion sequence using phase-functioned dynamics models (Holden, Komura, and Saito 2017), we develop a novel phase-parametric action-value function (e.g., Q-function or critic) and phase-parametric policy function (e.g. actor) generated from a cyclic low dimensional phase space. We show how our proposed phase dependent Q-function and policy representation can be used as function approximators in reinforcement learning algorithms. We call these specific instantiations as Phase Deep Q-Networks (Phase-DQN), Phase Deep Recurrent Q-Networks (Phase-DRQN) and Phase Deep Deterministic Policy Gradient (Phase-DDPG).

We address reinforcement learning problems where the reward function and transition dynamics change during learning in cyclic patterns. We also consider the problem of simulating monoped and biped locomotion where motion of the agent generates an approximately cyclic pattern in the observed states as shown in Figure 1. We introduce a novel phase-functioned recurrent neural network architecture for problems with varying model parameters which require memory. Experiments on both discrete and continuous state-action spaces show that the proposed Phase-Parametric Networks outperform traditional networks with phase as additional input for reinforcement learning problems with cyclic environments.

Related Work

Phase Neural Networks Holden, Komura, and Saito (2017) recently introduced *Phase-functioned neural networks* for interactive character motion generation. A Phase-functioned neural network is a multi-layer perceptron, the weights of which are smoothly varied as a function of phase. While Holden, Komura, and Saito (2017) use this architecture in a supervised learning setting for smooth motion generation, we use this architecture to overcome ambiguities which arise in reinforcement learning problems when different actions are optimal in the same environment state due to a change in reward function or transition dynamics as the environment evolves. We also extend the concept to recurrent architectures.

Deep Q-Networks Deep Q-Networks (DQN) have been used most notably for playing video games. Mnih et al. (2015) first used DQNs on the Atari 2600 testbed to show that DQNs stabilized with tricks such as using experience replay and target networks can outperform human players on a variety of games. Van Hasselt, Guez, and Silver (2016) further improved results on this testbed by using Double Q-learning to train deep networks to address the issue of overestimation of action-values. Wang et al. (2016) introduced a network with two separate streams to compute the state-values and advantages. While we use the methodology of Mnih et al. (2015), our phase-parameterized architectures can be trained with any of the more recent algorithms.

Deterministic Policy Gradient Policy gradient algorithms are commonly used for solving reinforcement learning problems with continuous action spaces. Deterministic policy gradients (Silver et al. 2014) is an actor-critic method which maintains parameterized actor and critic functions which correspond to the policy and Q-function. In this work, we use the Deep Deterministic Policy Gradient (DDPG) algorithm of Lillicrap et al. (2015) but use our phase-parameterized networks as the function approximators.

Deep Recurrent Q-Networks Hausknecht and Stone (2015) introduced Deep Recurrent Q-Networks (DRQN) which add a recurrent layer after a convolutional neural network to deal with the problem of partial observability of states. A recurrent layer provides the Q-network with a long term memory to remember events more distant than the number of screens given as input. Other works have used DRQNs to learn multi-agent communication strategies (Foerster et al. 2016) and to play first-person video games (Lample and Chaplot 2017).

Cyclic Environments Shibuya and Yasunobu (2014) train agents for environments with cyclic time dependent rewards by using sinusoidal action-value functions in phasor representation. Benbrahim and Franklin (1997) propose a self scaling reinforcement learning algorithm to deal with continuous actions for biped walking. Ogino et al. (2004) use a neural oscillator to generate rhythmic biped motion. Peng et al. (2017) add a phase variable to their state representation to learn biped locomotion using hierarchical reinforcement learning. Koppejan and Whiteson (2011) propose a neuro-evolutionary method based approach for the generalized helicopter hovering problem. In contrast, we learn phase-parameterized networks which can also be used for other problems which exhibit phase-structure and can also deal with varying transition functions. To the best of our knowledge, previous methods do not explicitly consider the problem of change in model during training.

Preliminaries

Phase-Functioned Multi-Layer Perceptron (PF-MLP). As described in Holden, Komura, and Saito (2017), the weights of the network are parameterized by the phase and control weights. The *phase* represents the current state of the model. For example, it can refer to the current direction of the wind or the level of traffic on a scale of 1 to 10. The *control weights* are the weights to be used by the network at a small number of representative phases. One can interpret the control weights as the basis vectors that span the parameter space. The weights of the network at any other phase are then computed on the basis of these control weights. For example, given weights for North and West wind directions as control weights, the weights of the network for North-West direction of wind may be computed using combination of these control weights.

Consider $x \in \mathbb{R}^{d_i}$ to be the input to the neural network and $y \in \mathbb{R}^{d_o}$ to be the corresponding target output. Let the phase be denoted by p and the set of k control weights of layer j be denoted by $\boldsymbol{\beta^j} = \{\beta^{j,0}, \ldots, \beta^{j,k-1}\}$. The weights in layer j are denoted by $W_p^j = \theta(p; \boldsymbol{\beta^j})$. For a PF-MLP with two hidden layers h_0 and h_1, and an output layer o, the entire network can be written as:

$$o = \phi_o(W_p^o \phi_{h_1}(W_p^{h_1} \phi_{h_0}(W_p^{h_0} x + b_0) + b_1) + b), \quad (1)$$

where ϕ_i is a non-linear function (e.g., ReLU, Sigmoid).

