Constrained Policy Gradient Method for Safe and Fast Reinforcement Learning: a Neural Tangent Kernel Based Approach

Balázs Varga[1a], Balázs Kulcsár[a], Morteza Haghir Chehreghani[a]

[a]*Department of Electrical Engineering, Chalmers University of Technology, Hörsalsvägen 11, Gothenburg, Sweden*

Abstract

This paper presents a constrained policy gradient algorithm. We introduce constraints for safe learning with the following steps. First, learning is slowed down (lazy learning) so that the episodic policy change can be computed with the help of the policy gradient theorem and the neural tangent kernel. Then, this enables us the evaluation of the policy at arbitrary states too. In the same spirit, learning can be guided, ensuring safety via augmenting episode batches with states where the desired action probabilities are prescribed. Finally, exogenous discounted sum of future rewards (returns) can be computed at these specific state-action pairs such that the policy network satisfies constraints. Computing the returns is based on solving a system of linear equations (equality constraints) or a constrained quadratic program (inequality constraints). Simulation results suggest that adding constraints (external information) to the learning can improve learning in terms of speed and safety reasonably if constraints are appropriately selected. The efficiency of the constrained learning was demonstrated with a shallow and wide ReLU network in the Cartpole and Lunar Lander OpenAI gym environments. The main novelty of the paper is giving a practical use of the neural tangent kernel in reinforcement learning.

[1]balazsv@chalmers.se

1. Introduction

In reinforcement learning (RL), the agent learns in a trial and error way. In a real setting, it can lead to undesirable situations which may result in damage or injury of the agent or the environment system. In addition, the agent might waste a significant amount of time exploring irrelevant regions of the state and action spaces. Safe RL can be defined as the process of learning policies that maximize the expectation of the return in problems under safety constraints. Thus, safe exploration often includes some prior knowledge of the environment (e.g. a model) or has a risk metric [10]. In RL, safety can be guaranteed in a probabilistic way. Learning in this context aims to strike a balance between exploration and exploitation so that the system remains within a safe set. On the other hand, in a safety-critical setting, exploration cannot be done blindly.

Recently, several different approaches were conceived to tackle safe learning. [30] deals with balancing between exploitation and exploration. It includes the exploration explicitly into the objective function. [3] proposed a safe RL algorithm with stability guarantees expanding an initial safe set gradually. The proposed learning algorithm safely optimizes the policy in an asymptotically stable way. The authors exploit the fact that value functions in RL are Lyapunov functions if the costs are strictly positive away from the origin. In [36] safe active learning of time-series models is presented for Gaussian Processes. The data is sequentially selected for labeling such that information content is maximized. Then, an additional risk model is approximated by a GP. Only trajectories that are deemed safe are applied for the learning. A novel safe Bayesian optimization algorithm is introduced in [24]. It addresses the challenge of efficiently identifying the total safe region and optimizing the utility function within the safe region for Gaussian Processes. The safe optimization problem is done in two stages: an exploration phase where the safe region is iteratively expanded, followed by an optimization phase in which Bayesian optimization is performed within the safe region. [28], formulates a teacher-student safe-learning algorithm: the agent (student) learns to perform a task in an environment while the teacher intervenes if it reaches an unsafe region, affecting the student's reward. The teacher learns an intervention policy too. The teacher has only limited information on the environment and the student. Thus the adaptive teacher is an agent operating in a partially observable Markov Decision Process. [8] takes a different approach and creates an RL algorithm that can solve problems within a few episodes based on Gaussian Processes (GPs). Some works aim at connecting RL with robust control (i.e. achieving the best performance under worst-case disturbance), which can be thought of as a form of safe learning [17], [21]. Constrained learning is an intensively studied topic, having close ties to safe learning [33]. [13] argues that constrained learning has better generalization performance and a faster convergence rate. [27] presents a constrained policy optimization, which uses an alternative penalty signal to guide the policy. [29] proposes an actor-critic method with constraints that define the feasible policy space.

In this paper, we develop a deterministic policy gradient algorithm augmented with equality and inequality constraints via shaping the rewards. Compared to [27], training does not rely on non-convex optimization or additional heuristics. In policy gradient methods, the function approximator predicts action values or probabilities from which the policy can be derived directly. Finding the weights is done via optimization (gradient ascent) following the policy gradient theorem [25]. Policy gradient methods have many variants and extensions to improve their learning performance [35]. For example, in [15] a covariant gradient is defined based on the underlying structure of the policy. [7] deduced expected policy gradients for actor-critic methods. [6] deals with the reduction of the variance in Monte-Carlo (MC) policy gradient methods. In this work, we employ the REINFORCE algorithm (Monte-Carlo policy gradient) [31]. Under a small learning rate, with the help of the recently introduced Neural Tangent Kernel [14] the evolution of the policy can be described during their training by gradient descent. Earlier, policy iteration has been used in conjunction with NTK for learning the value function [11]. On the other hand, it was used in its analytical form as the covariance kernel of a GP ([22], [18], [32]). In this work, we directly use the NTK to project a one-step policy change in conjunction with the policy gradient theorem. Then, safety is incorporated via constraints. It is assumed that there are states where the agent's desired behaviour is known. At these "safe" states action probabilities are prescribed as constraints for the policy. Finally, returns are computed in such a way that the agent satisfies the constraints. This assumption is mild when considering physical systems: limits of a controlled system (saturation) or desired behavior at certain states are usually known (i.e. the environment is considered a gray-box). The proposed algorithm is developed for continuous state spaces and discrete action spaces. According to the proposed categorization in [10], the proposed algorithm falls into constrained optimization with external knowledge.

The contribution of the paper is twofold. First, we analytically give the policy evolution under gradient flow, using the NTK. Second, we extend the REINFORCE algorithm with constraints. Our version of the REINFORCE algorithm converges within a few episodes if constraints are set up correctly. The constrained extension relies on computing extra returns via convex optimization. In conclusion, the paper provides a practical use of the neural tangent kernel in reinforcement learning.

The paper is organized as follows. First, we present the episode-by-episode policy change of the REINFORCE algorithm (Section 2.1). Then, relying on the NTK, we deduce the policy change at unvisited states, see Section 2.2. Using the results in Section 2.2, we compute returns at arbitrary states in Section 3. We introduce equality constraints for the policy by computing "safe" returns by solving a system of linear equations (Section 3.1). In the same context, we can enforce inequality constraints by solving a constrained quadratic program, see Section 3.2. In Section 4.1 we investigate the proposed learning algorithm in two OpenAI gym environments: in the Cartpole environment and in the Lunar Lander (Section 4.2). Finally, we summarize the findings of this paper in Section 5.

2. Kernel-based analysis of the REINFORCE algorithm

In this section, the episode-by-episode learning dynamics of a policy network is analyzed and controlled in a constrained way. To this end, first, the REINFORCE algorithm is described. Then, the learning dynamics of a wide and shallow neural network is analytically given. Finally, returns are calculated that force the policy to obey equality and inequality constraints at specific states.

2.1. Reformulating the learning dynamics of the REINFORCE algorithm

Policy gradient methods learn by applying the policy gradient theorem. An agent's training means tuning the weights of a function approximator episode-by-episode. In most cases function approximator is a Neural Network [23], [1]. If the training is successful, then the function approximator predicts values or probabilities from which the policy can be derived. More specifically, the output of the policy gradient algorithm is a probability distribution, which is characterized by the function approximator's weights. In the reinforcement learning setup, the goal is maximizing the (discounted sum of future) rewards. Finding the optimal policy is based on updating the weights of the policy network using gradient ascent. The simplest of the policy gradient methods is the REINFORCE algorithm [31], [26]. The agent learns the policy directly by updating its weights $\theta(e)$ using Monte-Carlo episode samples. Thus, one realization is generated with the current policy (at episode e) $\pi(a_e|s_e, \theta(e))$. Assuming continuous state space, and discrete action space the episode batch (with length n_B) is $\{\underline{s}_e(k), \underline{a}_e(k), \underline{r}_e(k)\}$, for all $k = 1, 2, ..., n_B$, where $\underline{s}_e(k) \in \mathbb{R}^{n_n}$ is the n_n dimensional state vector in the k^{th} step of the MC trajectory, $\underline{a}_e(k) \in \mathbb{Z}$ is the action taken, and $\underline{r}_e(k) \in \mathbb{R}$ is the reward in step k. For convenience, the states, actions, and rewards in batch e are organized into columns $\underline{s}_e \in \mathbb{R}^{n_B \times n_n}$, $\underline{a}_e \in \mathbb{Z}^{n_B}$, $\underline{r}_e \in \mathbb{R}^{n_B}$, respectively. The REINFORCE algorithm learns as summarized in Algorithm 1. The update rule is based on the policy gradient theorem [25] and for the whole episode it can be written as the sum of gradients induced by the batch:

$$\theta(e+1) = \theta(e) + \alpha \sum_{k=1}^{n_B} \left(G_e(k) \frac{\partial}{\partial \theta} log \pi(\underline{a}_e(k)|\underline{s}_e(k), \theta(e)) \right)^T \tag{1}$$

