Gradient Monitored Reinforcement Learning

Mohammed Sharafath Abdul Hameed, Gavneet Singh Chadha, Andreas Schwung, and Steven X. Ding

Abstract—This article presents a novel neural network training approach for faster convergence and better generalization abilities in deep reinforcement learning (RL). Particularly, we focus on the enhancement of training and evaluation performance in RL algorithms by systematically reducing gradient’s variance and, thereby, providing a more targeted learning process. The proposed method, which we term gradient monitoring (GM), is a method to steer the learning in the weight parameters of a neural network based on the dynamic development and feedback from the training process itself. We propose different variants of the GM method that we prove to increase the underlying performance of the model. One of the proposed variants, momentum with GM (M-WGM), allows for a continuous adjustment of the quantum of backpropagated gradients in the network based on certain learning parameters. We further enhance the method with the adaptive M-WGM (AM-WGM) method, which allows for automatic adjustment between focused learning of certain weights versus more dispersed learning depending on the feedback from the rewards collected. As a by-product, it also allows for automatic derivation of the required deep network sizes during training as the method automatically freezes trained weights. The method is applied to two discrete (real-world multirobot coordination problems and Atari games) and one continuous control task (MuJoCo) using advantage actor–critic (A2C) and proximal policy optimization (PPO), respectively. The results obtained particularly underline the applicability and performance improvements of the methods in terms of generalization capability.

Index Terms—Atari games, deep neural networks (DNNs), gradient monitoring (GM), MuJoCo, multirobot coordination, OpenAI GYM, reinforcement learning (RL).

I. INTRODUCTION

Research in deep reinforcement learning (RL) has seen tremendous progress in recent years with widespread success in various areas, including video games [1], board games [2], robotics [3], industrial assembly [4], continuous control tasks [5], and cooperative multiagent systems [6], among others. This rapid increase in interest in the research community can be particularly traced back to advances made in the training of deep neural network (DNN) in the last decade, as well as novel RL algorithms developed recently. Notable examples of the latter include value function-based methods, such as deep Q-networks [7], and policy gradient methods, such as deep deterministic policy gradient [5], advantage actor–critic (A2C) [8], trust region policy optimization [9], and proximal policy optimization (PPO) [10], to name a few. Also, additional training components have helped in improving RL capabilities, such as improved exploration strategies [11], intrinsic motivation [12], and curiosity-driven methods [13].

Revisiting the training of DNN, regularization and better optimization methods have played a crucial role in improving their generalization capabilities, where batch normalization [14], dropout [15], and weight decay [16] are the most prominent examples that have become a standard in supervised learning. Surprisingly, little attention has been paid to methods for improving the generalization capabilities of DNN during RL although this appears to be crucial in supervised and unsupervised learning tasks. Regardless, most of the abovementioned approaches are also utilized in RL although there are stark differences between supervised learning and RL. It must be noted, however, that the above methods, nevertheless, also assist in RL training [17]. Our goal, however, is to develop a principled optimization and training approach for RL, especially considering its dynamic learning process.

In the literature, generalization in RL is usually done by testing the trained agent’s performance on an unseen variation of the environment, usually performed by procedurally generating new environments [17]. We, however, want to improve the evaluation performance in the same environment rather than generating new and unseen environments for the agent. An introduction to the existing methods for generalization in RL is provided in Section II. As a related problem, the derivation of suitable network sizes for a particular RL problem is rarely addressed. In practice, the size, i.e., depth and width of the neural networks, is mainly adjusted by either random search or grid search methods [18]. The other recent methods usually tune other hyperparameters, such as learning rate, entropy cost, and intrinsic reward, and do not consider the size of the network in RL [19]. Therefore, tuning for an optimal architecture requires knowledge on both the type of RL algorithm that is used and the application domain where the algorithms are applied, which inhibits the fast deployment of the learning agents. An automatic adjustment of the required network parameters is highly desirable because of the long training times in RL together with a large number of hyperparameters to be tuned.

To this end, we tackle the above-described weaknesses in current RL methods, namely, targeted training in the evolving learning setting and the automatic adjustment of the trainable parameters in the neural network. We present...
gradient monitored reinforcement learning (GM-RL), which maintains trust regions and reduces gradient variance, from the initial training phase in two of those algorithms, for targeted training. The original proposal for gradient monitoring (GM) with network pruning was originally introduced in [20] for supervised training of DNN. We enhance the previous work by concentrating on the accumulated gradients of weights and weights in the network, rather than just the weights. Specifically, rather than pruning irrelevant weights, we focus on the adaptive learning of the most relevant weights during the course of training. We develop different methods for GM-RL, starting with a method that requires knowledge of the learning process and then developing a momentum-based dynamic learning scheme that particularly suits the sequential learning process of RL. We further develop a method to automatically adjust the GM hyperparameters, particularly the active network capacity required for a certain task. It is important to note that the proposed methods are independent of the type of RL algorithm used and are, therefore, universally applicable. We apply and test the proposed methods in various continuous and discrete application domains. The proposed GM methods with the A2C [8] algorithm are tested on a multirobot manufacturing station where the goal is a coordinated operation scheduling of two industrial robots. Thereafter, we test the methods on some well-known RL environments from OpenAI Gym [21], such as the Atari games from the Arcade Learning Environment [22] and MuJoCo [23], both with the PPO [10] algorithm. The results obtained underline the improved generalization performance and the capability to automatically adjust the network size allowing for successful training also in strongly overparameterized neural networks.

The contributions of the work can be summarized as follows.

1) We introduce four novel GM methods, namely, Frozen threshold with Gradient Monitoring (F-WGM), Unfrozen threshold with Gradient Monitoring (U-WGM), Momentum with Gradient Monitoring (M-WGM), and Adaptive Momentum with Gradient Monitoring (AM-WGM), each successively increasing the performance of the RL algorithm.

2) The methods reduce the gradient norm helping in improving the smoothness of training during the initial phase of RL with the M-WGM and AM-WGM methods acting as a replacement for gradient clipping in PPO. In addition to the superior evaluation performance, the methods are shown to expedite the convergence speed by increasing the learning rates and increasing the “k-epoch” updates in PPO.

3) The proposed AM-WGM method allows for continuous adjustment of the network size by dynamically varying the number of active parameters during training, i.e., we adjust the network capacity depending on the rewards collected during the learning progress, thereby considerably increasing the robustness of the RL algorithm with respect to the applied network size.

4) We conduct various experiments on different application domains, including a coordination problem of a multirobot station, Atari games, and MuJoCo tasks, to underline the performance gains and the general applicability of the proposed methods.

This article is organized as follows. Related work is presented in Section II. In Section III, the basics of the RL-framework employed are introduced. The proposed GM methods and their integration with RL are presented in Section IV. Section V presents a thorough comparison of the results obtained from the proposed methods on the various application domains. Section VI concludes this article along with an outlook for future work.

II. RELATED WORK

We first discuss general approaches in DNN training that helps in better generalization capabilities and, subsequently, focus on methods specifically for generalization in RL. Finally, we discuss the approaches for tuning the network size in RL.