By parameterizing the weights on the phase, PF-MLP has the ability to learn a different set of weights for each phase. The smooth variation of weights with change in phase is enforced by θ. In (Holden, Komura, and Saito 2017), Catmull-Rom spline function is used as the θ function and thus varies smoothly with the phase parameter p. In particular, using four control points yields:

$$\begin{aligned}\theta(p; \boldsymbol{\beta^j}) &= \beta^{j,k_1} + w_p(\frac{1}{2}\beta^{j,k_2} - \frac{1}{2}\beta^{j,k_0}) \\ &+ w_p^2(\beta^{j,k_0} - \frac{5}{2}\beta^{j,k_1} + 2\beta^{j,k_2} - \frac{1}{2}\beta^{j,k_3}) \\ &+ w_p^3(\frac{3}{2}\beta^{j,k_1} - \frac{3}{2}\beta^{j,k_2} + \frac{1}{2}\beta^{j,k_3} - \frac{1}{2}\beta^{j,k_0}),\end{aligned} \quad (2)$$

where $w_p = w(p) = \frac{4p}{2\pi} \mod 1$ and $k_n = \lfloor \frac{4p}{2\pi} \rfloor + n - 1 \mod 4$. Notice here that the modulo function is what makes the spline function cyclic.

The control weights can be learned through a supervised process by backpropagating an appropriate loss function.

Let L denote the loss function to be optimized and let $\partial L / \partial W_p^j$ denote the gradient of the loss L with respect to the weight W_p^j. Then, gradients for the control weights can be computed using the chain rule as the partial derivative of the loss with respect to the weights at phase p and the partial derivative of those weights with respect to the basis weights:

$$\frac{\partial L}{\partial \beta^{j,i}} = \frac{\partial L}{\partial W_p^j} \frac{\partial W_p^j}{\partial \beta^{j,i}}. \quad (3)$$

The second term in the chain rule expansion above can be computed analytically from equation 2, to yield the following:

$$\begin{aligned}\frac{\partial W_p^j}{\partial \beta^{j,k_0}} &= -\frac{1}{2}w_p + w_p^2 - \frac{1}{2}w_p^3, \\ \frac{\partial W_p^j}{\partial \beta^{j,k_1}} &= 1 - \frac{5}{2}w_p^2 + \frac{3}{2}w_p^3, \\ \frac{\partial W_p^j}{\partial \beta^{j,k_2}} &= \frac{1}{2}w_p + 2w_p^2 - \frac{3}{2}w_p^3, \\ \frac{\partial W_p^j}{\partial \beta^{j,k_3}} &= \frac{1}{2}w_p^2 + \frac{1}{2}w_p^3.\end{aligned}$$

Phase-Parameteric Functions

For problems with cyclically evolving environments, we propose to explicitly enforce a phase structure to the value space and policy space. We focus on the use of neural networks as function approximators for the value function and policy and describe how phase information can be embedded into the parameter space. In particular, we look at how our phase-parameterized networks can be used as function approximators in Q-Learning for discrete action spaces and as actor-critic networks for continuous actions.

Phase Deep Q-Network

In order to learn the weights of a Q-function approximator in an reinforcement learning setting, we use Q-learning (Watkins and Dayan 1992). We use the PF-MLP as a phase parameterized action-value function and denote it as the Phase Deep Q-Network (Phase-DQN). A Phase-DQN takes state s and phase p as input and outputs a vector of action values $Q(s, \cdot; p, \boldsymbol{\beta})$ where $\boldsymbol{\beta}$ is the set of control weights.

When using (deep) neural networks as function approximators, a 'target' network is often used to stabilize learning and create targets for learning (Mnih et al. 2015). The control weights of the target network $\hat{\boldsymbol{\beta}}$ are copied every τ steps from the online network and kept fixed in between these updates. The target for learning can then be written as:

$$y_s^a = r_{s,s'}^a + \gamma \max_{a'} Q(s', a'; p', \hat{\boldsymbol{\beta}}),$$

where s' and p' are the next state and phase respectively and $r_{s,s'}^a$ is the reward received by the agent on taking action a in state s and reaching s'. A mean squared error loss can then be used to create the loss function for learning as $L = \frac{1}{2}(Q(s, a; p, \boldsymbol{\beta}) - y_s^a)^2$. Figure 2 (left) shows a pictorial representation of the proposed Phase-DQN.