Algorithm 1: The REINFORCE algorithm

```
Initialize $e = 1$.
Initialize the policy network with random $\theta$ weights.
while not converged do
    Generate a Monte-Carlo trajectory $\{\underline{s}_e(k), \underline{a}_e(k), \underline{r}_e(k)\}$,
     $k = 1, 2, ..., n_B$ with the current policy $\pi(\underline{a}_e(k)|\underline{s}_e(k), \theta(e))$.
    for the whole MC trajectory ($k = 1, 2, ..., n_B$) do
        Estimate the return as
            $$G_e(k) = \sum_{\kappa=k+1}^{n_B} \gamma^{\kappa-k} \underline{r}_e(\kappa) \in \mathbb{R} \tag{2}$$
         with $\gamma \in [0, 1)$ being the discount factor.
        Update policy parameters with gradient ascent (Eq. (1)).
    Increment $e$.
```

Taking Algorithm 1 further, it is possible to compute the episodic policy change $\frac{\partial \pi(\underline{a}_e|\underline{s}_e, \theta(e))}{\partial e}$ due to batch e, assuming gradient flow ($\alpha \to 0$).

Theorem 1. *Given batch* $\{\underline{s}_e(k), \underline{a}_e(k), \underline{r}_e(k)\}_{k=1}^{n_B}$*, and assuming gradient flow, the episodic policy change with the REINFORCE algorithm at the batch state-action pairs are*

$$\frac{\partial \underline{\pi}(\underline{a}_e|\underline{s}_e, \theta(e))}{\partial e} = \underline{\underline{\Theta}}_{\pi,e}(s, s')\underline{\underline{\Pi}}_e^I(\underline{s}_e, \underline{a}_e, \theta(e))\underline{G}_e(\underline{r}_e),$$

where $\underline{\underline{\Theta}}_{\pi,e}(s, s') \in \mathbb{R}^{n_B \times n_B}$ *is the neural tangent kernel,* $\underline{\underline{\Pi}}_e^I(\underline{s}_e, \underline{a}_e, \theta(e)) \in \mathbb{R}^{n_B \times n_B}$ *is a diagonal matrix containing the inverse policies (if they exist) at state-action pairs of batch e, and* $\underline{G}_e(\underline{r}_e) \in \mathbb{R}^{n_B}$ *is the vector of returns.*

Proof. Assuming very small learning rate α, the update algorithm (gradient ascent) can be written in continuous form (gradient flow), [20]:

$$\frac{d\theta(e)}{de} = \sum_{k=1}^{n_B} \left(G_e(k) \frac{\partial}{\partial \theta} log\pi(\underline{a}_e(k)|\underline{s}_e(k), \theta_e) \right)^T. \tag{3}$$

Furthermore, to avoid division by zero, it is assumed that the evaluated policy is strictly positive. The derivative on the left hand side is a column vector with size n_θ. Denote $\underline{s}_e = \{\underline{s}_e(k)\}_{k=1}^{n_B}$, $\underline{a}_e = \{\underline{a}_e(k)\}_{k=1}^{n_B}$, $\underline{r}_e = \{\underline{r}_e(k)\}_{k=1}^{n_B}$ and rewrite the differential equation in vector form as

$$\frac{d\theta(e)}{de} = \dot{\underline{\underline{\Pi}}}_{log}(\underline{s}_e, \underline{a}_e, \theta(e))\underline{G}_e(\underline{r}_e), \tag{4}$$

where

$$\underline{G}_e(\underline{r}_e) = [G_e(1), G_e(2), ..., G_e(n_B)]^T \tag{5}$$

is the vector of returns in episode e (Eq. (2)). The matrix of partial log policy derivatives ($\dot{\underline{\underline{\Pi}}}_{log}(\underline{s}_e, \underline{a}_e, \theta(e)) \in \mathbb{R}^{n_\theta \times n_B}$) is

$$\dot{\underline{\underline{\Pi}}}_{log}(\underline{s}_e, \underline{a}_e, \theta(e)) = \begin{bmatrix} \frac{\partial log\pi(\underline{a}_e(1)|\underline{s}_e(1),\theta(e))}{\partial \theta_1} & \cdots & \frac{\partial log\pi(\underline{a}_e(n_B)|\underline{s}_e(n_B),\theta(e))}{\partial \theta_1} \\ \frac{\partial log\pi(\underline{a}_e(1)|\underline{s}_e(1),\theta(e))}{\partial \theta_2} & \cdots & \frac{\partial log\pi(\underline{a}_e(n_B)|\underline{s}_e(n_B),\theta(e))}{\partial \theta_2} \\ \vdots & \ddots & \vdots \\ \frac{\partial log\pi(\underline{a}_e(1)|\underline{s}_e(1),\theta(e))}{\partial \theta_{n_\theta}} & \cdots & \frac{\partial log\pi(\underline{a}_e(n_B)|\underline{s}_e(n_B),\theta(e))}{\partial \theta_{n_\theta}} \end{bmatrix}, \tag{6}$$

where subscripts of θ denote weights and biases of the policy network. By using an element-wise transformation $\frac{logf(x)}{dx} = \frac{f'(x)}{f(x)}$, $\dot{\underline{\underline{\Pi}}}_{log}(\underline{s}_e, \underline{a}_e, \theta(e))$ can be rewritten as a product:

$$\dot{\underline{\underline{\Pi}}}_{log}(\underline{s}_e, \underline{a}_e, \theta(e)) =$$
$$\begin{bmatrix} \frac{\partial \pi(\underline{a}_e(1)|\underline{s}_e(1),\theta(e))}{\partial \theta_1} & \cdots & \frac{\partial \pi(\underline{a}_e(n_B)|\underline{s}_e(n_B),\theta(e))}{\partial \theta_1} \\ \frac{\partial \pi(\underline{a}_e(1)|\underline{s}_e(1),\theta(e))}{\partial \theta_2} & \cdots & \frac{\partial \pi(\underline{a}_e(n_B)|\underline{s}_e(n_B),\theta(e))}{\partial \theta_2} \\ \vdots & \ddots & \vdots \\ \frac{\partial \pi(\underline{a}_e(1)|\underline{s}_e(1),\theta(e))}{\partial \theta_{n_\theta}} & \cdots & \frac{\partial \pi(\underline{a}_e(n_B)|\underline{s}_e(n_B),\theta(e))}{\partial \theta_{n_\theta}} \end{bmatrix} \cdot$$
$$\cdot \begin{bmatrix} \frac{1}{\pi(\underline{a}_e(1)|\underline{s}_e(1),\theta(e))} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \frac{1}{\pi(\underline{a}_e(n_B)|\underline{s}_e(n_B),\theta(e))} \end{bmatrix}, \tag{7}$$

where the matrix on the left is a transposed Jacobian, i.e. $\dot{\underline{\underline{\Pi}}}(\underline{s}_e, \underline{a}_e, \theta(e)) = \left(\frac{\partial}{\partial\theta}\underline{\pi}(\underline{a}_e|\underline{s}_e, \theta(e))\right)^T$. Denote the inverse policies $\frac{1}{\pi(\underline{a}_e(k)|\underline{s}_e(k),\theta(e))}$ with $\pi_e^I(k)$ and $\underline{\pi}_e^I(\underline{s}_e, \underline{a}_e, \theta(e)) = \{\pi_e^I(k)\}_{k=1}^{n_B} \in \mathbb{R}^{n_B}$. The change of the agent weights based on batch e is

$$\frac{d\theta(e)}{de} = \dot{\underline{\underline{\Pi}}}(\underline{s}_e, \underline{a}_e, \theta(e))\underline{\underline{\Gamma}}_e(\underline{r}_e)\underline{\pi}_e^I(\underline{s}_e, \underline{a}_e, \theta(e)) \tag{8}$$

with $\underline{\underline{\Gamma}}_e(\underline{r}_e) = diag(\underline{G}_e(\underline{r}_e)) \in \mathbb{R}^{n_B \times n_B}$. Next, write the learning dynamics of the policy, using the chain rule:

$$\frac{\partial \underline{\pi}(\underline{a}_e|\underline{s}_e, \theta(e))}{\partial e} = \frac{\partial}{\partial \theta}\underline{\pi}(\underline{a}_e|\underline{s}_e, \theta(e))\frac{d\theta(e)}{de}. \tag{9}$$