A. Generalization in DNN

Deep feedforward neural networks had been notoriously difficult to train in the past due to various factors, including vanishing gradient [24], highly nonconvex optimization problems [25], and the tendency to overfit [26]. All these shortcomings have been virtually mitigated in modern deep learning architectures through a myriad of techniques. They include initialization of the trainable parameters [27], [28], the use of sparse and nonsaturating activation functions, such as rectified linear function [29] in the hidden layers, and the use of more efficient stochastic gradient descent optimization algorithms, such as Adam [30]. The other approaches enhancing generalization capabilities are batch normalization [14] to overcome the internal covariate shift and dropout [15], which masks the neural activations with a masking matrix drawn from a Bernoulli distribution. For dropout, various variants improving on the vanilla dropout have been developed, including variational dropout [31] and targeted dropout [32]. Similarly, individual weights instead of hidden activation units are dropped in [33]. Recently, it has been investigated that overparameterization also leads to better generalization performance in supervised learning with DNN [34]–[36]. Another popular approach is the incorporation of auxiliary loss functions into the main loss resulting in either $L_1$ or $L_2$ regularization. An increasingly popular method for optimizing neural network training is gradient clipping [37], originally developed for the exploding gradient problem in recurrent neural networks. It has been proven to increase convergence speed in supervised learning in [38]. Also, a multitude of approaches for network pruning has been reported to help in the generalization performance. Generally, the pruning methods are applied iteratively based on the magnitude of weights [39], gradients, or Hessian [40], [41]. Recent methods, such as [42] and [43], calculate the sensitivity of each connection and prune the weights with a single-shot approach. Please refer [44] for a recent overview of the various pruning methods that have been developed for neural networks. We emphasize that our method does not include pruning weights but freezing them by not allowing the accumulated gradients of weights to flow to the respective weights. Also, a direct application of pruning methods in RL
is not apparent as these methods usually require retraining, which is far-fetched for the evolving dataset scenario during RL training. Indeed, all the above methods that have been used in RL, were specifically developed for supervised learning, but just found themselves to be used in RL.

B. Variance Reduction and Generalization in RL

Variance reduction techniques for gradient estimates in RL have been introduced in [45] where control variables are used for estimating performance gradients. An averaged deep Q network approach has been proposed in [46] where averaging previously learned Q-values estimates leads to a more stable training procedure. Also, variance reduction in the gradient estimate for policy gradient RL methods has been proposed in [47] with an input-dependent baseline that is a function of both the state and the entire future input sequence. Contrary to the previous approaches, we consider variance reduction in the gradient estimate by freezing the gradient update of a particular weight.

The literature on generalization in RL usually focuses on the performance of the trained agent in an unseen environment [17], [48]–[50]. However, better generalization methods for evaluating the agent in the same environment are missing in the literature. This is especially the case in industrial production environments where the production setup does not change drastically with time. The proposed method is focused on this area where a fast and reliable training procedure has been developed for discrete and continuous environments.

C. Neural Architecture Search

There are a number of hyperparameters in neural network training, with the size of the network being one of the most important ones. Apart from grid search and random search, there also exist a number of approaches, including Bayesian optimization [51], evolutionary methods [52], many-armed bandit [53], population-based training [19], and RL [54]. All the above methods search for neural architectures for a supervised learning setup in a sequential and very computationally expensive manner, while we consider RL problems and just adjust the active set of parameters during training and, hence, increase robustness with respect to the network size. In fact, we start with a possibly overparameterized network and increase or decrease the learning capacity during training to adjust the learning procedure. The adaptive GM method is partly inspired by Schaul et al. [55] where a multiarm bandit approach for adaptively adjusting a set of hyperparameters to optimize a proxy of the learning progress is proposed. In contrast, we propose a continuous adjustment of the hyperparameters based on a proxy, which can be interpreted as a continuous bandit approach.

III. INTRODUCTION TO REINFORCEMENT LEARNING

RL is the branch of machine learning that deals with training agents to take an action \( a \), as a response to the state of the environment at that particular time, \( s_t \), to get a notion of reward, \( r \), as shown in Fig. 1. The objective of the RL agent is to maximize the collection of this reward. Sutton and Barto [56] define RL as “learning what to do—how to map situations to actions—so as to maximize a numerical reward signal.”

A RL system has two major components: the agent and the environment where the overall system is characterized as a Markov decision process (MDP). The dynamics of the MDP are defined by the tuple \((S, A, P, R, P_0)\), with the set of states \( S \), the set of actions \( A \), a transition model \( P \), in which, for a given state \( s \) and action \( a \), there exists a transition probability to the next state \( s' \in S \), a reward function \( R: S \times A \times S \rightarrow \mathbb{R} \), which provides a reward for each state transition \( s_t \rightarrow s_{t+1} \) induced by action \( a_t \), and an initializtion probability \( P_0 \).

The policy \( \pi(a|s) \), provides an action \( a \in A \) for a state \( s \) presented by the environment. In value-based methods, a policy could use the state-value function, \( v(s) = \mathbb{E}[R_t|S_t = s] \), which is the expected return from the agent when starting from a state \( s \), or an state-action-value function, \( q(s,a) = \mathbb{E}[R_t|S_t = s, A_t = a] \), which is the expected return from the agent when starting from a state \( s \), while taking action \( a \). Here, \( R_t = \sum_{i=0}^{t} \gamma^i r_i \) is the discounted reward that the agent collects over \( t \) times steps and \( \gamma \) is the discount factor, where \( 0 \leq \gamma \leq 1 \). The policy then can be defined by an \( \epsilon - \)greedy strategy where the actions are chosen based on \( \pi(a|s) = \text{argmax}(q(s,A)) \) for a greedy action or a completely random action otherwise.

Alternatively, policy gradient methods use a parameterized policy, \( \pi(a|s, \theta) \), to take action without using the value functions at all. However, as shown by the policy gradient theorem, value functions may still be useful to improve the learning of the policy itself as seen in A2C [8]. The objective of the agent is to find an optimal policy, \( \pi^*(a|s) \), which collects the maximum reward. To find the optimal policy, the trainable parameters of the policy are updated in such a way that it seeks to maximize the performance as defined by the cost function \( J(\theta) \) as

\[
\theta_{t+1} = \theta_t + \rho \nabla J(\theta_t)
\]

where \( \theta \) is the parameter of the policy \( \pi \) and \( \rho \) is the learning rate. To derive at \( J(\theta_t) \), different algorithms have been developed as detailed next.

A. Advantage Actor–Critic

A2C is a synchronous variant of the A3C algorithm proposed in [8]. It uses the value function to reduce the variance in
the calculated cost function. Here, the actor refers to the policy $\pi(a|s, \theta_1)$, and the critic refers to the value function $v(s, \theta_2)$, where $\theta_1$ and $\theta_2$ are the parameters of the actor and critic, respectively. The parameters $\theta_1$ and $\theta_2$ are partially shared in case of the A2C algorithm, with $\theta_1, \theta_2 \subset \theta$. The cost function for the actor and the critic of the A2C algorithm is given by

$$\nabla J(\theta) = \mathbb{E}_{-\pi_\theta} \left[ \sum_{t=0}^{T-1} \nabla \log \pi_\theta(a_t, s_t) \cdot A_{\pi_\theta}(s_t, a_t) \right]$$  \tag{2}$$

where

$$A_{\pi_\theta}(s_t, a_t) = R - V_{\pi_\theta}(s_t)$$  \tag{3}$$

with

$$R = \mathbb{E}[r_{t+1} + \gamma V_{\pi_\theta}(s_{t+1})]$$  \tag{4}$$

and

$$V_{\pi_\theta}(s_{t+1}) = 0$$  \tag{5}$$

for the terminal state.

We use two cooperative agents that use indirect communication channels to solve the multirobot coordination environment.

### B. Proximal Policy Optimization

In this section, PPO is explained. In A2C, the focus of the algorithm was to get a good estimate of the gradients of the parameter $\theta$. However, applying multiple optimization steps on this empirically leads to large policy updates that destabilize the learning process. A surrogate objective function is used in this empirically leads to large policy updates that destabilize the parameter algorithm was to get a good estimate of the gradients of the parameter to overcome this by 

$$\max_{\theta} \mathbb{E}_{-\pi_\theta} \left[ \min(r_t(\theta) \hat{A}_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon) \hat{A}_t) \right]$$  \tag{6}$$

where

$$r_t(\theta) = \frac{\pi_\theta(s_t, a_t)}{\pi_{\theta_0}(s_t, a_t)}$$  \tag{7}$$

and $\hat{A}_t$ is the estimator of the advantage function at time step $t$. Refer to [10] for a full overview of the algorithm. Due to this controlled nature of the policy updates, PPO is found to work well with continuous control problems. PPO is used in MuJoCo and Atari learning environment.

All the experiments conducted here have a multilayered perceptron (MLP) as the function approximator. The MLP uses a “multiheaded” network architecture, where the policy network and the actor network have shared parameters, as shown in Fig. 2. The critic head provides the state-value function, while the policy head provides the actions.