Phase Deep Recurrent Q-Network

Recurrent Neural Networks (RNNs) have the ability to retain information about previously seen inputs. Hausknecht and Stone (2015) showed how recurrent Q-networks can outperform their feed-forward counterparts in games where memory beyond a few frames of the game is required. We describe how weights of a Q-network with recurrent connections can also be parameterized by phase. In particular, we use RNNs with Gated Recurrent Units (GRU) (Cho et al. 2014) as our running example but a similar explanation follows for any other popular architecture.

In a Phase-functioned GRU (PF-GRU), weights of the GRU unit and output layer at every time step t may be different and are parameterized by the phase and control weights. In constrast, a typical GRU model uses same weights at all time steps. Consider an input sequence $X = \{x_0, \cdots, x_T\}$ with corresponding targets $Y = \{y_0, \cdots, y_T\}$ with $x_t \in \mathbb{R}^{d_i}$ denoting the input and $y_t \in \mathbb{R}^{d_o}$ denoting the target output at time t. We denote phase at time t by p_t and the set of k control weights of type $j \in \{z, r, \tilde{h}, o\}$ by $\boldsymbol{\beta^j} = \{\beta^{j,0}, \ldots, \beta^{j,k-1}\}$. Using the notation $W_{p_t}^j = \theta(p_t; \boldsymbol{\beta^j})$ for the weight of type j at time step t, the equations for the forward pass of a Phase-functioned GRU are given by:

$$\begin{aligned}z_t &= \sigma(W_{p_t}^z \cdot [h_{t-1}, x_t]), \\ r_t &= \sigma(W_{p_t}^r \cdot [h_{t-1}, x_t]), \\ \tilde{h}_t &= \tanh(W_{p_t}^{\tilde{h}} \cdot [r_t * h_{t-1}, x_t]), \\ h_t &= (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t, \\ o_t &= g(W_{p_t}^o \cdot h_t).\end{aligned} \quad (4)$$

The function θ, which maps the phase to network weights, is the Catmull-Rom spline function with four control points (equation 2). During the backward pass, let $\partial L / \partial W_{p_t}^j$ denote the gradient of the loss L with respect to weight $W_{p_t}^j$

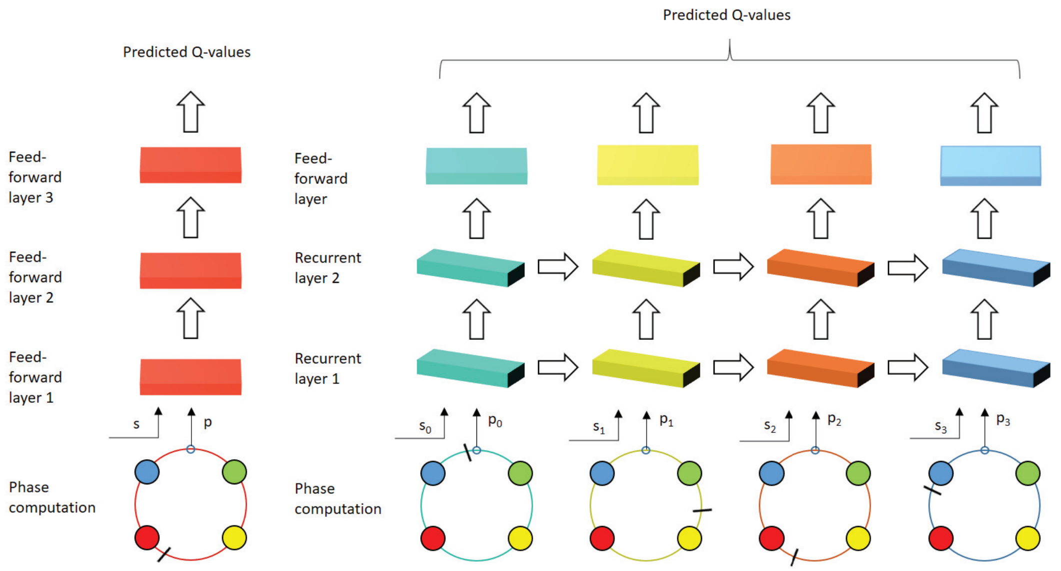


Figure 2: Phase Deep Q-Network (left) and Phase Deep Recurrent Q-Network (right). The four colored circles denote control points and their color represents the phase associated with those control points. The ring on which these circles lie represent the weight manifold. Color of the rings denote the current phase, marked by a black line on the ring. Cuboids denoting the layers are colored by the color of the ring, representing that weights are a function of phase at each time.

used at time t. Then, gradients for control weights can be computed by using the chain rule and aggregating the gradient across all time steps as:

$$\frac{\partial L}{\partial \beta^{j,i}} = \sum_t \frac{\partial L}{\partial W^j_{p_t}} \frac{\partial W^j_{p_t}}{\partial \beta^{j,i}}.$$

The second partial derivative term inside the sum can again be computed from equation 2 as before.