First, extract $\frac{d\theta(e)}{de}$ as in Eq. (8):

$$\frac{\partial \underline{\pi}(\underline{a}_e|\underline{s}_e, \theta(e))}{\partial e} =$$
$$\dot{\underline{\underline{\Pi}}}(\underline{s}_e, \underline{a}_e, \theta(e))^T \dot{\underline{\underline{\Pi}}}(\underline{s}_e, \underline{a}_e, \theta(e))\underline{\underline{\Gamma}}_e(\underline{r}_e)\underline{\pi}_e^I(\underline{s}_e, \underline{a}_e, \theta(e)). \tag{10}$$

Note that $\dot{\underline{\underline{\Pi}}}(\underline{s}_e, \underline{a}_e, \theta(e))^T \dot{\underline{\underline{\Pi}}}(\underline{s}_e, \underline{a}_e, \theta(e)) = \frac{\partial}{\partial\theta}\underline{\pi}(\underline{a}_e|\underline{s}_e, \theta(e)) \left(\frac{\partial}{\partial\theta}\underline{\pi}(\underline{a}_e|\underline{s}_e, \theta(e))\right)^T$ is the Neural Tangent Kernel (NTK), as defined in [14]. Denote it with $\underline{\underline{\Theta}}_{\pi,e}(s, s') \in \mathbb{R}^{n_B \times n_B}$. Finally, the policy update due to episode batch e at states $\underline{s}_e(k)$ for actions $\underline{a}_e(k)$ become:

$$\frac{\partial \underline{\pi}(\underline{a}_e|\underline{s}_e, \theta(e))}{\partial e} = \underline{\underline{\Theta}}_{\pi,e}(s, s')\underline{\underline{\Gamma}}_e\underline{\pi}_e^I(\underline{s}_e, \underline{a}_e, \theta(e)). \tag{11}$$

Similarly, by defining $\underline{\underline{\Pi}}_e^I(\underline{s}_e, \underline{a}_e, \theta(e)) = diag(\underline{\pi}_e^I(\underline{s}_e, \underline{a}_e, \theta(e)))$ we can write Eq. (12) as

$$\frac{\partial \underline{\pi}(\underline{a}_e|\underline{s}_e, \theta(e))}{\partial e} = \underline{\underline{\Theta}}_{\pi,e}(s, s')\underline{\underline{\Pi}}_e^I(\underline{s}_e, \underline{a}_e, \theta(e))\underline{G}_e(\underline{r}_e) \tag{12}$$

too. □

Remark. *From Eq. (11), the learning dynamics of the REINFORCE algorithm is a nonlinear differential equation system. If the same data batch is shown to the neural network over and over again, the policy evolves as* $\dot{x}(t) = \beta\frac{1}{x(t)}$.

2.2. Evaluating the policy change for arbitrary states and actions

In this section, we describe the policy change for any state and any action if the agent learns from batch e. In most cases, the learning agent can perform multiple actions. Assume the agent can take $a^1, a^2, ..., a^{n_A}$ actions (the policy net has n_A output channels). Previous works that deal with NTK all consider one output channel in the examples (e.g. [4], [14], [32]), however state that it works for multiple outputs too. [14] claim that a network with n_A outputs can be handled as n_A independent neural networks. On the other hand, it is possible to fit every output channel into one equation by generalizing Theorem 1. First, reconstruct the return vector $\underline{G}_e$ as follows.

$$\underline{G}_e(\underline{a}_e, \underline{r}_e) = [G_e^{a_1}(1), G_e^{a_2}(1), ..., G_e^{a_{n_A}}(n_B)]^T \in \mathbb{R}^{n_A n_B}, \tag{13}$$

where

$$G_e^{a_i}(k) = \begin{cases} G_e(k) & \text{if } a^i \text{ is taken at step } k \\ 0 & \text{otherwise} \end{cases} \tag{14}$$

and $i = 1, 2, ...n_A$. In other words, $\underline{G}_e(\underline{a}_e, \underline{r}_e)$ consists of $n_A \times 1$ sized blocks with zeros at action indexes which are not taken at $\underline{s}_e(k)$, ($n_A - 1$ zeros) and the original return (Eq. (2)) at the position of the taken action. Note that action dependency of $\underline{G}_e(\underline{a}_e, \underline{r}_e)$ stems from this assumption.

Theorem 2. *Given batch* $\{\underline{s}_e(k), \underline{a}_e(k), \underline{r}_e(k)\}_{k=1}^{n_B}$, *and assuming gradient flow, the episodic policy change with the REINFORCE algorithm at the batch states for an* n_A *output policy network is*

$$\frac{\partial \underline{\pi}(\underline{a}|\underline{s}_e, \theta(e))}{\partial e} = \underline{\underline{\Theta}}_{\pi,e}(s, s')\underline{\underline{\Pi}}_e^I(\underline{s}_e, \theta(e))\underline{G}_e(\underline{a}_e, \underline{r}_e),$$

where $\frac{\partial \underline{\pi}(\underline{a}|\underline{s}_e, \theta(e))}{\partial e} \in \mathbb{R}^{n_A n_B}$ *and* $\underline{\underline{\Theta}}_{\pi,e}(s, s') \in \mathbb{R}^{n_A n_B \times n_A n_B}$.

Proof. We can deduce the multi-output case by rewriting Eq. (4). For simplicity, we keep the notations, but the matrix and vector sizes are redefined for the vector output case. Therefore, the log policy derivatives are evaluated for every possible action at the states contained in a batch:

$$\underline{\underline{\dot{\Pi}}}_{log}(\underline{s}_e, \theta(e)) =$$
$$\begin{bmatrix} \frac{\partial log\pi(a^1|\underline{s}_e(1),\theta(e))}{\partial\theta_1} & \dots & \frac{\partial\pi(a^{n_A}|\underline{s}_e(n_B),\theta(e))}{\partial\theta_1} log \\ \frac{\partial}{\partial\theta_2} log\pi(a^1|\underline{s}_e(1),\theta(e)) & \dots & \frac{\partial log\pi(a^{n_A}|\underline{s}_e(n_B),\theta(e))}{\partial\theta_2} \\ \vdots & \ddots & \vdots \\ \frac{\partial log\pi(a^1|\underline{s}_e(1),\theta(e))}{\partial\theta_{n_\theta}} & \dots & \frac{\partial log\pi(a^{n_A}|\underline{s}_e(n_B),\theta(e))}{\partial\theta_{n_\theta}} \end{bmatrix}, \tag{15}$$

$\underline{\underline{\dot{\Pi}}}_{log}(\underline{s}_e, \theta(e)) \in \mathbb{R}^{n_\theta \times n_A n_B}$. The zero elements in $\underline{G}_e(\underline{a}_e, \underline{r}_e)$ will cancel out log probabilities of actions that are not taken in episode e, (Eq. (14)). Therefore, the final output $\frac{\partial \underline{\pi}(\underline{a}_e|\underline{s}_e,\theta(e))}{\partial e}$ does not change. Note that, the action dependency is moved from $\underline{\underline{\dot{\Pi}}}_{log}(\underline{s}_e, \theta(e))$ to $\underline{G}_e(\underline{a}_e, \underline{r}_e)$. That is, because the policy is evaluated for every output channel of the policy network, but nonzero reward is given only if an action is actually taken in the MC trajectory. Continue by separating $\underline{\underline{\dot{\Pi}}}_{log}(\underline{s}_e, \theta(e))$ into two matrices, in the same way as in Eq. (7).