### IV. REINFORCEMENT LEARNING WITH GRADIENT MONITORING

Modern deep learning architectures are, in general, over-parameterized, i.e., the parameters drastically outnumber the available dataset size [34], [57]. While this has been empirically shown to improve the training performance compared to shallow architectures, determining a suitable number of layers and neurons depending on the problem at hand remains to be an open issue. To circumvent the determination of network size, successful workarounds have focused on reducing the number of actively learning parameters per iteration. They end up reducing the degrees of freedom during the training using methods, such as dropout, drop connect, and their various subforms where network activations or weights are randomly switched OFF.

GM follows a different approach in which it intends to steer the learning process of the DNN by actively manipulating the backward pass of the training process. Specifically, the gradients in the backward pass are purposefully activated or deactivated, for a subset of weights, based on the learning conditions. Since Adam is used as the optimizer throughout this article, the accumulated gradients of weights will simply be called gradients henceforth. Although applicable to deep learning setting, we find GM particularly useful for RL since it reduces the variance in the gradient estimates during the crucial initial part of the learning process and also introduces a dynamic way to clip the gradients that are applied layerwise as opposed to the norm of the entire gradients popularly used.

### A. Gradient Monitoring in DNN

To illustrate the training procedure with GM, we consider fully connected feedforward DNN with more than one hidden layer trained with minibatch gradient descent and any arbitrary gradient-based optimizers although we found momentum-based gradient optimizers, such as Adam [30], to be most effective. However, we emphasize that GM is universally applicable to other network structures, such as convolutional or recurrent NN. The training procedure in NN minimizes the loss function by calculating the partial derivative of the loss functions with respect to each of the weight parameters recursively. Hence, for an NN model with $m \geq 2$ hidden layers, $W_m$ is denoted as the weight matrix for the $m$th layer, and $\nabla W_1, \nabla W_2, \ldots, \nabla W_m$ denote the gradients for each weight matrix. The gradient calculated as per the Adam
optimizer is given by
\[ \nabla L_W = \frac{\hat{m}_i}{\sqrt{\sigma^2 + \epsilon}}. \] (8)

To deactivate the gradients, elements of the gradient matrix \( \nabla L_W \), in (8) are set to zero. To accomplish this, we define a masking matrix \( M_t \), whose values are either one or zero, and calculate the new gradient matrix
\[ \nabla \hat{L}_W = M_t \circ \nabla L_W, \] (9)
where \( \circ \) denotes the Hadamard product. The weight update is performed then with a standard gradient descent update
\[ W_{t+1} = W_t + \rho \nabla \hat{L}_W. \] (10)

The steering of the learning process is decided based on the effect that each parameter has on the forward and the backward pass. Therefore, the masking matrix, \( M_t \), is calculated based on a function that takes as input, the weights \( W_t \), their respective gradients \( \nabla L_W \) from the backward pass, a learning threshold \( \mu (W_t, \nabla L_W) \), and a learning factor \( \lambda \). A decision matrix \( D_t(W_t, \nabla L_W) \) is constructed to estimate the learning process. This decision matrix \( D_t(W_t, \nabla L_W) \) is compared with the learning threshold \( \lambda \mu (W_t, \nabla L_W) \), in order to make the decision if the masking value is active, 1, or inactive, 0. The decision matrix can be calculated using lots of combinations like \( \vert \langle \nabla L_W / W_t \rangle \vert \), \( \langle W_t / \nabla L_W \rangle \), or \( \nabla L_W \circ W_t \). We use the absolute values since we are interested in the quantum of learning. To keep a concise representation, we denote \( D_t(W_t, \nabla L_W) \) and \( \mu (W_t, \nabla L_W) \) as just \( D_t \) and \( \mu \), respectively. Specifically, the masking matrix \( M_t \) can be defined as
\[ M_t = H(D_t - \lambda \mu) \] (11)
where \( H \) is the Heaviside step function in which the gradients of weight connections that do not reach the relative amount of learning are deactivated, i.e., receive no gradient during the backward pass. Note that, due to the use of Adam optimizer, the decision for freezing gradients is not only based on the actual gradient calculated over a minibatch but based on the decaying average of the previous gradients. We emphasize that GM is applied to each layer in the NN. The list of hyperparameters used along with their symbols and the method that they are used is given in Table I.

| Symbol | Description | VGM | M-WGM | AM-WGM |
|--------|-------------|-----|-------|--------|
| \( \lambda \) | Learning factor | ✓   | ✓    | ✓      |
| \( \eta_{\text{start}} \) | Start of GM   | ✓   |       |        |
| \( \tau_{\text{repeat}} \) | Mask update frequency |       | ✓    |       |
| \( \zeta \) | Masking momentum |       |       |        |
| \( M_t \) | Momentum matrix |       | ✓    | ✓      |
| \( \alpha_\lambda \) | change rate of \( \lambda \) |   |       |        |
| \( \phi \) | Reward collection rate |       |       |        |
| \( R \) | Rewards collected |       |       |        |

B. Vanilla Gradient Monitoring

The core of GM is the derivation of suitable conditions for activating and deactivating the gradients, \( \nabla L_W \), flow that includes deriving learning threshold \( \mu \) and factor \( \lambda \) based on the actual status of learning, and a schedule for freezing and unfreezing gradients. It is obvious that keeping a constant integer value as the learning threshold \( \mu \) for all the gradients is not appropriate as the proportion of learning represented in the gradients might have different distributions in different layers and different time steps. Furthermore, choosing a single constant learning value for different learning tasks is not trivial. Hence, the learning threshold is made adaptable by the use of functions, such as the mean or the percentile of the values of the decision matrix \( D_t \). This provides a method that ensures that a certain portion of the gradients is allowed in any situation. The Heaviside function, \( H \), is defined such that all gradients below the learning condition are deactivated.

In this article, the mean of all the elements \( d_{ij} \) in the decision matrix \( D_t \), as shown in (12), for each layer \( m \) is used as the \( \mu \) function, and \( \langle \nabla L_W / W_t \rangle \) is used as the decision function \( D_t \).
\[ \mu = \frac{1}{n} \sum_{ij} d_{ij}. \] (12)

The pseudocode for F-WGM and U-WGM that are together called Vanilla GM (VGM) is provided in Algorithm 1. For F-WGM and U-WGM, \( \eta_{\text{start}} \) has to be defined, which defines after which epoch the masking matrix is applied, along with the \( \lambda \) parameter, which is a multiplicative factor for \( \mu \). The only difference between F-WGM and U-WGM is that, in the case of F-WGM, \( \lambda \) is kept constant, and \( M_t \) is updated with the same \( \lambda \) value for every few iterations (\( \tau_{\text{repeat}} \)), while, in U-WGM, the \( \lambda \) is made variable, decreasing in value after every update. The rationale behind U-WGM is that the weight parameters that do not have a relatively high impact on the learning process during the initial phase of training (till epoch \( \eta_{\text{start}} \)) might, nevertheless, have an impact later, e.g., once other weights have settled. Hence, by reducing the learning condition threshold, those weights can participate again in the learning process. The factor \( \lambda \) is a hyperparameter that, in practice, we found that halving, i.e., \( \lambda = \lambda / 2 \) at every \( \tau_{\text{repeat}} \) works well.

Motivation for M-WGM and AM-WGM: Besides the question of which gradients to deactivate, we also have to answer the question: when to deactivate to make the training most effective? This problem is solved in two ways. First, as with the learning rate decay in SGD, similar schedules for deactivating are set up, depending on the problem at hand. The VGM methods use this setup. Alternatively, a momentum parameter is introduced on top of the masking matrix to alleviate the problem in deciding when to start deactivating the gradients. Therefore the M-WGM and AM-WGM methods are developed and are discussed in the following sections. Basically, \( \eta_{\text{start}} \) is a hyperparameter that needs to be tuned. However, some guidelines can be derived to automate the start of GM application. Particularly, the point of the first successful episode appears to be a reasonable choice as empirically this point was found to have the highest amount of gradients being backpropagated. Thus, creating and applying the first masking at this point make sense and perform better empirically.

