Action Permissibility in Deep Reinforcement Learning and Application to Autonomous Driving

Sahisnu Mazumder
Department of Computer Science, University of Illinois at Chicago
IL, USA
sahisnumazumder@gmail.com

Bing Liu
Department of Computer Science, University of Illinois at Chicago
IL, USA
liub@cs.uic.edu

Shuai Wang
Department of Computer Science, University of Illinois at Chicago
IL, USA
shuaiwanghk@gmail.com

Yingxuan Zhu
FutureWei Technologies, 1 Broadway, Cambridge
MA, USA
yinxuan.zhu@huawei.com

Lifeng Liu
FutureWei Technologies, 1 Broadway, Cambridge
MA, USA
lfeng.liul@huawei.com

Jian Li
FutureWei Technologies, 1 Broadway, Cambridge
MA, USA
jian.lll@huawei.com

ABSTRACT
This paper is concerned with deep reinforcement learning (deep RL) in continuous state and action space. It proposes a new method that can drastically speed up RL training for problems that have the property of state-action permissibility (SAP). This property says that after an action \( a_t \) is performed in a state \( s_t \) and the agent reaches the new state \( s_{t+1} \), the agent can decide whether the action \( a_t \) is permissible or not permissible in state \( s_t \). An action is not permissible in a state if the action can never lead to an optimal solution and thus should not have been tried. We incorporate the proposed method into a state-of-the-art deep RL algorithm to guide its training and apply it to solve the lane keeping (steering control) problem in self-driving or autonomous driving. It is shown that the proposed method can help speedup RL training markedly for the lane keeping task as compared to the RL algorithm without exploiting the SAP-based guidance and other baselines that employ constrained action space exploration strategies.

KEYWORDS
Deep reinforcement learning, exploration-exploitation trade-off, state action permissibility, lane keeping, autonomous driving

1 INTRODUCTION
Reinforcement learning (RL) involves agents learning in their environments based on feedback from the environments [21, 41]. Most existing RL algorithms are generic algorithms. They can be applied to any application problem that can be modeled as a RL problem. However, these algorithms often take a long time to train [4, 21]. But in many real-life applications, some properties of the problems can be exploited to drastically reduce the RL training time. In this paper, we identify such a property for continuous state and action space RL problems. We call it state-action permissibility (SAP). We show that this property can be leveraged to speed up RL training markedly.

The SAP property says that after an action \( a_t \) is performed in a state \( s_t \) and the agent reaches the new state \( s_{t+1} \), the agent can decide whether the action \( a_t \) is permissible or not permissible in state \( s_t \). An action is not permissible in a state if the action can never lead to an optimal solution and thus should not have been tried. An action is permissible if it is not known to be non-permissible (i.e., the action can still be non-permissible but it is not known at the time). As a non-permissible action cannot be a part of optimal solution, the agent should avoid choosing the action in similar situations (states) in its future trails and focus more on choosing actions from permissible action space (actions that are not non-permissible). This property is quite intuitive because we humans often encounter situations where we regret an action that we have taken and based on that acquired knowledge, we avoid choosing the same action again in an "identical" or "similar" situation in the future. Thus, the knowledge of state-action permissibility gained in the past trials can help guide exploration for future trials, cutting down the exploration space drastically and gaining speed-up in RL training.

Let us consider the example of the lane keeping task in autonomous driving (see Figure 1) to illustrate the property and also the proposed technique. In this example, the car needs to learn appropriate steering control action so that it can learn to drive within a lane (often called lane keeping). A and B are the lane separation
We use the term "track" and "lane" interchangeably in the rest of this paper. The ideal trajectory for the car to drive on is the center line. We assume that at a particular time step \( t \) the car is in state \( s_t \) (see Figure 1). It takes an action \( a_t \), i.e., it turns the steering wheel counterclockwise for a certain degree. This action leads the car to the new state \( s_{t+1} \). As we can see, \( s_{t+1} \) is a worse state than \( s_t \). Assuming that the agent gets maximum reward while driving following the center line with increasing speed, it is quite clear that for the concerned lane keeping problem, the chosen action \( a_t \) is a non-permissible one, a bad action in state \( s_t \) that would never lead to an optimal solution and should not have been taken. So, when facing a similar turn of the track in the future, the agent should discard choosing \( a_t \) with high probability to reduce the possibility of making repetitive mistakes in future trials.

The SAP property can be leveraged to achieve the aforementioned goal of drastically reducing the action exploration space for RL speedup. Following the above example, we know that \( a_t \) in state \( s_t \) is not permissible as it moved the car away further from center line. However, knowing this fact only after the action has been taken is not very useful. It will be much more useful if the information like this can be used to help predict permissible and non-permissible actions in each new state so that a permissible action can be chosen in the state in the first place. This is the goal of the proposed technique.

We propose to make use of previous states, their actions, and the permissibility information of the actions to build a predictive model, which is a binary classification problem. The two classes are permissible and non-permissible. Given the current state and a candidate action in the state, the goal of the model is to predict whether the action is permissible or not in the state. We will discuss how to make use of this predictor to guide the RL training in Section 4.3. A major advantage of the proposed predictive model is that it is trained concurrently with the RL model. It does not require any human labeling of training data (permissible and non-permissible examples), which are obtained automatically during RL training by defining an Action Permissibility (AP) function and exploiting the SAP property (discussed in Section 4.1). As the agent experiences more states and actions during RL training and gathers knowledge of action permissibility, the predictive model becomes more accurate (stabilizes after some time), which in turn provides more accurate guidance to the RL training and makes it more efficient.