$$\underline{\underline{\dot{\Pi}}}_{log}(\underline{s}_e, \theta(e)) = \underline{\underline{\dot{\Pi}}}(\underline{s}_e, \theta(e))\underline{\underline{\Pi}}^I_e(\underline{s}_e, \theta(e)) =$$
$$\begin{bmatrix} \frac{\partial\pi(a^1|\underline{s}_e(1),\theta(e))}{\partial\theta_1} & \frac{\partial\pi(a^2|\underline{s}_e(1),\theta(e))}{\partial\theta_1} & \dots & \frac{\partial\pi(a^{n_A}|\underline{s}_e(n_B),\theta(e))}{\partial\theta_1} \\ \frac{\partial\pi(a^1|\underline{s}_e(1),\theta(e))}{\partial\theta_2} & \frac{\partial\pi(a^2|\underline{s}_e(1),\theta(e))}{\partial\theta_2} & \dots & \frac{\partial\pi(a^{n_A}|\underline{s}_e(n_B),\theta(e))}{\partial\theta_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial\pi(a^1|\underline{s}_e(1),\theta(e))}{\partial\theta_{n_\theta}} & \frac{\partial\pi(a^2|\underline{s}_e(1),\theta(e))}{\partial\theta_{n_\theta}} & \dots & \frac{\partial\pi(a^{n_A}|\underline{s}_e(n_B),\theta(e))}{\partial\theta_{n_\theta}} \end{bmatrix}.$$
$$\cdot \begin{bmatrix} \frac{1}{\pi(a^1|\underline{s}_e(1),\theta(e))} & & & \\ & \frac{1}{\pi(a^2|\underline{s}_e(1)\theta(e))} & & \\ & & \ddots & \\ & & & \frac{1}{\pi(a^{n_A}|\underline{s}_e(n_B),\theta(e))} \end{bmatrix}, \tag{16}$$

where the diagonalized inverse policies are denoted with with $\underline{\underline{\Pi}}^I_e(\underline{s}_e, \theta(e))$. The weight change can be written as

$$\frac{d\theta(e)}{de} = \underline{\underline{\dot{\Pi}}}(\underline{s}_e, \theta(e))\underline{\underline{\Pi}}^I_e(\underline{s}_e, \theta(e))\underline{G}_e(\underline{a}_e, \underline{r}_e). \tag{17}$$

Following the same steps as for the proof of Theorem 1, the policy change for every output channel is

$$\frac{\partial \underline{\pi}(\underline{a}|\underline{s}_e, \theta(e))}{\partial e} = \underline{\underline{\Theta}}_{\pi,e}(s, s')\underline{\underline{\Pi}}_e^I(\underline{s}_e, \theta(e))\underline{G}_e(\underline{a}_e, \underline{r}_e). \tag{18}$$

□

Remark. *It is possible to write the average change of each policy output channel over a batch (with superscript B) as*

$$\frac{\partial \underline{\pi}^B(\underline{a}|\underline{s}_e, \theta(e))}{\partial e} = \\ \frac{1}{n_B}\underline{\underline{E}}\,\underline{\underline{\Theta}}_{\pi,e}(s, s')\underline{\underline{\Pi}}_e^I(\underline{s}_e, \theta(e))\underline{G}_e(\underline{a}_e, \underline{r}_e). \tag{19}$$

where $\underline{\underline{E}}$ is an $n_A \times n_A n_B$ matrix consisting of n_B eye matrices of size $n_A \times n_A$. $\underline{\underline{E}}$ is used to sum elements that correspond to the same output channel. The result of Eq. (19) is the policy change at an averaged state of $\underline{s}_e$.

With Theorem 2, it is possible to evaluate how the policy will change at states $\underline{s}_e$ when learning on batch data e if the learning rate is small. By manipulating Eq. (18) policy change can be evaluated at states not part of batch $\underline{s}_e$ too.

Theorem 3. *Given batch $\{\underline{s}_e(k), \underline{a}_e(k), \underline{r}_e(k)\}_{k=1}^{n_B}$, and assuming gradient flow, the episodic policy change with the REINFORCE algorithm at any state $s_0 \notin \underline{s}_e$ is*

$$\frac{\partial \underline{\pi}(a|s_0, \theta(e))}{\partial e} = \underline{\underline{\vartheta}}(\underline{s}_e, s_0)\underline{\underline{\Pi}}_e^I(\underline{s}_e, \theta(e))\underline{G}_e(\underline{a}_e, \underline{r}_e) \in \mathbb{R}^{n_A},$$

where $\underline{\underline{\vartheta}}(\underline{s}_e, s_0) = [\underline{\underline{\vartheta}}(s_1, s_0), \underline{\underline{\vartheta}}(s_2, s_0), ..., \underline{\underline{\vartheta}}(s_{n_B}, s_0)] \in \mathbb{R}^{n_A \times n_A n_B}$ is the neural tangent kernel evaluated for all $\underline{s}_e$, s_0 pairs.

Proof. First, we concatenate the states and returns $(\underline{s}_e, s_0)$, $(\underline{G}_e(\underline{a}_e, \underline{r}_e), \underline{G}_0(a_0, r_0))$, respectively and solve Eq. (18) for s_0. For more insight, we illustrate the matrix multiplication in Eq. (18) with the concatenated state in Figure 1.

$$\frac{\partial \underline{\pi}(a|s_0, \theta(e))}{\partial e} = \underline{\underline{\vartheta}}(\underline{s}_e, s_0)\underline{\underline{\Pi}}_e^I(\underline{s}_e, \theta(e))\underline{G}_e(\underline{a}_e, \underline{r}_e) + \\ + \underline{\underline{\vartheta}}(s_0, s_0)\underline{\underline{\Pi}}_e^I(s_0, \theta(e))\underline{G}_0(\underline{a}_0, \underline{r}_0). \tag{20}$$

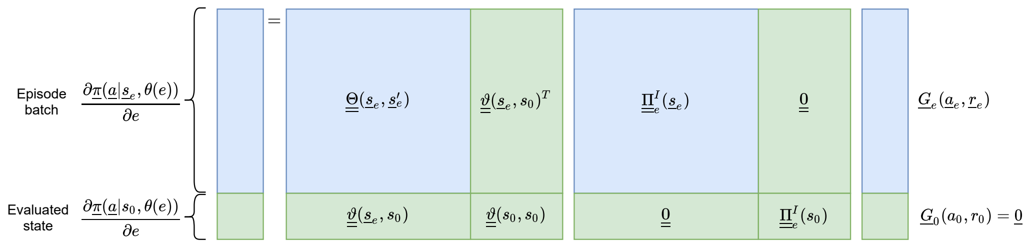

Episode batch
$\frac{\partial \underline{\pi}(\underline{a}|\underline{s}_e, \theta(e))}{\partial e}$
Evaluated state
$\frac{\partial \underline{\pi}(\underline{a}|s_0, \theta(e))}{\partial e}$
=
$\underline{\underline{\Theta}}(\underline{s}_e, \underline{s}'_e)$
$\underline{\underline{\vartheta}}(\underline{s}_e, s_0)^T$
$\underline{\underline{\Pi}}^I_e(\underline{s}_e)$
$\underline{\underline{0}}$
$\underline{G}_e(\underline{a}_e, \underline{r}_e)$
$\underline{\underline{\vartheta}}(\underline{s}_e, s_0)$
$\underline{\underline{\vartheta}}(s_0, s_0)$
$\underline{\underline{0}}$
$\underline{\underline{\Pi}}^I_e(s_0)$
$\underline{G}_0(a_0, r_0) = \underline{0}$

Figure 1: Matrix compatibility with the augmented batch. The blue section indicates the original Eq. (18), while the green blocks stem from the augmentation.

The NTK is based on the partial derivatives of the policy network and can be evaluated anywhere. Therefore, $\underline{\underline{\vartheta}}(\underline{s}_e, s_0) = [\underline{\underline{\vartheta}}(s_1, s_0), \underline{\underline{\vartheta}}(s_2, s_0), ..., \underline{\underline{\vartheta}}(s_{n_B}, s_0)] \in \mathbb{R}^{n_A \times n_A n_B}$ can be computed for any state. $\underline{\underline{\vartheta}}(\underline{s}_e, s_0)$ consists of symmetric $n_A \times n_A$ blocks. Since s_0 is not included in the learning, it does not affect the policy change, its return is zero for every action, $\underline{G}_0(a_0, r_0) = \underline{0} \in \mathbb{R}^{n_A}$. The zero return cancels out the term $\underline{\underline{\vartheta}}(s_0, s_0)\underline{\underline{\Pi}}^I_e(s_0, \theta(e))\underline{G}_0(a_0, \underline{r}_0) \in \mathbb{R}^{n_A}$. Therefore,

$$\frac{\partial \underline{\pi}(a|s_0, \theta(e))}{\partial e} = \underline{\underline{\vartheta}}(\underline{s}_e, s_0)\underline{\underline{\Pi}}^I_e(\underline{s}_e, \theta(e))\underline{G}_e(\underline{a}_e, \underline{r}_e) \in \mathbb{R}^{n_A}. \tag{21}$$

□

The numerical accuracy of the computed policy change is further discussed in Appendix A.