Again, Q-learning can be used to train a PF-GRU for reinforcement learning tasks. Such a Phase Deep Recurrent Q-Network (Phase-DRQN) takes state sequence $s = \{s_1, \dots, s_T\}$ and phase sequence $p = \{p_1, \dots, p_T\}$ as input and outputs a vector of action values $Q(s_t, h_t, \cdot; p_t, \boldsymbol{\beta})$ at each time step where $\boldsymbol{\beta}$ is the set of control weights of the network. For example, $\boldsymbol{\beta} = \{\boldsymbol{\beta}^z, \boldsymbol{\beta}^r, \boldsymbol{\beta}^{\tilde{h}}, \boldsymbol{\beta}^o\}$ in case of a network with one layer of GRUs and an output layer. Denoting control weights of the target network by $\hat{\boldsymbol{\beta}}$, the targets at every time step for learning can be written as:

$$y^{a_t}_{s_t} = r^{a_t}_{s_t, s_{t+1}} + \gamma \max_{a'} Q(s_{t+1}, a'; p_{t+1}, \hat{\boldsymbol{\beta}}).$$

By adding the mean squared error at every time step to create the loss, we get $L = \frac{1}{2}\sum_t (Q(s_t, a_t; p_t, \boldsymbol{\beta}) - y^{a_t}_{s_t})^2$. Figure 2 (right) illustrates the proposed Phase-DRQN.

Phase Deep Deterministic Policy Gradient

Policy gradient algorithms are an important class of algorithms for solving reinforcement learning problems with continuous actions. The Deterministic Policy Gradient algorithm maintains a parameterized policy function μ, called the actor and a parameterized Q-function called the critic. We use PF-MLPs for the actor and critic networks and denote the resulting algorithm as Phase-DDPG. The actor in Phase-DDPG takes the state s as input and outputs an action $a = \mu(s; p, \boldsymbol{\beta_a})$ where p and $\boldsymbol{\beta_a}$ denote the control weights of the actor and the phase. This action is then evaluated by the critic, which returns the Q-value of this state-action pair $Q(s, a; p, \boldsymbol{\beta_c})$ where $\boldsymbol{\beta_c}$ are the critic control weights. The critic is trained by minimizing the mean squared error loss with the corresponding target given by:

$$y^a_s = r^a_{s,s'} + \gamma Q(s', \mu(s'; p', \hat{\boldsymbol{\beta}}_{\boldsymbol{a}}); p', \hat{\boldsymbol{\beta}}_{\boldsymbol{c}}).$$

The actor maximizes the expected return by taking a step in the direction of the sample gradient with respect to the weights of the actor for phase p, $\boldsymbol{W} = \theta(p, \boldsymbol{\beta_a})$, given by:

$$\nabla_{\boldsymbol{W}} J \approx \sum_{s \in batch} \nabla_a Q(s, a; p, \boldsymbol{\beta_c})|_{a=\mu(s;p,\boldsymbol{\beta_a})} \nabla_{\boldsymbol{W}} \mu(s; p, \boldsymbol{\beta_a})$$

Experiments

In this section, we first describe the details of the environments we use and then analyze the results of our experiments. We evaluate our proposed approach on both discrete and continuous environments.

Discrete Problems

In order to perform controlled experiments for quantitative evaluation of our phase-parametric Q-Networks, we model two different types of discrete cyclic environments.

Freeway On-Ramp Problem (cyclic reward) In this grid world problem, there are two possible goal states. Each represents the on-ramp to a freeway. Depending on the time of day, the traffic congestion for each on-ramp changes. To simulate such an environment, we define the reward of each goal state according to a sinusoidal function. In particular, the reward at the first goal is $R_1 = A * \sin(\omega_1 t + \phi_1) + c$ and the second goal is $R_2 = A * \sin(\omega_2 t + \phi_2) + c$, where ω and ϕ are the frequency and phase offset (not to be confused with the phase of the policy), respectively. For all other locations (states), the reward is set to -1. The reinforcement learning algorithm must plan an efficient path by utilizing phase information so that it reaches an on-ramp at a certain time with the least amount of congestion. Neglecting phase information would lead to the agent receiving different rewards from the same goal state, creating ambiguity regarding the desirability of that goal state.

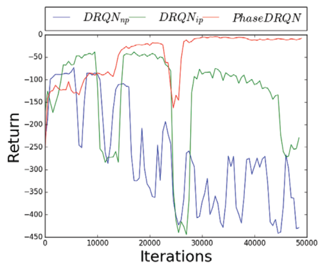
Flying with Wind Problem (cyclic dynamics) In this grid world problem, the aim is to fly a drone to a goal location in the midst of intense wind. The reinforcement learning algorithm must learn to adjust its actions depending on the wind direction, using the wind when advantageous or resisting it when necessary. To model such an environment, we change the state transition dynamics based on the wind direction. We generate the wind sequence by first sampling a wind direction from one of k possible directions. After a direction is sampled, wind continues to blow in that direction for four time steps. The wind intensity however is stochastic, exerting a push on the drone in the direction of the wind with probability p_{wind} and leaving it unaffected otherwise. This stochasticity introduces noise in the phase information which is in contrast to the previous problem. The rewards are set to 20 and -20 at the two goal states and -1 for all other states.