One question that one may ask is why the predictor can predict permissible and non-permissible actions in a future new state that may be different from any previous state (which is very likely in a continuous space). The reason is two-fold. First, in a continuous action space, if a non-permissible action is identified, it is often associated with a non-permissible action region rather than just that single action point. In our example in Figure 1, any action of turning the steering wheel further counterclockwise in state \( s_t \) would have ended up in an even worse state than the current \( s_{t+1} \) (also worse in the long run), and thus not permissible. However, the RL agent does not need to know that. After many initial RL steps, those permissible and non-permissible actions (used for training the predictor) can naturally cover those regions to a large extent. Second, it is often the case that in a continuous space, similar states have similar permissible and non-permissible action regions. The trained predictive model can capture them.

Two further questions are: (1) how to decide permissibility of an action, and (2) what happens if the predictive model predicts wrongly (as the model is unlikely to 100% accurate)? For (1), the answer is that it is domain dependent. Our approach allows the user to provide a function (the Action Permissibility (AP) function, as referred before) to make the decision. For (2), there are two cases. First, if an non-permissible action is predicted as permissible, it does not cause any problem. If a non-permissible action is chosen for a state, it just results in some waste of time. After the action is performed, the agent will detect that the action is non-permissible and it will be added to the training data together with its state information for the predictive model to improve upon in the next iteration. Second, if a permissible action is predicted as non-permissible, this is a problem because in the worst case (although unlikely), the RL may not find a solution. We solve this problem in Section 4.3.

The proposed approach can work with many RL models. In this work, we implement it to work with the Deep Deterministic Policy Gradient (DDPG) model in [25], which is a model-free RL method based on the actor-critic DPG algorithm in [38] and chose the lane keeping task for pilot study. It was also shown that DDPG performs better for the lane keeping task of autonomous driving compared to Deep Q-learning [29] and other variants [36].

In summary, this paper makes the following contributions:

1. It identifies a special property SAP in a class of continuous state and action space RL problems that can be leveraged to cut the exploration space to markedly improve the RL training efficiency. To our knowledge, the property has not been reported before.

2. It proposes a novel approach to using the SAP property, i.e., building a binary predictive model to predict whether an action in a state is permissible or not permissible. The predictive model is trained continuously along with the training of the RL model and it requires no manual labeling of training data.

3. Experimental results in our autonomous driving application show that the proposed approach can result in a huge speedup in RL training.

## 2 RELATED WORK

Reinforcement learning (RL) [41] has been studied for the past few decades [3, 39, 43]. The recent advent in deep reinforcement learning (DRL) [25, 28–30] have shown great success due to their powerful function approximation and representation learning capabilities that are important for continuous state and action space RLs that most traditional RL methods lack [4, 40]. Several variants of popular DRL methods have been proposed in recent years like Double Q-learning [44], asynchronous DRL [28] to increase the stability of policy learning, and Deep Recurrent Q-Network [19] for solving problems with partially observed environments. Our work belongs to this category of DRL methods, but we focus on leveraging a special property of the problem for DRL speed up.

Exploration-exploitation trade-off [21, 41] has been a persistent problem that makes RL slow. Although DRL methods’ performances are impressive in solving continuous control problems, they still rely on a large number of episodic trails to learn a stable policy. For the past several years, researchers have investigated on how to make RL more efficient. [23] proposed a form of policy gradient RL to
automatically search the set of possible parameters with the goal of finding the fastest possible quadrupedal locomotion. [17] formulated a RL problem in supervised learning setting where the optimal policy is obtained through a multi-class classifier. [32] proposed an approach based on the use of multiple models (or estimates) to enhance speed of convergence in learning. [14] proposed a method to speed up RL for spoken dialogue systems by combining a coarse grained abstract representation of states and actions with learning only in frequently visited states. Among other notable recent works, [16] proposed RL$_2$ to quickly learn new tasks in a few trials by encoding it in a recurrent neural network that learns through a general-purpose ("slow") RL algorithm. [46] proposed a method to adaptively balance the exploration-exploitation trade-off for opportunistic bandits and [31] tried to overcome the exploration problem in the actor-critic model DDPG [25] by providing demonstrations. [13] proposes a general policy-search framework for data-efficient learning from scratch. [6] focuses on learning the internal policies and the termination conditions of options, and the work of [5] focuses on potential-based shaping functions and its use in model-based learning algorithms. Although these works contribute in RL speed up, their problem set is the slow RL algorithm. [46] proposed a method to adaptively balance the exploration-exploitation trade-off for opportunistic bandits and [31] tried to overcome the exploration problem in the actor-critic model DDPG [25] by providing demonstrations. [13] proposes a general policy-search framework for data-efficient learning from scratch. [6] focuses on learning the internal policies and the termination conditions of options, and the work of [5] focuses on potential-based shaping functions and its use in model-based learning algorithms. Although these works contribute in RL speed up, their problem set differs significantly from ours.

The recent work in [1] focused on leveraging the knowledge of action priors which are provided by a human expert or learned through experiences from related problems in an OO-MDP (Object-oriented MDP) setup. In contrast, our work aims to learn the state-action permmissibility from the same problem in a standard MDP. Also, [1] does not introduce the concept of SAP or AP function. Several researchers also proposed some other general methods and specific techniques for detecting symmetry and state equivalence to speed up RL as well [7, 8, 18, 27, 34]. Compared to these works, we focus on constraining the exploration space by learning a guided exploration strategy by leveraging a special property of the underlying task.