3. The NTK-based constrained REINFORCE algorithm

Every physical system has some limits (saturation). Intuitively, the agent should not increase the control action towards the limit if the system is already in saturation (for example, if a robot manipulator is at an end position, increasing the force in the direction of that end position is pointless). Assuming we have some idea how the agent should behave (which actions to take) at specific states $\underline{s}_s = [s_{s1}, s_{s2}, ..., s_{sn_S}]$, equality and inequality constraints can be prescribed for the policy. Define equality and inequality constraints as $\underline{\pi}_{ref,eq}(\underline{a}_s|\underline{s}_s, \theta(e)) = \underline{c}_{eq}$ and $\underline{\pi}_{ref,ineq}(\underline{a}_s|\underline{s}_s, \theta(e)) \geq \underline{c}_{ineq}$, where $[\underline{c}_{eq},\ \underline{c}_{ineq}] \in \mathbb{R}^{n_S}$ is a vector constant probabilities.

In the sequel, relying on Theorem 3, we provide the mathematical deduction on how to enforce constraints during learning.

3.1. Equality constraints

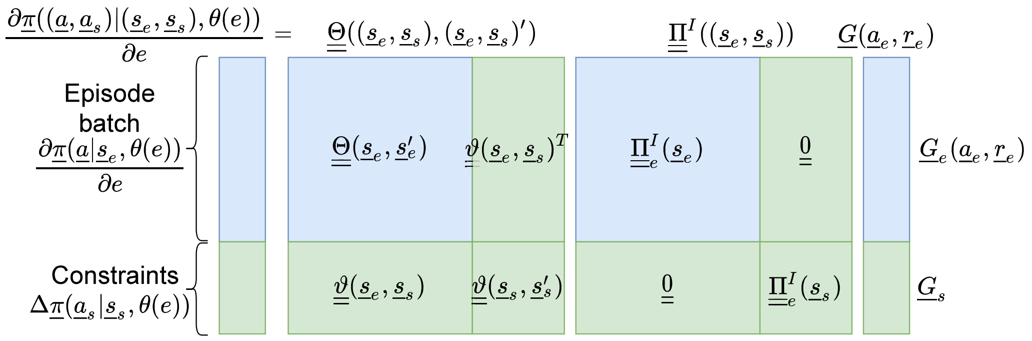
To deal with constraints, we augment Eq. (18) with the constrained state-action pairs. Visually, the policy change at the augmented batch states are shown in Figure 2. Since the desired policy change at the safe states can be given as

$$\Delta\underline{\pi}(\underline{a}_s|\underline{s}_s, \theta(e)) = \underline{\pi}_{ref,eq}(\underline{a}_s|\underline{s}_s, \theta(e)) - \underline{\pi}(\underline{a}_s|\underline{s}_s, \theta(e)), \tag{22}$$

$\Delta\underline{\pi}(\underline{a}_s|\underline{s}_s, \theta(e)) \in \mathbb{R}^{n_S}$. The only unknowns are the returns $\underline{G}_s = [G_{s1}, G_{s2}, ..., G_{sn_S}]$ for the safe actions at the safe states. Note that the upper block of Figure 2 contains differential equations, while the lower block consists of algebraic equations. It is sufficient to solve the algebraic part. With the lower blocks of Figure 2 we can write the linear equation system

$$\begin{aligned}&\Delta\underline{\pi}(\underline{a}_s|\underline{s}_s, \theta(e)) - \underline{\underline{\vartheta}}(\underline{s}_e, \underline{s}_s)\underline{\underline{\Pi}}^I_e(\underline{s}_e, \theta(e))\underline{G}_e(\underline{a}_e, \underline{r}_e) = \\ &\underline{\underline{\vartheta}}(\underline{s}_s, \underline{s}'_s)\underline{\underline{\Pi}}^I_e(\underline{s}_s, \theta(e))\underline{G}_s.\end{aligned} \tag{23}$$

This system has a single unique solution as there are n_S unknown returns and n_S equations and it can be solved with e.g. the DGSEV algorithm [12].



$\frac{\partial\underline{\pi}((\underline{a},\underline{a}_s)|(\underline{s}_e,\underline{s}_s),\theta(e))}{\partial e}$ =
$\underline{\underline{\Theta}}((\underline{s}_e,\underline{s}_s),(\underline{s}_e,\underline{s}_s)')$
$\underline{\underline{\Pi}}^I((\underline{s}_e,\underline{s}_s))$
$\underline{G}(\underline{a}_e,\underline{r}_e)$
Episode batch
$\frac{\partial\underline{\pi}(\underline{a}|\underline{s}_e,\theta(e))}{\partial e}$
$\underline{\underline{\Theta}}(\underline{s}_e,\underline{s}'_e)$
$\underline{\underline{\vartheta}}(\underline{s}_e,\underline{s}_s)^T$
$\underline{\underline{\Pi}}^I_e(\underline{s}_e)$
$\underline{\underline{0}}$
$\underline{G}_e(\underline{a}_e,\underline{r}_e)$
Constraints
$\Delta\underline{\pi}(\underline{a}_s|\underline{s}_s,\theta(e))$
$\underline{\underline{\vartheta}}(\underline{s}_e,\underline{s}_s)$
$\underline{\underline{\vartheta}}(\underline{s}_s,\underline{s}'_s)$
$\underline{\underline{0}}$
$\underline{\underline{\Pi}}^I_e(\underline{s}_s)$
$\underline{G}_s$

Figure 2: Matrix compatibility of the constrained policy change evaluation (concatenated batch)

In order to obey the constraints, a safe data batch is constructed. We concatenate the safe states, actions, and computed returns with the episode batch as: $\{(\underline{s}_e, \underline{s}_s), (\underline{a}_e, \underline{a}_s), (\underline{G}_e(\underline{a}_e, \underline{r}_e), \underline{G}_s)\}$. Then, the agent's weights are updated with the appended batch with gradient ascent, Eq. (1).

In the initial stages of learning, the difference between the reference policies and the actual ones will be large. Therefore high rewards are needed to eliminate this difference. This also means that the effect of the collected batch on the weights is minor compared to the safe state-action-return tuples. In addition, large returns might cause loss of stability during the learning. The returns computed from the linearized policy change might differ from the actual one, especially if large steps are taken (i.e. large returns are applied). On the other hand, when the policy obeys the constraints, the computed returns will be small and will only compensate for the effect of the actual batch. In a special case when the return is zero, the result is the unconstrained policy change at the specific state as in Section 2.2. In addition, if the policy is smooth, the action probabilities near the constrained points will be similar. In continuous state space, this implies that defining grid-based (finite) constraints is sufficient.

Remark. ***Time complexity:*** *The critical operations are kernel evaluations and solving the linear equation system. The DGSEV algorithm used for solving the linear equation system has time complexity $\mathcal{O}(n_S^3)$ [12]. The time complexity of kernel evaluations is $\mathcal{O}((n_B + n_S)n_S)$. If the kernel is computed for every output channel at the batch states, time complexity increases to $\mathcal{O}((n_A n_B + n_S)n_S)$.*

3.2. Inequality constraints

In the same way, inequality constraints can be prescribed too. Assume the there are some states of the environment where an action shall be taken with at least a constant probability: $\underline{\pi}_{ref,ineq}(\underline{a}_s|\underline{s}_s, \theta(e)) \geq \underline{c}_{ineq}$. Then, similar to Eq. (23), the inequality constraints can be written as

$$\begin{aligned}&\Delta\underline{\pi}(\underline{a}_s|\underline{s}_s, \theta(e)) - \underline{\underline{\vartheta}}(\underline{s}_e, \underline{s}_s)\underline{\underline{\Pi}}_e^I(\underline{s}_e, \theta(e))\underline{G}_e(\underline{a}_e, \underline{r}_e) \\ &\geq \underline{\underline{\vartheta}}(\underline{s}_s, \underline{s}_s')\underline{\underline{\Pi}}_e^I(\underline{s}_s, \theta(e))\underline{G}_s.\end{aligned} \tag{24}$$

Solving this system of inequalities can be turned into a convex quadratic programming problem. Since the original goal of reinforcement learning is learning on the collected episode batch data, the influence of the constraints on the learning (i.e. the magnitude of $\underline{G}_s$) should be as small as possible. Therefore, the quadratic program is

$$\min \sum_{i=1}^{n_S} G_{si}^2 \tag{25}$$

subject to Eq. (24).