For each of these problems, the size of the grid world is set to 12×12. The grid world consists of two 'goal' states: $(0, 11)$ and $(11, 11)$. To make the problem suitable for experiments with recurrent architectures, we make states partially observable by including four evenly spaced obstacles which follow a periodic motion[1]. State transition dynamics are designed such that an agent cannot occupy a grid location where there is an obstacle. The agent can take one of five actions — up, down, left, right or no movement. See supplemental material for more discussion on design choices for these experiments.

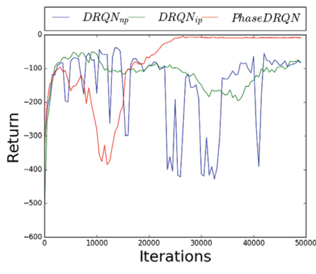
Continuous Problems

To evaluate the applicability of our approach to problems with continuous action spaces, we experiment using popular monoped and biped balancing tasks implemented using the MuJoCo physics simulator (Todorov, Erez, and Tassa 2012). The hopping motion of the monoped and the walking motion of the biped are approximately cyclic as shown in Figure 1 and we aim to see whether our proposed methods can learn better policies by taking advantage of this cyclic nature.

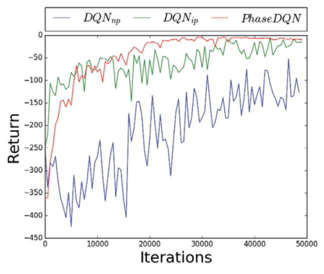
[1]To make the state fully observable for feed-forward networks, the states at current and two previous time steps were provided as input. The recurrent networks are provided with only the current state as input and must learn to retain necessary information.



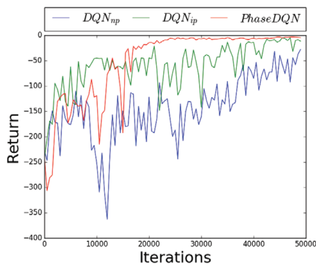
(a) $R_{8:8,\pi:0}$



(b) $R_{8:8,0:0}$



(c) $R_{8:8,\pi:0}$



(d) $R_{8:8,0:0}$

Figure 3: Performance plots for **Freeway On-Ramp** problem. The top and bottom rows show results using recurrent and feed-forward architectures respectively. The plots show the performance averaged over the previous 500 episodes, during training. Curves for baselines with no phase and input phase, and the proposed approach are shown in blue, green and red respectively.

Hopper The aim in this problem is to make a multiple degree of freedom monoped learn to move and keep from falling. The states $s \in \mathbb{R}^{11}$ and the actions $a \in [-1, 1]^3$.

Walker This task consists of making a biped walker learn to move as fast as possible without falling down. The dimensionality of the states and actions is higher than the Hopper task with $s \in \mathbb{R}^{17}$ and $a \in [-1, 1]^6$.

Comparative Baselines

Each of the proposed methods is compared against:

(1) A baseline approach in which phase is not given as input. We wish to see how networks without phase information perform on the tasks. We denote baselines of this type as DQN_{NP}, DRQN_{NP} and DDPG_{NP}, where NP stands for no phase.

(2) A baseline approach in which phase is appended to the state input. This baseline estimates how well networks provided with phase as an additional input work on the tasks. We denote these baselines as DQN_{IP}, DRQN_{IP} and DDPG_{IP}, where IP stands for input phase.

Network configuration

For both discrete problems, all networks had 2 hidden layers with 8 units in each layer and a linear output layer with 5 units (one for each action). Rectified Linear units (ReLU) were used for feed-forward architectures after every layer, except the output layer. The recurrent architectures used

	Recurrent Networks			Feed-forward Networks			
	DRQN$_{\text{NP}}$	DRQN$_{\text{IP}}$	**Phase-DRQN**	DQN$_{\text{NP}}$	DQN$_{\text{IP}}$	**Phase-DQN**	Random
$R_{8:8,\pi:0}$	33, -34.7	19, -31.5	**30, -9.9**	18, -26.2	19, -31.5	**24, -18**	203, -178.5
$R_{4:8,0:0}$	20, -14	22, -1.9	**26, 0**	21, -29.1	19, 1.1	**18, 8**	448, -442
$R_{4:8,\pi:0}$	21, 3.5	35, -10.5	**22, 4**	**21, -0.9**	18, -32	**21, -0.9**	610, -624
$R_{8:8,0:0}$	51, -26.5	37, -12.5	**23, -9.3**	17, -3.4	22, -1.9	**20, 6**	339, -314.5

Table 1: Results for the **Freeway On-Ramp** problem. Two values are reported: Number of steps, Return. Phase-DRQN and Phase-DQN are the proposed methods