C. Momentum With GM

One of the disadvantages of the previous methods is that the performance of the method is dependent on the hyperparameter \( \eta_{\text{start}} \), which is the first episode in which the agents
complete the task. This is difficult to ascertain in long-horizon episodic tasks, such as Atari games and MuJoCo, and it also requires hindsight knowledge on the quantum of gradients. Hence, a new version of GM-RL was developed to tackle the same called M-WGM. Here, a momentum matrix $M_\zeta$ and a momentum hyperparameter $\zeta$ are introduced, where the momentum matrix is applied to the gradients right from the first episode and the momentum hyperparameter provides intuitive control over the learning process. The pseudocode for M-WGM is given in Algorithm 2.

**Algorithm 2 M-WGM**

1. **Input:** $\nabla L_W$, $W_t$, $\rho$, $\lambda$, $\zeta$
2. **Init:** $M_\zeta$, $M_t$
3. **Sequence:**
4. **if** $\eta > = \eta_{\text{start}}$ **then** and every $\eta_{\text{repeat}}$
5. **for** each layer $m$ **do**
6. Decision matrix $D_t = \frac{\nabla L_W}{W_t}$
7. Learning factor $\mu = \text{Average}(\frac{\nabla L_W}{W_t})$
8. Masking matrix $M_t = H(D_t - \lambda \mu)$
9. Gradients: $\nabla \hat{L}_W = \nabla L_W \circ M_t$
10. **Output:** Weight $W_{t+1} = W_t + \rho \nabla \hat{L}_W$

D. Adaptive Momentum With GM

The $\lambda$ parameter in the M-WGM method is kept constant throughout the training. However, as noticed empirically in the U-WGM method, modifying the learning factor $\lambda$ improves the performance. Hence, in this section, AM-WGM is introduced, where, instead of manually setting the threshold for masking matrix activation, it is made adaptable based on the agent’s performance. To this end, we cast the continuous adjustment of $\lambda$ as a continuous bandit problem, where the parameter is adjusted based on a proxy of the learning performance. As a proxy, we define the reward collection rate $\phi_n$ as

$$\phi_n = \frac{R_n}{R_{n-1}}$$

where

$$R_n = \frac{1}{T} \sum_{i=(n-1)T+1}^{nT} r_i$$

is the reward collected during the $n$th episode and $T$ is the episode length. This criterion can be used for adjustment as follows. If the agent performs worse than before, i.e., $\phi_n/\phi_o < 1$, $\lambda$ is increased such that fewer gradients are active in the masking matrix. Contrary, if $\phi_n/\phi_o \geq 1$, $\lambda$ is reduced, which allows more gradients to be used. Consequently, if the performance is low, learning is restricted to fewer weights, while, in the case of improved performance, the agent is provided more weights in the learning process. The hyperparameter $\alpha_\lambda$ controls the amount of change in the $\lambda$ value. The pseudocode for the AM-WGM is provided in Algorithm 3. To ensure the stability in the initial training episodes, $\lambda$ is not modified until a few episodes are completed, usually at about 30% of the total episodes, and it is also updated after every few episodes, usually at about 10% of the total episodes. Hence, AM-WGM is similar to M-WGM in the initial stages of the learning process, and it only activates after a certain number of updates are applied and the learning process has stabilized. We emphasize, however, that the results are not sensitive to the above parameters such that $\eta_{\text{start}}$ and $\eta_{\text{repeat}}$ are much easier to tune than in the VGM. The adaptable nature of the method has empirically shown to increase the performance of the agents.

E. Theoretical Analysis

In this section, the theoretical basis for the faster convergence of the GM methods is discussed. To this end, we first conclude that all GM methods use a form of gradient manipulation (VGM) or a momentum-based method of gradient manipulation (M-WGM/AM-WGM). The gradient manipulation can be interpreted as a form of gradient clipping. While gradient clipping, as used in the literature, clips the gradients that are above a threshold, GM allows gradients above a certain threshold and clips/zeros gradients below it. Gradient clipping is a widely used method in natural language processing [58] and RL [10] to enable faster convergence by reducing the upper bound on the smoothness of the objective replacement for the global gradient clipping usually done in RL algorithms.
Algorithm 3 AM-WGM

1: Input: $\nabla L_W, W_t, \rho, \lambda, \zeta, a, \eta, \eta_{\text{start}}, \eta_{\text{repeat}}$
2: Init: $M_t, M_t, R_n, \phi_n, \phi_o, \eta_{\text{start}}, \eta_{\text{repeat}}$
3: Sequence:
4: if $\eta > \eta_{\text{start}}$ and every $\eta_{\text{repeat}}$ then
5: $\phi_n = R_n / R_o$
6: if $\phi_n / \phi_o \geq 1$ then
7: change = -1
8: else if $\phi_n / \phi_o < 1$ then
9: change = 1
10: $\lambda = \text{clamp}(\lambda + (a, \text{change}), 0, 1)$
11: $\phi_o = \phi_n$
12: $R_o = R_n$
13: for each layer do
14: Decision matrix $D_t = \frac{\nabla L_W}{W_t}$
15: Learning factor $\mu = \text{Average}(\frac{\nabla L_W}{W_0})$
16: Masking matrix $M_t = H(D_t - \lambda \mu)$
17: Momentum matrix $M_t = M_{t-1} \zeta + M_t(1 - \zeta)$
18: New Gradients: $\nabla L_W, = \nabla L_W \circ M_t$
19: Output: Weight $W_{t+1} = W_t + \rho \nabla L_W$

function [38]. Hence, by proving that GM, in fact, reduces the upper bound on the smoothness of the objective function, we are able to derive similar convergence proof. Traditionally, the smoothness of a function is given as follows. Let $f$ be the objective function to be minimized/maximized. $f$ is considered $L$-smooth if

$$||\nabla f(x) - \nabla f(y)|| \leq L ||x - y|| \text{ for all } x, y \in \mathbb{R}^d$$

(15)

where $L$ is the Lipschitz constant. Applying mean value theorem on (15), it becomes

$$||\nabla^2 f(x)|| \leq L \forall x \in \mathbb{R}^d$$

(16)

for all twice differentiable functions $f$. As shown in [38], this traditional definition has its limitations, where even simple functions break assumptions. Hence, a relaxed smoothness assumption is proposed in [38], where the local smoothness grows with the gradient norm. From [38], we refer to the following definition, assumption, and theorem.

Definition 1 ([38]): A second order differentiable function $f$ is $(L_0, L_1)$ smooth if

$$||\nabla^2 f(x)|| \leq L_0 + L_1||\nabla f(x)||.$$  

(17)

With this relaxed smoothness assumption, it has been proven that reducing the gradient norm reduces the upper bound on the iteration complexity of the gradient descent algorithms.

Assumption 1 ([38]): For an initialization $x_0$, the gradient norm is given by

$$G_n := \sup(||\nabla f(x)|| | x \text{ such that } f(x) \leq f(x_0)) < \infty.$$  

(18)

Theorem 1 ([38]): The iteration complexity of gradient descent algorithms, with $L_0 \geq 1, L_1 \geq 1$, and $G_n > 1$, and parameterized by fixed step size $h$, is lower bounded by

$$\frac{L_1 G_n(f(x_0) - f^* - 5\epsilon/8)}{8\epsilon^2(\log G_n + 1)}.$$  

(19)

It can be seen that, for a large $G_n$, the iteration complexity of WOGM will be higher than the GM methods if we can show that the GM methods reduce the gradient norm $||\nabla f(x)||$. This is formally stated in the following lemma.