Note that, the quadratic cost function is needed to similarly penalize positive and negative returns. Quadratic programming with interior point methods has polynomial time complexity ($\mathcal{O}(n_S^3)$, [34]) and has to be solved after every episode. In practical applications, numerical errors are more more of an issue than time complexity (i.e. solving the optimization with thousands of constraints).

We summarize the NTK-based constrained REINFORCE algorithm in Algorithm 2.

Algorithm 2: The NTK-based constrained REINFORCE algorithm

```
Initialize $e = 1$.
Define equality constraints $\underline{\pi}_{ref,eq}(\underline{a}_s|\underline{s}_s, \theta(e)) = \underline{c}_{eq}$.
Define inequality constraints $\underline{\pi}_{ref,ineq}(\underline{a}_s|\underline{s}_s, \theta(e)) \geq \underline{c}_{ineq}$.
Initialize the policy network with random $\theta$ weights.
while not converged do
    Generate a Monte-Carlo trajectory $\{\underline{s}_e(k), \underline{a}_e(k), \underline{r}_e(k)\}$,
     $k = 1, 2, ..., n_B$ with the current policy $\pi(\underline{a}_e(k)|\underline{s}_e(k), \theta(e))$.
    for the whole MC trajectory ($k = 1, 2, ..., n_B$) do
        Compute the returns $G_e(k)$ with Eq. (2).
    Construct $\underline{\underline{\vartheta}}(\underline{s}_e, \underline{s}_s)$, $\underline{\underline{\vartheta}}(\underline{s}_s, \underline{s}_s')$, $\underline{\underline{\Pi}}_e^I(\underline{s}_e, \theta(e))$, $\underline{\underline{\Pi}}_e^I(\underline{s}_s, \theta(e))$.
    Compute $\underline{G}_s$ with Eq. (23) and Eq. (24).
    Concatenate the MC batch and the constraints:
     $\{(\underline{s}_e, \underline{s}_s), (\underline{a}_e, \underline{a}_s), (\underline{G}_e(\underline{a}_e, \underline{r}_e), \underline{G}_s)\}$.
    for the augmented MC trajectory ($k = 1, 2, ..., n_B + n_S$) do
        Update policy parameters with gradient ascent (Eq. (1)).
    Increment $e$.
```

Remark. *One extension of REINFORCE is policy gradient with baseline. There, a baseline (typically the value function) is subtracted from the gains to reduce variance. The policy is then updated with these modified returns using the policy gradient theorem [25]. Constraints can be adapted to the policy gradient with baseline too. Since the returns at the constrained states are shaped to satisfy specific action probabilities, baselines should not be subtracted from the safe returns. Therefore, in a constrained REINFORCE with baseline, batch returns are offset by the baseline while the safe returns are not.*

4. Experimental studies

We evaluate the proposed constrainted learning algorithm in two OpenAI gym environments: in the Cartpole-v0 and in the Lunar Lander-v2 environments (Figure 3), [5].

Figure 3: Visualization of the environments. Left: cartpole, right: lunar lander.

4.1. Cartpole v0

The cart pole problem (also known as the inverted pendulum) is a common benchmark in control theory. The goal is to balance a pole to remain upright by horizontally moving the cart. In the OpenAI gym environment, it is implemented under the name Cartple-v0 [2]. The agent in this environment can take two actions: accelerating the cart left (a^0) or right (a^1). The cart pole has four states: the position of the cart (x), its velocity ($\dot{x}$), the pole angle (φ), and the pole angular velocity ($\dot{\varphi}$). In the reinforcement learning setting, the agent's goal is to balance the pole as long as possible. Reward is given for every discrete step if the pole is in vertical direction, and the episode ends if the pole falls or successfully balancing for 200 steps. The pass criteria for this gym environment is reaching an average reward of 195 for 100 episodes.

The learning agent is a 4 input, 2 output, 2 layer deep fully-connected ReLU network with softmax output nonlinearity. The hidden layer width is 5000 neurons with bias terms. That is to comply with the assumptions in [14], i.e. a shallow and wide neural network. On the other hand, the NTK can be computed for more complex NN structures too e.g. [32]. Note that the primary purpose is not finding the best function approximator, merely demonstrating the efficiency of the constrained learning. To achieve lazy learning, the learning rate is set to $\alpha = 0.0001$. The small learning rate ensures that the approximation of the policy change remains accurate, see Appendix A.

At different cart positions, based on the pole's position and velocity, a reference policy is defined. Selecting suitable constraints requires some experimenting: selecting too many states to constrain slow down the computation significantly while defining conflicting constraints can make Eq. (24) unsolvable. Inequality constraints are imposed on the pole angle and its angular velocity at discrete cart positions resulting in 18 constrained states, see Table B.1 in Appendix B.

With the proper selection of constraints, it was possible to train the agent in 5 steps. Figure 4 shows the episode rewards of the trained agent. These results would place the algorithm in the top 10 of the OpenAI gym leaderboard, competing with deterministic policies or closed-loop controllers [19]. For comparison, the unconstrained learning is presented in Figure 5. For

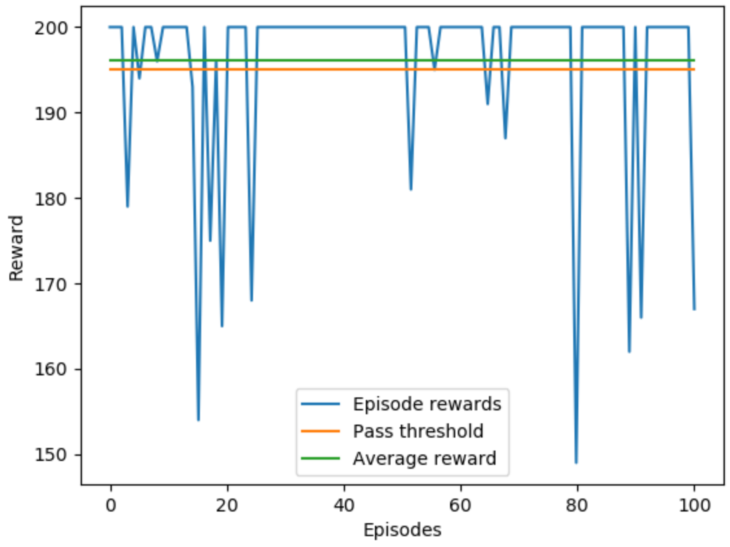


Figure 4: Episode rewards of the trained agent in the Cartpole-v0 environment (after 5 training steps). Pass criteria is an average reward above 195.

comparison, the unconstrained learning agent requires 1157 episodes to reach the pass criteria. This proves the assumption that constraining the policy greatly speeds up learning. Note that the agent is very simple. Selecting a more complex learning agent (e.g. [16] could most likely solve this problem within much fewer episodes.

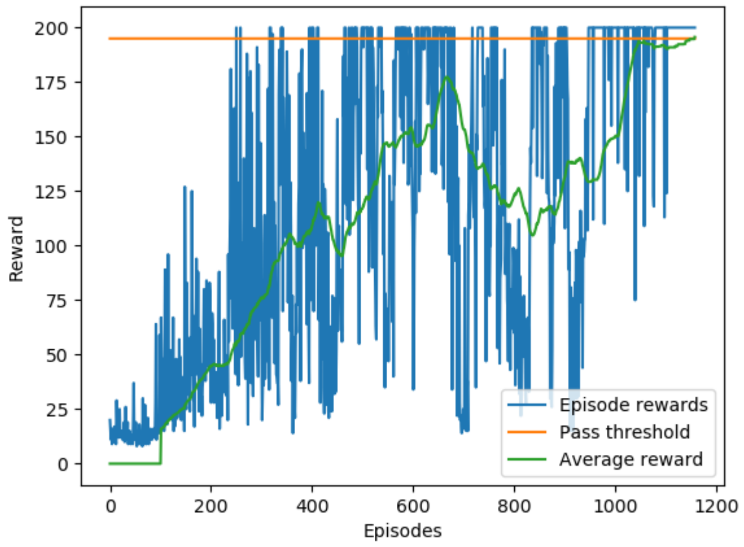


Figure 5: Episode rewards of learning in the Cartpole-v0 without safety constraints.