Transfer learning [33, 42] and lifelong learning [2, 10] have also been used to speed up RL training. However, these methods involve multiple tasks, but our method works on one task.

In autonomous driving, vision-based learning approaches [9, 11, 15, 20, 22], imitation learning [12, 24, 47] and deep RL (DRL) methods [35, 37] have been proposed. Our work falls into the DRL category. However, compared to the exiting DRL methods for learning the lane keeping task (driving within a lane) as discussed in [35, 36], our work defines a special property of the problem to make the learning of the lane keeping task much faster.

3 BACKGROUND

Reinforcement Learning (RL). We consider the standard Markov Decision Process (MDP) setting of RL, where an agent interacts with a fully observable environment $E$ having the state space $S$ and action space $A$. Both $S$ and $A$ can be either discrete or continuous. In our work, we deal with continuous state and 1-D continuous valued action space. Given this setting, at any time step $t$ while in state $s_t$, the agent takes an action $a_t$, receives a scalar immediate reward $r_t$ and reaches a new state $s_{t+1}$. This one state transition of RL agent constitutes an experience of the agent, given by the tuple $(s_t, a_t, r_t, s_{t+1})$. The return from a state is defined as the sum of discounted future rewards, $R_t = \sum_{i=0}^{T} \gamma^{i} r_t$, where $T$ is the horizon the agent optimizes over and $\gamma \in (0, 1]$ is the discount factor. Given these experiences over a period in time, the goal of the RL agent is to learn an optimal policy $\pi^*$ that maximizes the expected return from the start distribution, i.e., $J = \mathbb{E}_{(s_t, a_t, r_t, s_{t+1}, ...)}[R_t]$. A policy $\pi$ is a mapping from states $S$ to a probability distribution over the actions $\pi : S \rightarrow \mathcal{P}(A)$ and can be either stochastic or deterministic.

In our work, we deal with deterministic policy.

Many reinforcement learning algorithms have been proposed in past several decades to solve the mentioned RL problem. The action-value function $Q^\pi(s, a)$ is a popular choice of most algorithms for estimating the optimal policy. $Q^\pi(s_t, a_t)$ describes the expected return after taking an action $a_t$ in state $s_t$ and thereafter following policy $\pi$: $Q^\pi(s_t, a_t) = \mathbb{E}_{(s_{t+1}, r_{t+1}) \sim \pi}[R_{t+1}|s_t, a_t]$ and can be written recursively using Bellman equation. When the target policy is deterministic, denoted as $\mu : S \rightarrow \mathcal{A}$ (as in our case), the Bellman equation takes the following form:

$$Q^\theta(s_t, a_t) = \mathbb{E}_{r_{t+1}, s_{t+1} \sim \mathcal{P}(s_t, a_t)}[r_{t+1} + \gamma Q^\theta(s_{t+1}, \mu(a_{t+1}))]$$

Q-learning [45] is a commonly used algorithm that employs the greedy policy $\mu(s) = \arg \max_a Q(s, a)$. For continuous state space, Q-learning is performed with function approximators parameterized by $\theta^Q$, optimized by minimizing the mean square loss:

$$L(\theta^Q) = \mathbb{E}_{s_{t-1}, a_{t-1}, r_{t-1} \sim \rho_{t-1}, s_t \sim E(s_t, a_t)}[Q(s_t, a_t | \theta^Q) - y_t]^2$$

where, $y_t = r(s_t, a_t) + \gamma Q(s_{t+1}, \mu(a_{t+1}))|\theta^Q |$ and $\rho^t$ is the discounted state transition distribution for policy $\beta$. The dependency of $y_t$ on $\theta^Q$ is typically ignored.

For the past several decades, mostly linear function approximators were used to deal with continuous state and action spaces. Non-linear function approximators were considered to be unstable. Recently, [29, 30] adapted the Q-learning algorithm and made effective use of large neural networks as non-linear function approximators by stabilizing learning using of a replay buffer, and a separate target network for calculating $y_t$.

Deep Deterministic Policy Gradient (DDPG). Applying a Q-learning algorithm straightforwardly to RL problems with continuous action spaces is hard as the optimization gets slow while finding the greedy policy for large, unconstrained function approximators and nontrivial action spaces. To solve this problem, [25] proposed an off-policy, model-free DRL method called Deep Deterministic Policy Gradient (DDPG) based on the actor-critic based DPG algorithm [38]. In our work, we incorporate an action permmissibility (AP) predictor into DDPG to drastically improve its efficiency by leveraging the SAP property and concept of AP function (discussed in Section 4).

DDPG learns an action-value function (Critic) by minimizing the Bellman error, while simultaneously learning a policy (Actor) by directly maximizing the estimated action-value function with respect to the parameters of the policy. More precisely, DDPG maintains an Actor function $\mu(s)$ with parameters $\theta^\mu$, a Critic function $Q(s, a)$ with parameters $\theta^Q$, and a replay buffer $R$ as a set of experience tuples $(s_t, a_t, r_t, s_{t+1})$ to store transition history for training. DDPG alternates between running the policy to collect experiences and updating the parameters. Training rollouts are collected with extra noise for exploration: $a_t = \mu(s) + \mathcal{N}_t$, where $\mathcal{N}_t$ is a noise process. In each training step, DDPG samples a minibatch consisting of $N$ tuples from $R$ to update the Actor and Critic networks and minimizes...
the following loss w.r.t. \( \theta^Q \) (Critic parameters) to update the Critic:

\[
L(\theta^Q) = \frac{1}{N} \sum_{i} [y_i - Q(s_i, a_i|\theta^Q)^2]
\]

where, \( y_i = r_i + \gamma Q(s_{i+1}, \mu(s_{i+1})|\theta^Q) \).