		Recurrent Networks		Feed-forward Networks		Random
	p$_{\text{wind}}$	DRQN$_{\text{IP}}$	**Phase-DRQN**	DQN$_{\text{IP}}$	**Phase-DQN**	
1	0.0	**19.0, 2.0**	22.6, -1.6	**18.6, 2.4**	21.5, -0.5	496.8, -496.1
2	0.2	36.8, -12.9	**29.1, -8.1**	**18.5, 2.5**	21.9, -0.9	344.6, -344.6
3	0.4	40.3, -26.0	**25.4, -4.6**	**19.1, 1.9**	22.8, -1.8	236.8, -236.3
4	0.6	30.7, -9.8	**23.6, -2.6**	20.6, 0.4	**20.5, 0.5**	162.8, -162.6
5	0.8	31.7, -10.7	**25.0, -5.2**	24.4, -3.4	**22.1, -1.3**	116.7, -116.3
6	1.0	34.7, -13.7	**23.9, -2.9**	25.0, -4.0	**22.5, -1.8**	95.2, -94.3

Table 2: Results for the **Flying with Wind** problem with varying wind probability p_{wind} with 8 wind directions. Results are listed as Number of steps, Return. Phase-DRQN and Phase-DQN are the proposed methods.

Gated Recurrent Units for the hidden layer. All networks were trained for 50000 iterations using Adam optimization (Kingma and Ba 2014) with an initial learning rate of 10^{-4}.

We followed the settings outlined in Lillicrap et al. (2015) when implementing the baseline DDPG and our Phase-DDPG networks for the continuous problems. All networks had two hidden layers with 400 and 300 units and ReLU non-linearity. Actions were appended to the output of the first hidden layer for the critic networks. The networks were optimized using Adam with learning rates 10^{-4} and 10^{-3} for the actor and critic respectively. A replay memory of size 10^6 was used.

We refer the reader to the supplemental material for more details on the training procedure.

Freeway On-Ramp Problem Analysis

The rewards at the two goal states in this problem are given by sinusoidal functions of time. The desirability of a goal state thus depends on the current phase and its distance from the current state. An agent must learn to plan an efficient path from its starting state to the desirable goal state such that it reaches it at a time which yields the maximum possible return. Ignoring phase information may lead to ambiguous feedback since the agent would receive different rewards from the same goal state and such an agent should be unable to learn to plan a path to the most desirable goal state.

Table 1 lists the results for this problem. We report the return for an agent starting at state $(0, 5)$ as it is equidistant from the two goal states and is thus a good candidate for a state from which the optimal path is dependent on phase. The rows $R_{\omega_1:\omega_2,\phi_1:\phi_2}$ denote the different settings of the reward function at the two goal states.

DRQN$_{\text{NP}}$ and DQN$_{\text{NP}}$ agents often reach the goal state when the sinusoidal reward function is negative leading to much lower returns. Among the recurrent Q-networks, the proposed Phase-DRQN outperformed the recurrent baselines in all experiments. It is interesting to note that the proposed feed-forward Phase-DQN slightly outperformed all other models, including our recurrent model Phase-DRQN. We believe that this may be due to the explicit full observability of states in feed-forward models as compared to recurrent models where the network must learn to retain relevant information. Also, feed-forward models allow for random sampling of states for training while recurrent models are trained using highly correlated states in a single episode. The difficulty in training of recurrent policies has also been reported in prior works (Duan et al. 2016). Qualitatively, we observed that both the Phase-DRQN and Phase-DQN agents often learn to wait close to the goal state for the reward function to reach its maximum, which can also be seen by the higher number of steps in some cases for the proposed methods in Table 1.

Figure 3 shows the average return starting from random grid locations as training proceeds. Notice how curves for the proposed methods converge faster to a stable return as compared to the baselines. The variance in obtained returns towards the end of training is also smaller for the proposed approach, indicating that a more stable optimum has been reached.

Flying with Wind Problem Analysis

In this problem, a drone flying through the grid experiences a blowing wind, the direction of which changes with phase. The agent must learn to use the wind to its advantage when it is blowing towards the desirable goal state and resist it when it is pushing it towards the undesirable one. Unlike the previous problem, the phase information in this task is *noisy* due to the stochastic nature of the wind which is strong enough to exert a push with a probability p_{wind} and leaves the drone unaffected otherwise. We check robustness to noise in the phase by varying p_{wind} from 0.0 to 1.0.

Table 2 shows the results for this problem with 8 wind di-

Environment	Method						
	DDPG_{NP}		DDPG_{IP}		Phase-DDPG		
	(S)	(L)	(S)	(L)	(S)	(L)	(FT)
Hopper	2007.72	2050.00	2133.34	2559.47	1397.72	2630.16	**3325.57**
Walker	2820.99	3424.48	3333.92	3740.77	2532.24	4491.60	**5124.44**

Table 3: Average return over 100 episodes on **Hopper** and **Walker** environments. (S), (L) and (FT) denote short, long and fine-tuned training procedures respectively. Phase-DDPG is the proposed method.

rections. These results were computed by taking the mean of the performance over 1000 episodes for an agent starting at state $(0, 5)$. We select this starting state as it is equidistant and far from the goal states, allowing several phase variations. We found the baseline methods without phase information to perform poorly and do not report them for brevity.