4.2. Lunar Lander v2

The goal in this 2D environment is to land a rocket on a landing pad without crashing. The agent in this environment has eight states: its horizontal and vertical coordinates (x, y) and velocities $(\dot{x}, \dot{y})$, its angle φ, its angular velocity $\dot{\varphi}$ and the logical states whether the left and right legs are in contact with the ground (l_{left} and l_{right}). The agent can choose from four actions: 0: do nothing, 1: fire the left thruster, 2: fire the main engine, and 3: fire the right thruster. The episode finishes if the lander crashes or comes to rest. Reward is given for successfully landing close to the landing pad. Crashing the rocket results in a penalty. Firing the engines (burning fuel) also results in small penalties. In OpenAI gym the pass criteria for solving this environment is an average reward of 200 for 100 episodes.

The learning agent is the same shallow and wide ReLU network as in Section 4.1, but with adjusted input (8) and output (4) sizes. The learning rate is $\alpha = 0.0001$ and the discount factor is $\gamma = 0.9$.

During unconstrained learning, the two most common reasons observed for episode failures were the lander crashing too fast into the ground and tilting over mid-flight. To this end, constraints were imposed on the vertical velocity and the angle of the lander.

To speed up the learning, inequality constraints are imposed, trying to keep the lander on an ideal trajectory: as the lander comes closer to the ground, it should decelerate by firing the main engine (simulating hover slam). If the rocket has a too large horizontal velocity or is tilted, the side engines should be used. Constraints are summarized in Table B.2 (Appendix B). In order to ease computational load, only every 10^{th} step of the Monte-Carlo trajectory was logged. Too long trajectories slow down computation as the NTK has to be evaluated at more points, which has polynomial time complexity.

With the above setup, the agent was able to land after a few episodes successfully. However, after 200 episodes of training, the 100 runs average reward was only around 100, see Figure 6. With a more careful selection of constraint states, the success rate of the agent could be improved. This highlights the drawback of the constrained approach: if the dimension of the environment space is large, the number of required constraints in a grid-based fashion increase significantly (i.e. the curse of dimensionality applies). Therefore, a lot more manual tuning effort is required to achieve fast learning. On the other hand, the algorithm can achieve similar performance in significantly less steps as e.g. SARSA, [9].

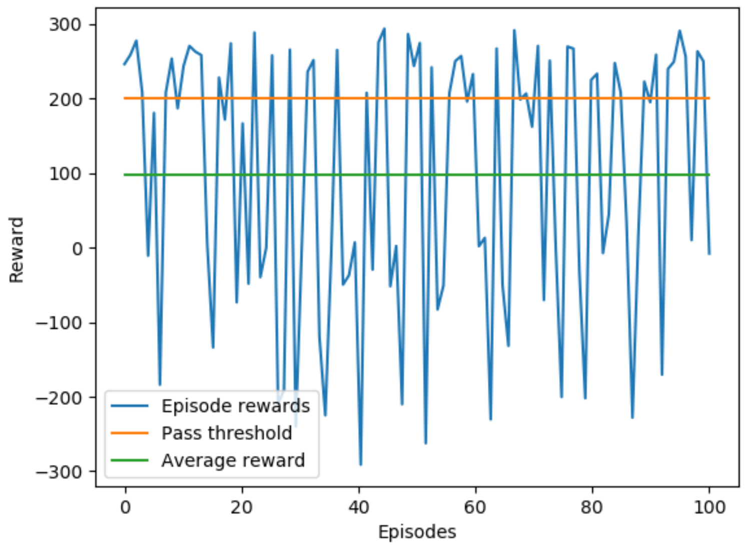


Figure 6: Episode rewards of the trained agent in the Lunar_lander-v2 environment (after 200 training steps). Pass criteria is an average reward above 200.

5. Conclusions

We proposed a solution to augment the REINFORCE algorithm with equality and inequality constraints. To this end, the policy evolution was computed with the help of the neural tangent kernel. Then, arbitrary states were selected with desired action probabilities. Next, for these arbitrary state-action pairs returns were computed that approximately satisfy the prescribed constraints under gradient ascent. The efficiency of the constrained learning was demonstrated with a shallow and wide ReLU network in the Cartpole and Lunar Lander OpenAI gym environments. Results suggest that constraints are satisfied after 2-3 episodes, and if they are set up correctly, learning becomes extremely fast (episode-wise) while satisfying safety constraints too. The computed returns become small if the constraints are satisfied, only slightly affecting the Monte-Carlo trajectory episode batch. On the other hand, selecting suitable constraints requires expert knowledge about the environment.

As a future line of research, other variants of policy gradient methods should be analyzed. We hypothesize that other, more complex policy-based approaches can be augmented with constraints using the NTK too.

Acknowledgements

This work has been supported and funded by the project RITE (funded by CHAIR, Chalmers University of Technology).

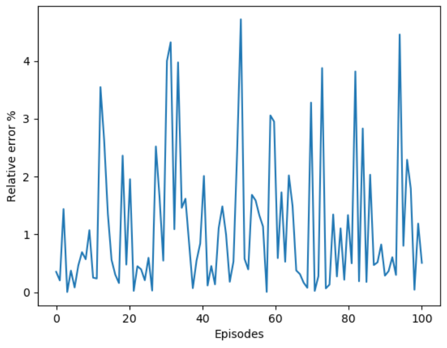
Appendix A. Accuracy of the prediction

Due to the linearization in the NTK (gradient calculation), by neglecting the non-differentiable property of the ReLU activation at the origin, as well as by numerical precision errors the NTK-based policy change prediction might be biased. In this appendix this bias is analyzed by comparing the predicted policy update (for the whole batch, Eq. (19)) $\frac{\partial \underline{\pi}^B(\underline{a}|\underline{s}_e, \theta(e))}{\partial e}$ with the actual one (at the batch average state) $\bar{q}_e(\underline{s}_e, \underline{a}_e, \theta(e), \theta(e+1)) = \frac{1}{n_B}\sum_{k=1}^{n_B}(\underline{\pi}(a_e(k)|s_e(k), \theta(e+1)) - \underline{\pi}(a_e(k)|s_e(k), \theta(e))$, assuming data batch $\{\underline{s}_e, \underline{a}_e, \underline{r}_e\}$. Assuming no constraints, the agent from Section 4.1 tries to learn to balance the pole in the cart pole environment for 100 episodes. The episode-by-episode relative errors during learning are computed with

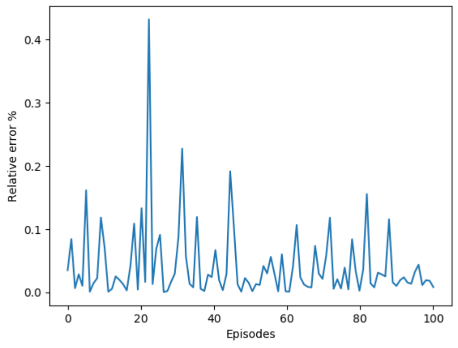
Eq. (A.1). Results are shown in Figure A.7 for three different learning rates.

$$\varepsilon = \frac{\frac{\partial \pi^B(\underline{a}|\underline{s}_e,\theta(e))}{\partial e} - \bar{q}_e(\underline{s}_e, \underline{a}_e, \theta(e), \theta(e+1)))}{(\bar{q}_e(\underline{s}_e, \underline{a}_e, \theta(e), \theta(e+1))} \cdot 100. \tag{A.1}$$

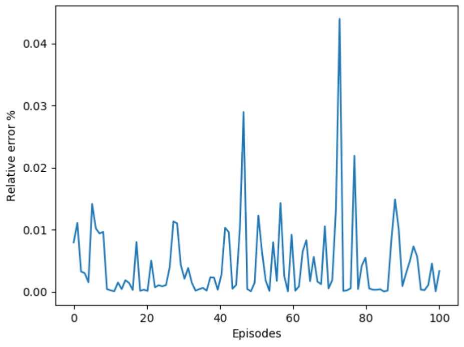
Results suggest that the prediction becomes more accurate as the learning rate decreases. The learning rate used in the simulations (Section 4, $\alpha = 0.001$) yields approximately 0.05% prediction error.



(a) $\alpha = 0.001$



(b) $\alpha = 0.0001$



(c) $\alpha = 0.00001$

Figure A.7: Relative estimation errors between the computed NTK based policy change and the actual, SGD based policy change (Eq. (A.1)). Results are shown for the first output channel of the policy network.