The Actor parameters \( \theta^\mu \) are updated using the sampled policy gradient:

\[
\nabla_{\theta^\mu} J = \frac{1}{N} \sum_{i} \nabla_a \left( Q(s_i, a_i|\theta^Q) \right)_{a=a(s_i)} \nabla_{\theta^\mu} \mu(s_i)\right)
\]

To stabilize learning, the \( Q \) value in equation 3 is usually computed using a separate network (called the target network) whose weights are an exponential average over time of the Critic network, which gives smoother target values.

4 THE PROPOSED TECHNIQUE

We now present our proposed framework, which consists of the state-action permissibility (SAP) property, the permissible and non-permissible action predictive model, and the integration of the predictive model in the RL training to guide the RL model. We detail them in turn.

4.1 State-Action Permissibility

Let \( r : (S, \mathcal{A}) \rightarrow \mathbb{R} \) be the reward function for a given MDP with state space \( S \) and action space \( \mathcal{A} \). In this work, we assume that the action space is one-dimensional (which can be expressed by one variable) \(^1\).

**Definition 1** (strictly non-permissible): Let the RL agent be in state \( s_t \) at a time step \( t \). If a possible action \( a_t \) in state \( s_t \) cannot lead to an optimal solution, the action is said to be a *strictly non-permissible* action in state \( s_t \). We denote the set of all strictly non-permissible actions in \( s_t \) by \( snp_t \).

**Definition 2** (strictly permissible): Let the RL agent be in state \( s_t \) at a time step \( t \). If a possible action \( a_t \) in state \( s_t \) is not strictly non-permissible in state \( s_t \), the action is said to be *strictly permissible* in state \( s_t \). We denote the set of all strictly permissible actions in state \( s_t \) by \( sp_t \).

**Definition 3** (state-action permissibility): Let a state transition in a RL problem be \((s_t, a_t, r_t, s_{t+1})\). We say that the RL problem has the SAP property if there is an action permissibility (AP) function \( f : (S, \mathcal{A}) \rightarrow \{0, 1\} \) that determines whether the action \( a_t \) in state \( s_t \) is permissible \( f(s_t, a_t|s_{t+1}) = 1 \) or non-permissible \( f(s_t, a_t|s_{t+1}) = 0 \) in state \( s_{t+1} \) after the action \( a_t \) has been performed such that

1. if \( a_t \) is determined to be non-permissible in state \( s_t \), it must be strictly non-permissible in \( s_t \), i.e., \( a_t \in snp_t \);
2. if \( a_t \) is determined to be permissible, it is not non-permissible and it can be strictly permissible or strictly permissible.

Definition 3 basically says that our action permissibility (AP) function \( f \) (which is dependent on the application and is provided by the user) may miss some strictly permissible actions, but if an action is determined to be non-permissible, it must be strictly non-permissible. That is, the precision for non-permissible actions must be perfect but the recall may not be. \( f \) may not be unique for a problem. The best \( f \) function is the one such that if \( a_t \) is a strictly non-permissible action or a strictly permissible action in state \( s_t \), it

\(^1\)We leave the multi-dimensional action space case to our future work.

\(^2\)https://yanpanlau.github.io/2016/10/11/Torcs-Keras.html

![Figure 2: Visualizing the parameters of lane keeping task immediate reward function.](Image 362x557 to 512x713)
thus the aforementioned AP function satisfies the SAP property for the concerned lane keeping problem.

Note that, the SAP property is dependent on the state, i.e., a non-permissible action in one state \( s_t \) may become permissible in some other state. The AP function determines the permissibility of an executed action in a given state, not globally and thus, the SAP property is local to each given state.

4.2 Learning Action Permissibility (AP) Predictor

AP function only gives knowledge about the permissibility of an “executed” action (not for a chosen action yet to be executed). It cannot predict the next state because the system does not know how much to turn will result in a permissible action. Also, as our problem is model-free, we can only have the knowledge of action permissibility when the state transition occurs. For example, assuming zero prior knowledge in driving, we get to know about the outcome of an unexplored action (taking a random turn), only when the state of the car changes due to that action. And we can foresee the implication of a chosen action, when we encounter a state similar to the one explored before. Thus, we need to continuously learn the permissibility of actions for a given state utilizing our past experiences and decide about future action exploration for unseen states. The AP function provides the ground truth (corresponding to the explored state-action pairs) for the AP prediction problem which can be leveraged to train an action premissibility (AP) prediction or classification model, called AP predictor which can foresee the permissibility of a chosen action for the current state and guide the exploration process.

As indicated earlier, we formulate AP prediction as a binary classification problem with two classes, permissible and non-permissible. Given the current state \( s \) and an action \( a \), the goal of the AP predictor is to predict whether \( a \) is permissible or not in \( s \). Note that AP predictor is a learned predictive model (or classifier), which is different from the AP function (a user provided function) and both have their distinct objectives. The labeled training data is produced by the AP function \( f \), which determines whether an action at a particular state was permissible or non-permissible during the RL training. Each example of the training data consists of values of all variables representing a state and the action taken in the state with its class (permissible or non-permissible). After many initial steps of RL, a set of training examples for building the AP predictor is collected.