The Phase-DRQN models outperformed the DRQN_{IP} models for all values of p_{wind} except for 0.0. This makes sense as the problem is under-determined making it difficult for the model to fit four different control weights to data from only one transition dynamic model. More importantly, we observe that the return of our Phase-DRQN agents remains largely unchanged as compared to the DRQN_{IP}, showing that our approach is more robust to noise in the phase. Without explicitly modeling the phase, the return of the DRQN_{IP} baseline varies greatly as the probability of p_{wind} varies from 0.0 to 1.0. The Phase-DQN models performed better than the DQN_{IP} models for high values of p_{wind} when the wind behaviour is more consistent with the phase. As with Phase-DRQN agents, the performance of the Phase-DQN agents does not vary much as p_{wind} is changed. The feed-forward models performed slightly better than the recurrent models, possibly because of reasons discussed previously. The superiority of the proposed Phase-Parametric Q-Networks indicates that these networks were better able to learn strategies specific to each phase. Moreover, while we used 4 control points in our architecture for the 8 phase problem, similar trends are observed when the task contains 4 phases (see supplemental material), showing that the network was able to share relevant information between phases.

Hopper and Walker Problem Analysis

In the Hopper and Walker problems, we wish to evaluate whether the approximately cyclic nature of monoped and biped motion can be leveraged by the proposed architectures to learn better policies. Apart from dealing with continuous state and action spaces, these problems also differ from the problems considered in the previous section in that we do not have knowledge of an exact phase function. Instead, we construct an approximate phase function using the desired cyclic motion of the agent. The phase is thus self-generated rather than exogenous as considered previously.

Since we did not have access to foot contact information, we constructed the phase function as follows. We assign a phase 0 whenever the centre of mass is at or below a threshold d_{low}, representing the height of the agent when in contact with the ground and assign a phase π whenever the centre of mass is at or above a threshold d_{high}, representing the highest height that the agent achieves during its motion. Heights between d_{low} and d_{high} are partitioned into $n/2$ parts where n is the desired number of discrete phases. A phase $2\pi i/n$ is assigned whenever the height of the centre of mass is greater than the i^{th} but less than the $(i+1)^{th}$ lower bound of the partition. If the vertical velocity of the centre of mass is positive, the phase assigned in the previous step is the current phase. Otherwise, the computed phase in the previous step is subtracted from 2π. The thresholds were selected using the motion of the agent trained using the DDPG_{NP} method.

Table 3 shows the return obtained on the Hopper and Walker tasks averaged over 100 episodes. We first discuss results with short training (S), where the models were trained for 10000 episodes. The proposed Phase-DDPG algorithm performed poorly compared to the baselines. We hypothesize that this is due to the self-generating nature of the phase. The control weights of the proposed model will be trained sequentially as the agent learns to walk and achieve states belonging to phases later in the cycle. This may lead to slower training. In contrast, since the baseline models use the same weights across phases, these models can learn faster. To verify this hypothesis, we trained our models for longer (40000 episodes). The results for this setting are listed under columns labeled (L). The performance of our proposed Phase-DDPG models improves greatly with longer training as the agent learns to walk and visits newer phases, allowing control weights for these to be learned. The baseline performance only improves marginally. The improved performance of our models provides evidence in support of our hypothesis.

In another experiment to test our hypothesis, we initialized the control weights of our proposed model using the weights of a baseline DDPG_{NP} model trained for 7000 episodes (same weights for all control weights) and then fine-tuned using a lower learning rate for another 3000 episodes. Note that the total episodes of training for the proposed approach with fine-tuning is the same as that for the baseline methods in the short training (S) setting. Column labelled (FT) in Table 3 shows the results after fine-tuning. The return using the proposed method with fine-tuning is significantly higher than the baseline approaches, lending further credence to our hypothesis. The fine-tuning approach even outperforms models trained for much longer, indicating that bootstrapping the proposed models with meaningful weights may be the best strategy in case of self-generating phases. Qualitatively, we observed that the agents learned using fine-tuning moved much faster and maintained better balance than the agents learned using baseline approaches.

Conclusion

Current deep reinforcement learning methods do not explicitly consider the changes that may occur in the model as the agent learns. Such changes can lead to ambiguities where different actions become optimal for the same state as the model changes. A naive solution of including the model *phase* as input often fails as it becomes difficult for the network to produce different outputs when a large fraction of the network's input is the same. In this paper, we use phase-functioned multi-layer perceptron and recurrent neural networks for solving this problem. Phase-Parametric policy and value networks based on these architectures were used on two grid world navigation tasks with phase varying reward function and transition dynamics, and two popular benchmark locomotion tasks with continuous state and actions spaces. Extensive experiments show that our Phase-Parametric networks outperform traditional networks with phase inputs on both discrete and continuous settings. Phase-Parametric networks are able to learn phase specific policies while being able to share information across phases. We show that the proposed networks are robust to noise and can be used with self-generated phases.