Lemma 1: Denote $||\nabla f(x)\text{GM}||$ and $||\nabla f(x)\text{WOGM}||$ the gradient norms with GM applied and without GM (WOGM), respectively. Then, it holds

$$||\nabla f(x)\text{GM}|| < ||\nabla f(x)\text{WOGM}||.$$  

Proof: We begin by noting that the decision matrix $M_t$ is a binary matrix, with elements having value of either 0 or 1. Applying the Hadamard product between $M_t$ and $\nabla f(x)\text{WOGM}$ yields $\nabla f(x)\text{GM}$. Hence,

$$||\nabla f(x)\text{GM}|| = ||\nabla f(x)\text{WOGM} \circ M_t||$$

$$= \sqrt{\nabla f_0^T(x)\text{WOGM} \cdot M_0 + \nabla f_1^T(x)\text{WOGM} \cdot M_1}$$

$$< ||\nabla f(x)\text{WOGM}||$$

where $M_0$ and $M_1$ denote the vectors resulting by vectorizing the decision/momentum matrix and splitting it into vectors containing elements 0 and 1 only, respectively. $\nabla f_0(x)$ and $\nabla f_1(x)$ are the corresponding gradient vectors containing frozeed and unfreezed gradients, respectively. It is clear that the part $\sqrt{\nabla f_0^T(x)\text{WOGM} \cdot M_0}$ becomes zero as $M_0$ is an all zero vector. Hence, if GM is applied and at least one gradient is frozen, $||\nabla f(x)\text{GM}||$ is strictly lower than $||\nabla f(x)\text{WOGM}||$.

The same can also be extended to the momentum GM methods since the momentum matrix $M_t$ contains only elements in [0, 1], which, in turn, reduces the norm of the gradient matrix, when taking the Hadamard product. At an intuitive level, it could also be said that the GM methods provide an additional trust-region constraint for the policy updates and provide a variance reduction of the gradient updates. The time complexity (O) of the GM methods is the same as the WOGM since the additional operations are all Hadamard products or divisions.

V. EXPERIMENTAL RESULTS

We test the GM-RL methods on a variety of different environments to prove their applicability empirically and compare them with similar methods WOGM. We discuss and apply the proposed methods to a real-world multirobot coordination environment with discrete state and action space. Furthermore, we apply the two best-performing methods, M-WGM and AM-WGM, which operate without any hindsight information, on the OpenAI Gym environments of Atari games (continuous state space and discrete action space) and MuJoCo simulations (continuous state space and continuous action space). The results from the OpenAI Gym environments prove the general “plug and play” nature of the M-WGM and AM-WGM methods, wherein any previously proven RL algorithm can be improved upon by the usage of GM-RL. The results are
tested on various random seed initializations (Atari: five seeds and MuJoCo: ten seeds) to test for stability of the method. All the RL and proposed GM methods have been developed using PyTorch [59].

A. Multirobot Coordination Environment

This section describes the application of the GM-RL method on a cooperative, self-learning robotic manufacturing cell. The environment along with the corresponding RL agent setup is described in section below, followed by the results achieved on the various trials.

1) Learning Setup: The multirobot coordination problem is set up in a cooperative test bed. The test bed has a dual-robot setup, consisting of an Adept Cobra i600 SCARA robot and an ABB IRB1400 6-DOF-robot. The test bed has six handling stations: two input buffers, three handling stations, and one output buffer, as shown in Fig. 3. There are two different types of workpieces to be handled through the stations, Work-Piece-1 (WP1) and Work-Piece-2 (WP2), each with their respective input buffers. Both the workpieces have their own predefined sequences that are encoded through embedded RFID chips, as shown in Fig. 4. The schematic of the test bed is given in Fig. 3. The robots pick and place the workpieces from one station to the other. The work space where the handling stations are located is accessible to both the robots; hence, it is possible to have a shared working space denoted by the striped gray area in Fig. 3. Each robot has its own proprietary software to control its movements. Hence, a supervisory control system is implemented through a programmable logic control (PLC) that controls and coordinates the robot movements. This supervisory control system sends signals to the robot’s onboard control system where the actual action is taken by the robot. The task of the agents in this test bed is to move a predefined number of workpieces through the handling stations within optimal steps or time frames into the output buffer.

We use a simulation of the cooperative, self-learning robotic manufacturing cell to interact with the RL agent. Training is done in the simulated environment since training the RL agent on the real manufacturing cell is time-consuming, and it requires large amounts of data to converge to a good minima [60]. The simulated environment closely resembles the actual environment, emulating all the necessary features, such as the position of the workpiece and the status of machines, accelerating the learning process from taking weeks to few hours. The simulation environment is developed in Python.

An RL implementation on the real PLC system has also been accomplished [61].

Agent, Environment, and Reward Representations: In this part, the agent type, number of agents, the action space, and the state representation for the agents are discussed. The robots are used as the agents, where both robots act independently in a multiagent setup. The robots take the universal state of the system and output an action each. For the architecture of the robotic multiagent, independent learners with communication enabled between the agents were chosen since it is established that communication between the agents helps in faster convergence [62]. This setup gives the RL agent a good overview of the global and local conditions. Furthermore, Schwung et al. [61] have shown that independent learners outperform centralized approaches in such environments. For the action space, the agent has predefined movements of the workpiece paths implemented in the robot’s proprietary control system. The action space by extension controls the input, output, and loading of the workpiece in the resources. This eliminates ineffective actions, and the size of the action space is reduced to ten instances per robot. The state of the resources is chosen to represent the state space since it is independent of the work-order size and also had the computational advantage of being in a discrete space. The state space has 27 states given by 3^3, three work stations, and three job types (WP1, WP2, and empty). In addition, the workpiece completion percentage is also given as part of the state space. This acts as a communication channel between the agents to identify the work done by the other agent.

The rewards are set up as follows. Every action taken by the robots incurred a reward of −1. During the course of the study of the environment, two states were identified as being locked, meaning that no further movement of the job is possible. If the robot reached these states, it gets a reward of

| State                        | Reward |
|------------------------------|--------|
| Each step                    | -1     |
| Locked state                 | -100   |
| Incomplete after 1000 steps  | -100   |
| Unequal WP movement          | -30    |
| WP Output                    | +50    |
| Target achieved              | 500    |

| Fig. 3. Schematic of the multirobot setup with the common operation region of both robots shown in gray stripes. Here, IB1 and IB2 are the input buffers for WP1 and WP2, respectively. S1, S2, and S3 are the three work stations, and OB is the output buffer common for both WP1 and WP2. | Fig. 4. Multirobot setup of two industrial robots and a working platform. |
TABLE III

| Method         | Hyperparameter    | Value  |
|----------------|-------------------|--------|
| All            | NN input size     | 29     |
| All            | Body - network layers | 2     |
| All            | Body - layer neurons | 10    |
| All            | Head - network layers | 2     |
| All            | Head - layer neurons | 10    |
| All            | Batch size        | 10     |
| All            | Learning factor ($\lambda$) | 0.5    |
| All            | Discount factor ($\gamma$) | 0.99   |
| WOGM           | Learning rate ($\rho$) | 1e-3   |
| VGM, M-WGM     | Learning rate ($\rho$) | 2e-3   |
| AM-WGM         | Learning rate ($\rho$) | 5e-3   |
| M-WGM, AM-WGM  | Momentum value ($\zeta$) | 0.0005 |
| AM-WGM         | Repeat (t) | 1000 |
| AM-WGM         | Masking momentum ($\alpha_m$) | 0.999 |
| AM-WGM         | Threshold change ($\alpha_c$) | 0.001 |

Also, if the agent is not able to reach the required target in 1000 steps, it receives a reward of $-100$. To ensure that the equal quantities of WP1 and WP2 are processed, a constraint was placed on the system such that, if one of the workpiece reaches completion without the other workpiece reaching even 75% of its target, then it gets a reward of $-30$ as this behavior should be improved upon. Every individual output from the environment incurred a reward of $+50$, while the agent gets a reward of $+500$ if the global targets of both agents are achieved. The reward for the individual output can be characterized as an intermediate reward, which guides the agent to make more such actions that will eventually lead to achieving the global target. The global target is set as 20 workpieces each of WP1 and WP2. The rewards are shown in Table II.