Since the training of the AP predictor is performed continuously along with the RL training, to manage the process and the stream of new training examples, we maintain a training data buffer \( K \) similar to the replay buffer \( R \) in [25, 29] to train the AP predictor. Given a RL experience tuple \( (s_t, a_t, r_t, s_{t+1}) \) at time step \( t \), we extract the tuple \( (s_t, a_t, l(a_t)) \) and store it in \( K \). Here, \( l(a_t) \) is the class label for \( a_t \) in \( s_t \), permissible (+ve class) or non-permissible (-ve class) and is inferred using the AP function \( f \) as discussed above. Similar to the replay buffer, \( K \) is finite in size and when it gets full, newer tuples replace oldest ones having the same class label \( l(a_t) \).

We train AP predictor \( E \) with a balanced dataset at a time step \( t \) as follows. For time step \( t \), if both the number of +ve as well as -ve tuples (or examples) in \( K \) are at least \( N_E/2 \) (ensures \( N_E/2 \) +ve and \( N_E/2 \) -ve examples can be sampled from \( K \)), we sample a balanced dataset \( D_E \) of size \( N_E \) from \( K \) using stratified sampling. Then, we train the neural network AP predictor \( E \) with parameter \( \theta^E \) using \( D_E \).

Note that, AP predictor is just a general supervised learning model. Since we deal with real-valued sensor information as state and 1D action space, we use a feedforward network for our AP predictor design. For training \( E \), we use mini-batch gradient decent to update \( \theta^E \) and minimize the L2-regularized binary cross-entropy loss:

\[
L(\theta^E) = -\frac{1}{N_E} \sum_{(s_i, a_i, l(a_i)) \in D_E} \left[ l(a_i) \log E(s_i, a_i|\theta^E) + (1 - l(a_i)) \log (1 - E(s_i, a_i|\theta^E)) \right] + \frac{\lambda}{2} \sum_{\theta^E} \| \theta^E \|^2_2
\]

where \( \lambda \) is the regularization parameter. We discuss the use of the AP predictor in a RL model in the next subsection.

One may argue that, instead of learning an AP predictor utilizing the AP function, what if we can define directly constraints to guide the exploration process? Defining such constraints is harder (specially considering continuous state and action space) and often leads to sub-optimal solution. In fact, our baselines use strong domain knowledge to constrain actions to be taken at a state. But, our experiment results show, they are inferior to our approach.

4.3 Guiding RL Model with AP Predictor

The proposed AP predictor can work with various RL models. In this work, we use the deep actor-critic model Deep Deterministic Policy Gradient (DDPG) [25], which we have introduced in Section 3. We chose DDPG because it is a state-of-the-art for learning continuous control tasks and in [35], it was also shown that DDPG performs better for our concerned lane keeping task compared to Deep Q-learning and other variants [35]. Our integrated algorithm of DDPG and the AP predictor \( E \) is called DDPG-AP. The training process of Actor \( \mu \) and Critic \( Q \) of DDPG-AP is identical to the DDPG algorithm described in Section 3. We will not repeat it here. The training process of the AP predictor \( E \) of DDPG-AP has been discussed in Section 4.2, which is performed simultaneously with the training of Actor \( \mu \) and Critic \( Q \). In the following paragraph, we discuss how DDPG-AP uses the AP predictor \( E \) (trained up to time step \( t \)) and the Actor network \( \mu \) for action selection. Note that in the original DDPG, the action for each state is only generated by the Actor network.

Algorithm 1 presents the action selection process of DDPG-AP. Given the trained AP predictor \( E \) at time step \( t \), the action selection process for \( s_t \) works as follows: Initially, when the agent starts learning the optimal policy, the tuples stored in \( K \) are few in number and thus, are not reliable enough to build a good AP predictor. Moreover, \( E \) also needs a diverse set of training examples (tuples) for learning a generalized AP prediction or classification model. Thus, for the initial few time steps (say, \( t \leq t_0 \)), the Actor chooses action \( a_t \) using the noise based exploration (lines 2-4) which helps in populating \( K \) with tuples (or examples) from diverse experiences obtained from the environment (see Sec 4.2). \( N_E \) is the noise (line 4) added to the action \( a_t \) estimated by the Actor network (line 1).

For time steps \( t > t_0 \) (lines 5-14), the action estimated by the Actor network \( a_t = \mu(s) \) (line 1) is fed to \( E \) for AP prediction with probability \( \alpha \). Some explanation is in order here about \( \alpha \). As we mentioned in the introduction section, since the AP predictor is hard to be 100% accurate, we need to deal with the case where a permissible action is predicted as non-permissible (false negative) (false positive is not an issue, see Section 1). This is a problem...
Algorithm 1 DDPG-AP Action Selection

**Input:** Current state $s_t$, Actor $\mu(s_t|\theta^p)$ and AP predictor $E(s_t,a|\theta^E)$, exploration noise process $N_t$, time step $t$, time step threshold $t_k$ and probability $\alpha$ for consulting $E$.