Acknowledgments

This work was sponsored in part by JST CREST grant (JPMJCR14E1) and NSF NRI grant (1208598).

References

Benbrahim, H., and Franklin, J. A. 1997. Biped Dynamic Walking using Reinforcement Learning. *Robotics and Autonomous Systems* 22(3-4):283–302.

Cho, K.; van Merrienboer, B.; Bahdanau, D.; and Bengio, Y. 2014. On the Properties of Neural Machine Translation: Encoder-Decoder Approaches. In *SSST@EMNLP*.

Duan, Y.; Chen, X.; Houthooft, R.; Schulman, J.; and Abbeel, P. 2016. Benchmarking Deep Reinforcement Learning for Continuous Control. In *International Conference on Machine Learning*, 1329–1338.

Foerster, J.; Assael, Y. M.; de Freitas, N.; and Whiteson, S. 2016. Learning to Communicate with Deep Multi-agent Reinforcement Learning. In *Advances in Neural Information Processing Systems*, 2137–2145.

Hausknecht, M., and Stone, P. 2015. Deep Recurrent Q-Learning for Partially Observable MDPs. *CoRR, abs/1507.06527*.

Holden, D.; Komura, T.; and Saito, J. 2017. Phase-functioned Neural Networks for Character Control. *ACM Transactions on Graphics (Proc. SIGGRAPH 2017)* 36(4):42:1–42:13.

Kingma, D. P., and Ba, J. 2014. Adam: A Method for Stochastic Optimization. *CoRR* abs/1412.6980.

Koppejan, R., and Whiteson, S. 2011. Neuroevolutionary Reinforcement Learning for Generalized Control of Simulated Helicopters. *Evolutionary intelligence* 4(4):219–241.

Lample, G., and Chaplot, D. S. 2017. Playing FPS Games with Deep Reinforcement Learning. In *AAAI*, 2140–2146.

Levine, S.; Finn, C.; Darrell, T.; and Abbeel, P. 2016. End-to-end Training of Deep Visuomotor Policies. *Journal of Machine Learning Research* 17(39):1–40.

Lillicrap, T. P.; Hunt, J. J.; Pritzel, A.; Heess, N.; Erez, T.; Tassa, Y.; Silver, D.; and Wierstra, D. 2015. Continuous Control with Deep Reinforcement Learning. *arXiv preprint arXiv:1509.02971*.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; et al. 2015. Human-level Control through Deep Reinforcement Learning. *Nature* 518(7540):529–533.

Mnih, V.; Badia, A. P.; Mirza, M.; Graves, A.; Lillicrap, T.; Harley, T.; Silver, D.; and Kavukcuoglu, K. 2016. Asynchronous Methods for Deep Reinforcement Learning. In *International Conference on Machine Learning*, 1928–1937.

Ogino, M.; Katoh, Y.; Aono, M.; Asada, M.; and Hosoda, K. 2004. Reinforcement Learning of Humanoid Rhythmic Walking Parameters Based on Visual Information. *Advanced Robotics* 18(7):677–697.

Peng, X. B.; Berseth, G.; Yin, K.; and van de Panne, M. 2017. Deeploco: Dynamic Locomotion Skills Using Hierarchical Deep Reinforcement Learning. *ACM Transactions on Graphics (Proc. SIGGRAPH 2017)* 36(4).

Schulman, J.; Levine, S.; Abbeel, P.; Jordan, M.; and Moritz, P. 2015a. Trust Region Policy Optimization. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, 1889–1897.

Schulman, J.; Moritz, P.; Levine, S.; Jordan, M.; and Abbeel, P. 2015b. High-dimensional Continuous Control Using Generalized Advantage Estimation. *arXiv preprint arXiv:1506.02438*.

Shibuya, T., and Yasunobu, S. 2014. Reinforcement Learning for Environment with Cyclic Reward Depending on the Time. *IEEJ Transactions on Electronics Information and Systems C* 134(9):1325–1332.

Silver, D.; Lever, G.; Heess, N.; Degris, T.; Wierstra, D.; and Riedmiller, M. 2014. Deterministic Policy Gradient Algorithms. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, 387–395.

Todorov, E.; Erez, T.; and Tassa, Y. 2012. MuJoCo: A Physics Engine for Model-based Control. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, 5026–5033. IEEE.

Van Hasselt, H.; Guez, A.; and Silver, D. 2016. Deep Reinforcement Learning with Double Q-Learning. In *AAAI*, 2094–2100.

Wang, Z.; Schaul, T.; Hessel, M.; Van Hasselt, H.; Lanctot, M.; and De Freitas, N. 2016. Dueling Network Architectures for Deep Reinforcement Learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, 1995–2003. JMLR.org.

Watkins, C. J., and Dayan, P. 1992. Q-Learning. *Machine learning* 8(3-4):279–292.