Appendix B. Constraints

	x	$\dot{x}$	φ	$\dot{\varphi}$	$\underline{\pi}_{ref}(a^0 \mid s_{si}, \theta(e))$
s_{s1}	-2	0	0.25	0.05	≤ 0.05
s_{s2}	-1.5	0	0.25	0.05	≤ 0.05
s_{s3}	-1	0	0.25	0.05	≤ 0.05
s_{s4}	-0.5	0	0.25	0.05	≤ 0.05
s_{s5}	0	0	0.25	0.05	≤ 0.05
s_{s6}	0.5	0	0.25	0.05	≤ 0.05
s_{s7}	1	0	0.25	0.05	≤ 0.05
s_{s8}	1.5	0	0.25	0.05	≤ 0.05
s_{s9}	2	0	0.25	0.05	≤ 0.05
s_{s10}	-2	0	-0.25	-0.05	≤ 0.95
s_{s11}	-1.5	0	-0.25	-0.05	≥ 0.95
s_{s12}	-1	0	-0.25	-0.05	≥ 0.95
s_{s13}	-0.5	0	-0.25	-0.05	≥ 0.95
s_{s14}	0	0	-0.25	-0.05	≥ 0.95
s_{s15}	0.5	0	-0.25	-0.05	≥ 0.95
s_{s16}	1	0	-0.25	-0.05	≥ 0.95
s_{s17}	1.5	0	-0.25	-0.05	≥ 0.95
s_{s18}	2	0	-0.25	-0.05	≥ 0.95

Table B.1: Constrained states in Cartpole-v0

	x	y	$\dot{y}$	φ	l	$\underline{\pi}_{ref} \geq 0.95$
s_{s1}	0	0	0	0	1	$j = 0$
s_{s2}	0	0.9	-1	0	0	$j = 2$
s_{s3}	0	0.5	-0.75	0	0	$j = 2$
s_{s4}	0	0.2	-0.5	0	0	$j = 2$
s_{s5}	0	0.1	-0.5	0	0	$j = 2$
s_{s6}	0	1	0	-0.25	0	$j = 1$
s_{s7}	0	1	0	0.25	0	$j = 3$
s_{s8}	0	0.5	0	-0.25	0	$j = 1$
s_{s9}	0	0.5	0	0.25	0	$j = 3$
s_{s10}	0.3	1.3	0.1	0	0	$j = 1$
s_{s11}	-0.3	1.3	-0.1	0	0	$j = 3$

Table B.2: Constrained states in Lunar Lander-v2

References

[1] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. Deep reinforcement learning: A brief survey.

IEEE Signal Processing Magazine, 34(6):26–38, 2017.

[2] Andrew G Barto, Richard S Sutton, and Charles W Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE transactions on systems, man, and cybernetics*, 1(5):834–846, 1983.

[3] Felix Berkenkamp, Matteo Turchetta, Angela Schoellig, and Andreas Krause. Safe model-based reinforcement learning with stability guarantees. In *Advances in neural information processing systems*, pages 908–918, 2017.

[4] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. *JAX: composable transformations of Python+NumPy programs*, 2018.

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

[6] Ching-An Cheng, Xinyan Yan, and Byron Boots. Trajectory-wise control variates for variance reduction in policy gradient methods. In *Conference on Robot Learning*, pages 1379–1394. PMLR, 2020.

[7] Kamil Ciosek and Shimon Whiteson. Expected policy gradients for reinforcement learning. *arXiv preprint arXiv:1801.03326*, 2018.

[8] Marc Deisenroth and Carl E Rasmussen. Pilco: A model-based and data-efficient approach to policy search. In *Proceedings of the 28th International Conference on machine learning (ICML-11)*, pages 465–472. Citeseer, 2011.

[9] Soham Gadgil, Yunfeng Xin, and Chengzhe Xu. Solving the lunar lander problem under uncertainty using reinforcement learning. *arXiv preprint arXiv:2011.11850*, 2020.

[10] Javier Garcıa and Fernando Fernández. A comprehensive survey on safe reinforcement learning. *Journal of Machine Learning Research*, 16(1):1437–1480, 2015.

[11] Imene R Goumiri, Benjamin W Priest, and Michael D Schneider. Reinforcement learning via gaussian processes with neural network dual kernels. In *2020 IEEE Conference on Games (CoG)*, pages 1–8. IEEE, 2020.

[12] Azzam Haidar, Stanimire Tomov, Jack Dongarra, and Nicholas J Higham. Harnessing gpu tensor cores for fast fp16 arithmetic to speed up mixed-precision iterative refinement solvers. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 603–613. IEEE, 2018.

[13] Fei Han, Qing-Hua Ling, and De-Shuang Huang. Modified constrained learning algorithms incorporating additional functional constraints into neural networks. *Information Sciences*, 178(3):907–919, 2008.

[14] Arthur Jacot, Franck Gabriel, and Clément Hongler. Neural tangent kernel: Convergence and generalization in neural networks. *arXiv preprint arXiv:1806.07572*, 2018.

[15] Sham M Kakade. A natural policy gradient. *Advances in neural information processing systems*, 14, 2001.

[16] Swagat Kumar. Balancing a cartpole system with reinforcement learning–a tutorial. *arXiv preprint arXiv:2006.04938*, 2020.

[17] Jun Morimoto and Kenji Doya. Robust reinforcement learning. *Neural computation*, 17(2):335–359, 2005.

[18] Roman Novak, Lechao Xiao, Jiri Hron, Jaehoon Lee, Alexander A Alemi, Jascha Sohl-Dickstein, and Samuel S Schoenholz. Neural tangents: Fast and easy infinite neural networks in python. *arXiv preprint arXiv:1912.02803*, 2019.

[19] OpenAI. Openai gym leaderboard. https://github.com/openai/gym/wiki/Leaderboard, July 2021.

[20] Neal Parikh and Stephen Boyd. Proximal algorithms. *Foundations and Trends in optimization*, 1(3):127–239, 2014.

[21] Lerrel Pinto, James Davidson, Rahul Sukthankar, and Abhinav Gupta. Robust adversarial reinforcement learning. In *International Conference on Machine Learning*, pages 2817–2826. PMLR, 2017.

[22] Carl Edward Rasmussen, Malte Kuss, et al. Gaussian processes in reinforcement learning. In *NIPS*, volume 4, page 1. Citeseer, 2003.

[23] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *International conference on machine learning*, pages 387–395. PMLR, 2014.

[24] Yanan Sui, Vincent Zhuang, Joel W Burdick, and Yisong Yue. Stagewise safe bayesian optimization with gaussian processes. *arXiv preprint arXiv:1806.07555*, 2018.

[25] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

[26] Csaba Szepesvári. Algorithms for reinforcement learning. *Synthesis lectures on artificial intelligence and machine learning*, 4(1):1–103, 2010.

[27] Chen Tessler, Daniel J Mankowitz, and Shie Mannor. Reward constrained policy optimization. *arXiv preprint arXiv:1805.11074*, 2018.

[28] Matteo Turchetta, Andrey Kolobov, Shital Shah, Andreas Krause, and Alekh Agarwal. Safe reinforcement learning via curriculum induction. *Advances in Neural Information Processing Systems*, 33, 2020.

[29] Eiji Uchibe and Kenji Doya. Constrained reinforcement learning from intrinsic and extrinsic rewards. In *2007 IEEE 6th International Conference on Development and Learning*, pages 163–168. IEEE, 2007.

[30] Haoran Wang, Thaleia Zariphopoulou, and Xun Yu Zhou. Exploration versus exploitation in reinforcement learning: a stochastic control approach. *Available at SSRN 3316387*, 2019.

[31] R Williams. A class of gradient-estimation algorithms for reinforcement learning in neural networks. In *Proceedings of the International Conference on Neural Networks*, pages II–601, 1987.

[32] Greg Yang and Hadi Salman. A fine-grained spectral perspective on neural networks. *arXiv preprint arXiv:1907.10599*, 2019.

[33] Tianbao Yang. Advancing non-convex and constrained learning: Challenges and opportunities. *AI Matters*, 5(3):29–39, 2019.

[34] Yinyu Ye and Edison Tse. An extension of karmarkar's projective algorithm for convex quadratic programming. *Mathematical programming*, 44(1):157–179, 1989.

[35] Kaiqing Zhang, Zhuoran Yang, and Tamer Başar. Multi-agent reinforcement learning: A selective overview of theories and algorithms. *Handbook of Reinforcement Learning and Control*, pages 321–384, 2021.

[36] Christoph Zimmer, Mona Meister, and Duy Nguyen-Tuong. Safe active learning for time-series modeling with gaussian processes. In *Advances in Neural Information Processing Systems*, pages 2730–2739, 2018.