**Output:** $a_t$: Action selected for $s_t$

1: Select action $a_t = \mu(s_t|\theta^p)$ for $s_t$
2: if $t < t_k$ then
3: $a_t = a_t + N_t$
4: end if
5: if $t > t_k$ and $\text{Unif}(0, 1) < \alpha$ then
6: $l(a_t) \leftarrow E(s_t, a_t|\theta^E)$
7: if $l(a_t)$ is -ve (non-permissible) then 8: Sample $\mathcal{A}_{s_t}$ from $\mathcal{A}$ using low-variance uniform sampling and build $D_{s_t}$ as $\{ (s_t, a) \mid a \in \mathcal{A}_{s_t} \}$
9: $\mathcal{A}P(s_t) = \{ a \mid E(s_t, a) = +ve, (s_t, a) \in D_{s_t} \}$
10: if $\mathcal{A}P(s_t) \neq \emptyset$ then
11: Randomly sample $a_t$ from $\mathcal{A}P(s_t)$
12: end if
13: end if
14: end if
15: Return $a_t$

because in the worst case (although unlikely), RL may not find a solution. We deal with the problem as follows: The Actor listens to $E$ for $\alpha \%$ of the time. This probability ensures that Actor executes the action generated by itself (including some false negatives) on the environment (not in Algorithm 1) and returns action $a_t$. Such a sampling procedure ensures that the actions are sampled uniformly from the full action space $\mathcal{A}$ using low-variance uniform sampling and finds a permissible action for the current state $s_t$ as follows (line 8). First, $\mathcal{A}$ is split into $N$ equal sized intervals and an action is sampled from each interval following uniform distribution to produce a set of sampled actions for state $s_t$, denoted by $\mathcal{A}_{s_t}$. Such a sampling procedure ensures that the actions are sampled uniformly over $\mathcal{A}$ with variances between consecutive samples being low. Thus, any action in $\mathcal{A}$ will be equally likely to be selected, provided $\mathcal{A}$ is it is predicted to be permissible (+ve) by $E$ (line 9). Once $\mathcal{A}_{s_t}$ is sampled, Actor forms a dataset $D_{s_t}$ by pairing each $a \in \mathcal{A}_{s_t}$ and fed $D_{s_t}$ to $E$ in a single batch to estimate a permissible action space for $s_t$ as $\mathcal{A}P(s_t)$ (line 9). Here +ve means permissible. Next, Actor randomly samples an action $a_t$ from $\mathcal{A}P(s_t)$ (line 11) and returns it (line 15). The Actor will then executes it on environment (not in Algorithm 1). If $\mathcal{A}P(s_t) = \emptyset$, the original $a_t$ (line 1) is returned and Actor executes it on the environment.

In Algorithm 1, the values of the hyper-parameters $t_k$ and $\alpha$ is chosen empirically (reported in Section 5.1). Note that, the hyper-parameter $t_k$ denotes the threshold time step for using noise-based exploration. This is required for gathering a reasonable amount of diverse training data and allowing some time for training a sufficiently good AP predictor, before it’s employed in operation. So, lowering $t_k$ results in a poorly trained model, whereas a large $t_k$ introduces delay in the guided exploration process. On the other hand, $\alpha$ controls the degree to which the actor listens to the AP predictor. Smaller value of $\alpha$ means AP predictor will have less influence in the action selection (by the actor) and vice-versa. We set $\alpha$ empirically based on performance of AP predictor. For poorly performed AP predictor, $\alpha$ is set to a smaller value, so that the AP predictor does not influence actor much. When the AP predictor has learned sufficiently (evaluated based on validation accuracy), we set alpha to a higher value, so that the agent often listens to the AP predictor. In our experiments, if the validation accuracy of $E \geq 70\%$, we empirically set $\alpha = 0.9$.

Note that, instead of estimating $\mathcal{A}P(s_t)$ using sampling and binary classification, a reader may argue that we can model the problem as a regression problem and directly learn to predict the lower and upper bounds of $\mathcal{A}P(s_t)$. However, such formulation introduces noise in the training of $E$ as we cannot estimate the true bounds of $\mathcal{A}P(s_t)$ from past experiences. Also, in such cases, as the agent gathers more and more experiences over time, the newer experiences will be closer to the true estimates of the bounds and older ones will become more noisy and thus, affects the learning.

### Table 1: TORCS state and action variables along with their descriptions [26] used in our experiments.

<table>
<thead>
<tr>
<th>Name</th>
<th>Range (unit)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>angle</td>
<td>[-\infty, \infty] (deg)</td>
<td>Angle between the car direction and the direction of the track center axis.</td>
</tr>
<tr>
<td>trackPos</td>
<td>(-\infty, \infty)</td>
<td>Distance between the car and the track center axis.</td>
</tr>
<tr>
<td>speedX</td>
<td>(-\infty, \infty)(km/h)</td>
<td>Speed of the car along the longitudinal axis of the car.</td>
</tr>
<tr>
<td>speedZ</td>
<td>(-\infty, \infty)(km/h)</td>
<td>Speed of the car along the Z-axis of the car.</td>
</tr>
<tr>
<td>Action</td>
<td>[-1, 1]</td>
<td>Steering value: -1 and +1 means respectively full right and left, that corresponds to an angle of 0.366519 rad.</td>
</tr>
</tbody>
</table>

5 EXPERIMENTAL EVALUATION

We evaluate our proposed DDPG-AP framework in the application of lane keeping (steering control) task in terms of learning performance and accuracy of the learned policy and compare it with baselines. We use an open-source, standard autonomous driving simulator TORCS [26] following [35, 36] for both learning and evaluation purposes. Note that like other research on autonomous driving using RL [35, 36], we focus on the lane keeping task as we want to show the effect of the proposed method on RL training. We do not use the task of acceleration and brake control because they are more subjective involving human comfort and other preferences.