We use a similar approach as presented [8] with neural network architecture in the actor–critic algorithm. The main idea is to use multitask learning [63], which constructs neural networks that enable generalized learning of the tasks, in the context of the actor–critic algorithm. The “multiheaded neural network” also is known to generalize the tasks by taking advantage of the system-specific information from the signals [63]. The primary hyperparameters in focus here are the network size, the learning rate, the batch size, and the n-step size. The values of hyperparameters that gave the best results are shown in Table III, which were set using grid search. Although the network size can be arbitrarily large, we use this particular size that gave the best result for the WOGM method. The activation function in the actor and critic layers is rectified linear function, while those used in the shared layer are sigmoids.

2) Results: We discuss the results in three subparts, namely, the gradients during the backpropagation, the number of rewards the agents collected, and the task time and the number of workpiece outputs achieved by each agent in the multirobot environment. All the results provided here are from the deployment of the agents after the training is complete although we also notice that the GM-RL methods improve the training performance, leading to faster convergence in all GM methods, as shown in Fig. 5. All the agents are trained for 5000 episodes where the convergence is achieved before 2000 episodes in each of the methods, allowing for 3000 further episodes of training. The agents are eventually tested for 4000 episodes to assess the consistency of their performance, as shown in Fig. 6.

a) Gradients: We start by discussing the number of gradients that backpropagate through the neural network to analyze the targeted learning activity in the network. Since the gradient directions can be both positive and negative, in order to get the actual quantum of the gradients, the absolute sum of the gradients for each backward pass is calculated. The absolute sum for the GM methods is calculated after the masking matrix is applied; hence, the quantum of gradients backpropagated by the GM methods is considerably less than the WOGM method, as can be seen in Fig. 7. It can be noted that, in the case of U-WGM, the gradients spike is noted in the iterations at which the masking matrix is applied. While the F-WGM and U-WGM are still prone to the odd fluctuations in the gradients backpropagated, it should be noted that the momentum-based GM methods (M-WGM and AM-WGM) control their gradient variations well during the entire training process. The WOGM training is exposed to extensive variation in the amount of gradient flow. This variance reduction eventually leads to a stable learning process, which reflects in the rewards collected, as illustrated in Fig. 6. The AM-WGM method collects the most rewards, followed by the other GM methods, with the WOGM method collecting the least amount of rewards.
Fig. 7. Comparison between the absolute sum of the gradient of the cost function, $J(\theta)$, with respect to all the weights $W$, in each update for the two agents in various GM methods. The shaded regions are the absolute sum of the gradients, while the dark lines show the moving average. It can be seen that the GM methods use a lower magnitude of the gradient for updating the weights in each iteration while also performing better than the WOGM method, as illustrated in Fig. 5.

Fig. 8. Comparison of the percentage of accumulated gradients allowed to backpropagate by the two agents in each of the methods. The shaded portions show the % values, while the dark lines show the moving average.

b) Robustness to choice of network size: Another important advantage of using the GM methods is the higher degree of freedom or robustness to the size of the network chosen. This is because the threshold function ($\lambda\mu$) explained in Algorithm 1 adaptively selects only the required neurons for learning and ensures that the learning is focused only on them. In Fig. 8, the dynamic selection of the number of active neurons from all the GM methods is illustrated over the training progress. This dynamic selection accelerates the learning process while removing the need for hyperparameter tuning for the number of neurons in the DNN. To provide additional evidence for this phenomenon, the same multirobot coordination problem was trained with a randomly chosen bigger network size (three layers and 20 neurons per layer) with the M-WGM method. Three simulations were made: one without any GM method, one with M-WGM (threshold: 0.5), and one with M-WGM (threshold: 0.75). As illustrated in Fig. 9, the rewards collected by the WOGM method with more parameters are considerably less than all the M-WGM methods when the network size is changed. The drop in performance is substantially less in the GM methods. Furthermore, Fig. 10 illustrates the drastic increase in the number of steps required for the WOGM method to achieve the workpiece transportation goal. This shows the robustness to the change in the size of the network. Fig. 11 illustrates the automatic adjustment on the number of active weights in the M-WGM methods. It can be observed that, for the same learning factor $\lambda$ of 50%, the quantum of gradients backpropagated in the smaller network is higher than in the bigger network, further proving the automatic usable network size adjustment capability of the method.

c) Task time: The task time is the number of steps required by the agents to move the 20 workpieces through the production system. Figs. 12 and 13 show the episode wise steps taken and jobs completed, respectively. WOGM performs the worst in terms of the number of steps required and is also not stable in work completion. F-WGM improves the task time while still being a bit unstable in workpiece completion. U-WGM provides a very stable output at the same time reducing the task time further. While M-WGM provides the best task completion time and is stable enough for deployment, AM-WGM provides the combination of stability and task time completion. This also reflects in the number...
Fig. 12. Performance comparison of various GM methods w.r.t. steps taken to complete the target of 40 workpieces. The baseline at 80 steps is the solution as solved by a human.

Fig. 13. Performance comparison of various GM methods w.r.t. number of workpieces completed in each episode during the evaluation phase.

of rewards collected. To analyze the statistical significance of the improvement made by the GM methods (M-WGM and AM-WGM), Welch’s t-test was performed on the number of workpieces output by each GM method compared to WOGM over its training phase. The improvement in the number of workpieces output was found to be statistically significant with p-values of <0.0001 and 0 for M-WGM and AM-WGM, respectively. The statistical analysis was not performed on the evaluation output data since all the methods provided a consistent output of 40 workpieces after training.

B. MuJoCo

1) Environment Description: The MuJoCo engine facilitates accurate and fast simulations of physical systems for research and development. This engine, wrapped inside the OpenAI Gym environment, provides a frequently used training and testing benchmark environment in the domain of RL. The already established baseline performance in the form of the PPO algorithm helps in the direct comparison of the effect of introducing GM methods. The methods M-WGM and AM-WGM are tested on four randomly selected MuJoCo environments, i.e., Half Cheetah, Ant, Humanoid, and inverted double pendulum (IDP). Each GM method was run on the four environments with ten random seed initializations. The training setup is the same as in [10], if not stated otherwise.

2) Results: Since the quantum of the gradients backpropagated is reduced, the hyperparameters of the PPO algorithm are modified to reflect that and take advantage of the reduced variances. For example, the learning rate is increased compared to the original paper. This does not destabilize the learning process like in the vanilla PPO implementation due to the inherent variance reduction capabilities of the M-WGM and AM-WGM methods. It should be noted that, for the GM methods, the global gradient clipping implemented in the vanilla PPO is not used. The GM implementation provides better layerwise control over the norm of the gradients. As illustrated in Fig. 14, during our trials, both the GM methods performed better than WOGM. The M-WGM and AM-WGM methods both performed on average better in all four games across the ten random seeds, while being statistically significant as shown in Table IV. It should be noted that the AM-WGM provides the best-normalized improvement over the WOGM. The final mean scores with their corresponding confidence intervals rounded off to their nearest integer over the ten random seeds are presented in Table V. The values of the hyperparameters for all the different methods are shown in Table VI. The p-values were calculated using the stats module of the python library SciPy.

C. Atari

1) Environment Description: The Atari games were first introduced in [22] to aid the development of general domain-independent AI technology. The environment also provides for a baseline where previous RL algorithms have
been tested. We test a total of ten games, six of which were randomly selected (Battlezone, Frostbite, Gopher, Kangaroo, Timepilot, and Zaxxon). The other four (Montezuma’s Revenge, Pitfall, Skiing, and Solaris) are selected to specifically test the long-term credit assignment problem of the algorithms. These four are considered difficult for PPO since the reward collection is low. The raw memory information is used as input to the network with no frames being skipped. The ten games were run on the three methods (WOGM, M-WGM, and AM-WGM) over five random seed initializations. The training setup is the same as in [10], if not stated otherwise. The values of the hyperparameters for all the different methods are shown in Table VII.