5.1 Experimental Setup

The Open Racing Car Simulator (TORCS) provides us with graphics and physics engines for Simulated Car Racing (SCR). The availability of the diverse set of road tracks with varying curvatures and slopes in TORCS makes it an appropriate choice for model evaluation in different driving scenarios. It also allows us to play with different car control parameters like steering angle, velocity, acceleration, brakes, etc. TORCS details can be found in [26].

For our experiment, we used five sensor readings to represent the state vector which we found are sufficient for learning good policy
in diverse driving situations. The goal of our experiment is to assess the lane keeping task, i.e., how well the driving agent has learned to drive while positioning itself on the track/lane axis. Thus, our model and baselines focus on predicting the right steering angle that can keep the car aligned to the track axis while driving with a default speed. During training, whenever the car goes out of the track, we terminate the current episode and restarts the TORCS environment to initiate a new one. We present a summary of the state and action variables (used in our experiments) in Table 1. Figure 3 shows 5 road tracks used in our experiments. Among these 5 road tracks, we used the wheel-2 track for our training and rest of the four tracks for testing. Due to various curvature variations, we consider wheel-2 as ideal for training all possible scenarios.

**Parameter Settings.** For our task, we build a two layer (empirically chosen) feed-forward neural network of 128 and 256 hidden units respectively to learn the AP predictor, Actor and Critic networks. Other important empirically chosen parameters are: learning rates for Actor as 0.0001, Critic as 0.001 and for AP predictor as 0.01 and regularization parameter $\lambda$ as 0.01. Discount factor for Critic updates as 0.9, target network update parameter 0.001, $\alpha$ as 0.9, replay buffer size 100k, knowledge buffer size 10k (stores tuples in 9:1 ratio as training and validation examples), batch size for minibatch-SGD 128, sample size as 128 used for AP-based exploration, $t_k$ as 500 for 15k training and 150 for 3k training experiments, sample size for building dataset for training AP predictor at each step is 2k and validation sample dataset size is 200.

**Baselines.** Our main goal is to compare our proposed DDPG-AP model with the original DDPG model without using any additional action selection guidance. Apart from that, we also proposed some variations of our DDPG-AP algorithm based on some characteristics of driving, which also perform constrained exploration, as discussed below to show the importance of learning the AP predictor compared to constrained exploration strategies.

**DDPG-CE1.** DDPG-CE1 is an extension of DDPG algorithm that applies the following two constraints for action space exploration: (1) If the car is on the left of track axis/lane center line and the current action $a_t > a_{t-1}$ (previous action), instead of applying $a_t$ on environment, it samples actions uniformly from (-1.0, $a_{t-1}$) [see Table 1]. This constraint says that, when the car is on left of the center line, it should avoid taking any left turn further from its current position. Similarly, (2) if the car is on the right of the lane center line and $a_t < a_{t-1}$, then sample actions from ($a_{t-1}$, 1.0) and thus, it should avoid turning right further. Otherwise, the car executes $a_t$. In this case, the AP predictor is not used for any guidance.
**Table 2: Performance of DDPG, DDPG-AP and its variants on different test tracks.**

<table>
<thead>
<tr>
<th>Test track</th>
<th>DDPG</th>
<th>DDPG-CE1</th>
<th>DDPG-CE2</th>
<th>DDPG-AP</th>
<th>DDPG-(AP+CE2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training track: Wheel-2 [After 3k training steps]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E-road</td>
<td>Y (17.59%)</td>
<td>4460.96</td>
<td>2335.24</td>
<td>Y 53371.60</td>
<td>Y 49174.27</td>
</tr>
<tr>
<td>Spring</td>
<td>N (4.30%)</td>
<td>8258.29</td>
<td>166150.64</td>
<td>N 162875.81</td>
<td>Y 339468.72</td>
</tr>
<tr>
<td>CG Track 3</td>
<td>N (5.75%)</td>
<td>1574.31</td>
<td>19586.10</td>
<td>Y 46948.97</td>
<td>Y 42834.71</td>
</tr>
<tr>
<td>Oleth Ross Road</td>
<td>N (16.82%)</td>
<td>8784.68</td>
<td>45586.42</td>
<td>Y 105419.94</td>
<td>Y 95606.01</td>
</tr>
<tr>
<td>Training track: Wheel-2 [After 15k training steps]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E-road</td>
<td>Y</td>
<td>40785.05</td>
<td>51273.21</td>
<td>Y 53653.04</td>
<td>Y 55147.79</td>
</tr>
<tr>
<td>Spring</td>
<td>N (36.68%)</td>
<td>117519.69</td>
<td>N (35.78%)</td>
<td>125557.99</td>
<td>Y 368559.39</td>
</tr>
<tr>
<td>CG Track 3</td>
<td>Y</td>
<td>37011.54</td>
<td>44657.91</td>
<td>Y 46975.31</td>
<td>Y 48614.74</td>
</tr>
<tr>
<td>Oleth Ross Road</td>
<td>Y</td>
<td>86275.80</td>
<td>100232.72</td>
<td>Y 105584.83</td>
<td>Y 109311.58</td>
</tr>
</tbody>
</table>

**DDPG-CE2.** DDPG-CE2 improves DDPG-CE1, where we apply the constraints mentioned above, only when $\delta_{\text{track}}(t) - \delta_{\text{track}}(t-1) > 0$, i.e., only when the car moves away from the track center due to its previous action. If the car is moving closer to the track center, it is permissible. This method gives very strong constraints on car’s movement. Again, this model does not use the AP-predictor.