2) Results: As with the implementation in the MuJoCo environment, a higher learning rate is used, and the global clipping used in the vanilla PPO is not used. We also found that increasing the k-epoch update in AM-WGM increases its performance significantly. As shown in Fig. 15, the M-WGM method performs better than WOGM in four out of the six random games, while AM-WGM performs better in five out of the six random games. There was no performance improvement for the methods in the difficult games except in Solaris, where there is a drastic improvement made by the GM methods, as shown in Fig. 15. The mean scores along with their standard errors for 99.9% confidence intervals are also tabulated in Table VIII. The rewards collected in the evaluation phase of the Atari games are also found to be statistically significant compared to the WOGM for the games classified as easy, as shown in Table IX. The p-values of Montezuma and Pitfall are “nan” since the rewards collected were exactly the same. The rewards were only marginally different in skiing, which also reflects in the p-value of Welch’s t-test.

**VI. Conclusion**

We proposed novel neural network training methods called GM in RL, for more robust and faster training progress. The proposed methods incorporate a targeted training procedure in neural networks by reducing the upper bound of Lipschitz constant during training. We propose four novel GM methods, namely, F-WGM, UF-WGM, M-WGM, and AM-WGM. The F-WGM and UF-WGM methods consider point estimates of the gradients, whereas the other methods consider a momentum-based approach. The M-WGM method, therefore, controls the variance in the initial stages
stabilizing the training process. Furthermore, the AM-WGM method helps the network to choose the optimal number of parameters required for a particular training step based on the feedback from the rewards collected. This results in the training method being robust to the selection of the size of the network. The proposed methods outperform the standard A2C in the multirobot cooperation application and the standard PPO algorithm in all the MuJoCo environments and in most of the randomly selected Atari games. It also overcomes the long-term credit assignment problem in one of the difficult Atari games, Solaris. The “plug and play” nature of the GM methods means that improvements can be expected in any RL algorithm when gradient monitored RL is introduced.

A potential limitation of the F-WGM and UF-WGM methods is the occurrence of peaks in the gradients during training, which can, sometimes, disturb the training process. Another limitation of the AM-WGM is the selection of the hyperparameter \( f_{start} \). This can probably be eliminated by using feedback from the reward collection during training. This will be part of future work. Subsequent research will also focus on the performance improvement of the RL agent to generalize to unseen environment setup, such as in CoinRun [17], application to model-free on-policy RL algorithms, such as trust region policy optimization [9], and model-free off-policy RL algorithms, such as deep deterministic policy gradient [5].