**DDPG-(AP+CE2).** As we will see that DDPG-CE2 is a strong baseline, we thus combine our DDPG-AP and DDPG-CE2 to give us DDPG-(AP+CE2).

### 5.2 Results and Analysis

Figure 4(a) shows the comparative result of DDPG-AP and other algorithms with regard to the average reward over the training steps. We conducted training for 15k steps and report the moving average of reward over the past 100 steps. From equation 5, it is clear that the reward function depends on velocity, distance from the track center and the car direction. The minor fluctuations in the curve shows the stability in learning, i.e., how smoothly each algorithm has learned to keep the car aligned to the track center axis. A sharp fall in the curve corresponds to a sudden end of episode, i.e., when the car goes out of track with a large -ve reward. From Figure 4(a), we see that the moving average for DDPG-AP increases very rapidly compared to other algorithms and gets stable more quickly (around 2500 steps) compared to others. This also suggests that DDPG-AP learns very quickly to keep car steadily moving in track/lane. Whereas the moving average reward for DDPG, DDPG-CE1 are quite unstable. DDPG-CE2 performs comparatively better than DDPG, DDPG-CE1 and the combined algorithm DDPG-(AP+CE2) performs better than DDPG-CE2 in terms of learning stability. But still, they are less stable than DDPG-AP.

Figure 4(b) shows the number of episode taken by each algorithm for learning the lane keeping task. If an algorithm consumes lesser number of episodes, it means the algorithm learns quicker to keep the car moving without going out of track. As we can see in Figure 4(b), DDPG took more than 120 episodes to learn to drive for a considerable amount of distance. However, the sharp falls in average reward and beginning of a new episode indicate that the learning is yet not stable. Compared to the constrained versions of DDPG, DDPG-CE1 and the more strict version DDPG-CE1 learn quicker than DDPG to drive a considerable amount of distance. Among all, DDPG-AP and DDPG-(AP+CE2) learn very quickly, and as their curves do not fall down, it indicates the car has never gone out of track after 20 (for DDPG-(AP+CE2)) and 25 (for DDPG-AP) episodes.

Figure 5 shows the performance of AP predictor’s validation accuracy over first 5k training steps. The accuracy stays always above 70% during training and stabilizes with an average of 80%, signifying that our AP predictor learns well to classify permissible actions from non-permissible ones.

Table 2 shows the performance of DDPG, DDPG-AP and other algorithms on unseen test tracks considering both 3k and 15k steps of training. We use each algorithm to drive the car for one lap of each track and report the total reward obtained by each algorithm. The “lap done?” column indicates whether the car has completed the lap or not, and if not, (%) of the total track length the car has covered from its beginning position, before it went out of track.

Considering the results for the 3k training steps (which is very few for learning a stable policy), we see that DDPG has not learned to make the car complete one lap for any test track. DDPG-CE1 learns to make the car complete the lap. However, the very low reward value shows that the car did not move along the track center very often. Both DDPG-AP and DDPG-(AP+CE2) perform much better in term of lap completion considering the other three algorithms. Although DDPG-CE2 achieves high rewards (not surprising due to strong additional constraints) except track ‘Spring’, the lap ‘Spring’ was
Action Permissibility in Deep Reinforcement Learning


not completed before it went out of track, which shows the policy learned by DDPG-CE2 is less generic compared to DDPG-AP and DDPG-(AP+CE2).

Considering the results for 15k training steps, we see that all algorithms except DDPG and DDPG-C17 have learned to keep the car on track for all test tracks. The highest total reward values and lap completion information in DDPG-AP (considering all test tracks) indicate that DDPG-AP has learned to find the most general policy compared to others. Although DDPG-CE2 achieves highest reward for 3k training steps, the rewards obtained in 15k are less than those for DDPG-AP and DDPG-(AP+CE2). This further shows that the policy learned by DDPG-CE2 is sub-optimal. Our DDPG-AP method did the best without those constraints.

6 CONCLUSION

In this paper, we proposed an novel property, called state-action permissibility (SAP), for improving the RL training efficiency in problems with this property. To leverage this property, two new components are added to a deep RL (DRL) algorithm: action permissibility function and action permissibility predictor. They help the DRL algorithm select promising actions to speed up its training. Our experiments considering the lane keeping task in autonomous driving showed that the proposed method is highly effective.

Also note that, our work deals with a class of RL problems, where it is feasible to design an explicit AP function to get feedback about the exploration. Examples of such class of problems primarily include robot navigation problems, game playing, etc. apart from autonomous driving, where the agent has to make a sequence of optimal decisions/actions to achieve its goal. For such a problem, it’s easier to specify the AP function with expert’s domain knowledge. Our work does not aim to generalize the approach to all RL problems (as it’s not feasible to specify such an AP function for all RL problems). We also do not focus on designing procedures to identify AP functions as AP functions depend on specific problems and expert knowledge. Our work assumes that such a function is given and we show that an AP predictor can be learned using the function and past exploration experiences. The AP predictor then can guide the RL training. In the future, we plan to extend our framework to multidimensional continuous action spaces and apply it to other practical applications.

ACKNOWLEDGMENTS

This work was supported in part by a grant from National Science Foundation (NSF) under grant no. IIS-1407927 and a research award from Huawei Technologies.

REFERENCES