REFERENCES

[1] G. Lample and D. S. Chaplot, “Playing FPS games with deep reinforcement learning,” in Proc. 31st AAAI Conf. Artif. Intell., 2017, pp. 2140–2146.
[2] D. Silver et al., “Mastering the game of go without human knowledge,” Nature, vol. 550, no. 7676, pp. 354–359, 2017.
[3] S. Gu, E. Holly, T. Lillicrap, and S. Levine, “Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates,” in Proc. IEEE Int. Conf. Robot. Automat. (ICRA), May 2017, pp. 3389–3396.
[4] T. Inoue, G. De Magistris, A. Munawar, T. Yokoya, and R. Tachibana, “Deep reinforcement learning for high precision assembly tasks,” in Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS), Sep. 2017, pp. 819–825.
[5] P. T. Lillicrap et al., “Continuous control with deep reinforcement learning,” in Proc. 4th Int. Conf. Learn. Represent., 2015, pp. 1–14.
[6] C. Sun, W. Liu, and L. Dong, “Reinforcement learning with task decomposition for cooperative multiagent systems,” IEEE Trans. Neural Netw. Learn. Syst., vol. 32, no. 5, pp. 2054–2065, May 2021.
[7] V. Mnih et al., “Human-level control through deep reinforcement learning,” Nature, vol. 518, pp. 529–533, Feb. 2015.
[8] V. Mnih et al., “Asynchronous methods for deep reinforcement learning,” in Proc. 33rd Int. Conf. Mach. Learn., vol. 48, Jun. 2016, pp. 1928–1937.
[9] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Abbeel, “Trust region policy optimization,” in Proc. Int. Conf. Mach. Learn., 2015, pp. 1889–1897.
[10] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” 2017, arXiv:1707.06347. [Online]. Available: https://arxiv.org/abs/1707.06347
[11] E. Conti, V. Madhavan, F. P. Such, J. Lehman, K. Stanley, and J. Clune, “Improving exploration in evolution strategies for deep reinforcement learning via a population of novelty-seeking agents,” in Proc. Adv. Neural Inf. Process. Syst., 2018, pp. 5027–5038.
[12] S. Mohamed and D. J. Rezende, “Variational information maximisation for intrinsically motivated reinforcement learning,” in Proc. Adv. Neural Inf. Process. Syst., 2015, pp. 2125–2133.
[13] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell, “Curiosity-driven exploration by self-supervised prediction,” in Proc. IEEE Comput. Vis. Pattern Recognit. Workshops (CVPRW), Jul. 2017, pp. 2778–2787.
[14] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” in Proc. 32nd Int. Conf. Mach. Learn., 2015, pp. 448–456.
[15] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A simple way to prevent neural networks from overfitting,” J. Mach. Learn. Res., vol. 15, no. 1, pp. 1929–1958, 2014.
[16] I. Goodfellow, Y. Bengio, and A. Courville, Deep Learning. Cambridge, MA, USA: MIT Press, 2016.
[17] K. Cobbe, O. Klimov, C. Hesse, T. Kim, and J. Schulman, “Quantifying generalization in reinforcement learning,” in Proc. Int. Conf. Mach. Learn., 2019, pp. 1282–1289.
[18] J. Bergstra and Y. Bengio, “Random search for hyper-parameter optimization,” J. Mach. Learn. Res., vol. 13, pp. 281–305, Feb. 2012.
[19] M. Eaderberg et al., “Population-based training of neural networks,” 2017, arXiv:1711.09946. [Online]. Available: https://arxiv.org/abs/1711.09946
[20] G. S. Chadha, E. Meydani, and A. Schwung, “Regularizing neural networks with gradient monitoring,” in Proc. INNS Big Data Deep Learn. Conf., 2019, pp. 196–205.
[21] G. Brockman et al., “OpenAI gym,” 2016, arXiv:1606.01540. [Online]. Available: https://github.com/openai/gym
[22] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, “The arcade learning environment: An evaluation platform for general agents,” J. Artif. Intell. Res., vol. 47, pp. 253–279, Jun. 2013.
[23] E. Todorov, T. Erez, and Y. Tassa, “MuJoCo: A physics engine for model-based control,” in Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst., Oct. 2012, pp. 5026–5033.
[24] S. Hochreiter, “The vanishing gradient problem during learning recurrent neural nets and problem solutions,” in Proc. 8th Int. Conf. Neural Netw. Learn. Statist., vol. 6, no. 2, pp. 107–116, 1998.
[25] M. Gori and T. Sai, “On the problem of local minima in backpropagation,” IEEE Trans. Pattern Anal. Mach. Intell., vol. 14, no. 1, pp. 76–86, Jan. 1992.
[26] S. Lawrence, C. L. Giles, and A. C. Tsoi, “Lessons in neural network training: Overfitting may be harder than expected,” in Proc. AAAI/IAAI, 1997, pp. 540–545.
[27] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” in Proc. 13th Int. Conf. Intell. Statist., 2010, pp. 249–256.
[28] K. He, X. Zhang, S. Ren, and J. Sun, “Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification,” in Proc. IEEE Int. Conf. Comput. Vis., Dec. 2015, pp. 1026–1034.
[29] X. Glorot, A. Bordes, and Y. Bengio, “Deep sparse rectifier neural networks,” in Proc. Aistats, vol. 15, 2011, p. 275.
[30] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” in Proc. 3rd Int. Conf. Learn. Represent. (ICLR), 2015, pp. 1–15.
[31] D. P. Kingma, T. Salimans, and M. Welling, “Variational dropout and the local reparameterization trick,” in Proc. Adv. Neural Inf. Process. Syst., 2015, pp. 2575–2583.
[32] A. N. Gomez et al., “Learning sparse networks using targeted dropout,” 2019, arXiv:1905.13678. [Online]. Available: https://arxiv.org/abs/1905.13678
[33] L. Wan, M. Zeiler, S. Zhang, Y. Le Cun, and R. Fergus, “Regularization of neural networks using DropConnect,” in Proc. Int. Conf. Mach. Learn., 2013, pp. 1058–1066.
[34] B. Neyshabur, Z. Li, S. Bhojanapalli, Y. LeCun, and N. Srebro, “On over-parametrization in generalization of neural networks,” in Proc. Int. Conf. Learn. Represent., 2019, pp. 1–20.
[35] M. Belkin, D. Hsu, S. Ma, and S. Mandal, “Reconciling modern machine-learning practice and the classical bias–variance trade-off,” Proc. Nat. Acad. Sci. USA, vol. 116, no. 32, pp. 15849–15854, Aug. 2019.
[36] A. Brutzkus and A. Globerson, “Why do larger models generalize better? A theoretical perspective via the XOR problem,” in Proc. Int. Conf. Mach. Learn., 2019, pp. 822–830.
[37] R. Pascanu, T. Mikolov, and Y. Bengio, “On the difficulty of training recurrent neural networks,” in Proc. Int. Conf. Mach. Learn., 2013, pp. 1310–1318.
[38] J. Zhang, T. He, S. Sra, and A. Jadbabaie, “Why gradient clipping accelerates training: A theoretical justification for adaptivity,” in Proc. Int. Conf. Learn. Represent., 2020, pp. 1–21.
[39] S. Han, J. Pool, J. Tran, and W. Dally, “Learning both weights and connections for efficient neural network,” in Proc. Adv. Neural Inf. Process. Syst., 2015, pp. 1135–1143.
T. Schaul
Y. Zhu
et al.
D. Schwung, F. Csaplar, A. Schwung, and S. X. Ding, “An application
L. Panait and S. Luke, “Cooperative multi-agent learning: The state of
Reinforcement Learning:
R. S. Sutton and A. G. Barto, “Introduction,” in
A. Paszke et al., “PyTorch: An imperative style, high-performance
deprecated deep learning library,” in Proc. Adv. Neural Inf. Process. Syst., 2019, pp. 1045–1048.
A. Zhai et al., “Target-driven visual navigation in indoor scenes using
deprecated deep reinforcement learning,” in Proc. IEEE Int. Conf. Robot. Automat. (ICRA), May 2017, pp. 3357–3364.
D. Schwung, F. Csaplar, A. Schwung, and S. X. Ding, “An application of
reinforcement learning algorithms to industrial multi-robot stations
for cooperative handing operation,” in Proc. IEEE 15th Int. Conf. Intell. Syst. (INDIN), Jul. 2017, pp. 194–199.
L. Panait and S. Luke, “Cooperative multi-agent learning: The state
of the art,” Auton. Agents Multi-Agent Syst., vol. 11, no. 3, pp. 387–434, Nov. 2005.
[40] B. Hassibi and D. G. Stork, “Second order derivatives for network
pruning: Optimal brain surgeon,” in Proc. Adv. Neural Inf. Process. Syst., 1993, pp. 164–171.
[41] Y. Le Cun, J. S. Denker, and S. A. Solla, “Optimal brain damage,” in Advances in Neural Information Processing Systems. San Francisco, CA, USA: Morgan Kaufmann, 1990, pp. 598–605.
[42] N. Lee, T. Ajanthan, and P. H. S. Torr, “SNIP: Single-shot network
pruning based on connection sensitivity,” in Proc. Int. Conf. Learn. Represent., 2019, pp. 1–15.
[43] N. Lee, T. Ajanthan, S. Gould, and P. H. S. Torr, “A signal propagation
perspective for pruning neural networks at initialization,” in Proc. Int. Conf. Learn. Represent., 2020, pp. 1–16.
[44] D. Blalock, J. J. Gonzalez Ortiz, J. Frankle, and J. Guttag, “What is the
state of neural network pruning?” in Proc. Mach. Learn. Syst., 2020, pp. 129–146.
[45] E. Greensmith, P. L. Bartlett, and J. Baxter, “Variance reduction tech-
niques for gradient estimates in reinforcement learning,” J. Mach. Learn. Res., vol. 5, pp. 1471–1530, Nov. 2004.
[46] O. Ansche1, N. Baram, and N. Shinkin, “Averaged-DQN: Variance
reduction and stabilization for deep reinforcement learning,” in Proc. 34th Int. Conf. Mach. Learn., 2017, pp. 176–185.
[47] H. Mao, S. B. Venkatakrishnan, M. Schwarzkopf, and M. Alizadeh,
“Variance reduction for reinforcement learning in input-driven environ-
ments,” in Proc. Int. Conf. Learn. Represent., 2019, pp. 1–20.
[48] N. Justesen, R. R. Torrado, P. Bontrager, A. Khalifa, J. Togelius, and
S. Risi, “Illuminating generalization in deep reinforcement learning
through procedural level generation,” 2018, arXiv:1806.10729. [Online]. Available: https://arxiv.org/abs/1806.10729
[49] X. Song, Y. Jiang, S. Tu, Y. Du, and B. Neysabour, “Observational over-
fitting in reinforcement learning,” in Proc. Int. Conf. Learn. Represent., 2020, pp. 1–30.
[50] M. Igl et al., “Generalization in reinforcement learning with selective
noise injection and information bottleneck,” in Proc. Adv. Neural Inf. Process. Syst., 2019, pp. 13956–13968.
[51] J. Bergstra, D. Yamas, and D. D. Cox, “Making a science of model
search: Hyperparameter optimization in hundreds of dimensions for
vision architectures,” in Proc. 30th Int. Conf. Int. Conf. Mach. Learn., vol. 28, 2013, pp. 115–123.
[52] S. R. Young, D. C. Rose, T. P. Karnowski, S.-H. Lim, and R. M. Patton,
“Optimizing deep learning hyper-parameters through an evolutionary
algorithm,” in Proc. Workshop Mach. Learn. High-Performance Comput. Environ., New York, NY, USA, Nov. 2015, pp. 1–5.
[53] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar,
“Hyperband: A novel bandit-based approach to hyperparameter opti-
mization,” J. Mach. Learn. Res., vol. 18, no. 1, pp. 6765–6816, 2017.
[54] B. Baker, O. Gupta, N. Naik, and R. Raskar, “Designing neural network
architectures using reinforcement learning,” in Proc. Int. Conf. Learn. Represent., 2017, pp. 1–18.
[55] T. Schaul et al., “Adapting behaviour for learning progress,” 2019,
arXiv:1912.06910. [Online]. Available: https://arxiv.org/abs/1912.06910
[56] R. Sutton and A. G. Barto, “Introduction,” in Reinforcement Learning: An Introduction. Cambridge, MA, USA: MIT Press, 2018, pp. 1–13.
[57] S. Arora, N. Cohen, and E. Hazan, “On the optimization of deep
networks: Implicit acceleration by overparameterization,” in Proc. 35th Int. Conf. Mach. Learn., vol. 80, 2018, pp. 244–253.
[58] T. Mikolov, M. Karafiát, L. Burget, J. Černocký, and S. Khudanpur,
“Recurrent neural network based language model,” in Proc. 11th Annu. Conf. Int. Speech Commun. Assoc. (INTERSPEECH), no. 9, 2010, pp. 1045–1048.
[59] A. Paszke et al., “PyTorch: An imperative style, high-performance
deprecated deep learning library,” in Proc. Adv. Neural Inf. Process. Syst., 2019, pp. 8024–8035.
[60] Y. Zhao et al., “Target-driven visual navigation in indoor scenes using
deprecated deep reinforcement learning,” in Proc. IEEE Int. Conf. Robot. Automat. (ICRA), May 2017, pp. 3357–3364.
[61] D. Schwung, F. Csaplar, A. Schwung, and S. X. Ding, “An application of
reinforcement learning algorithms to industrial multi-robot stations
for cooperative handing operation,” in Proc. IEEE 15th Int. Conf. Intell. Syst. (INDIN), Jul. 2017, pp. 194–199.
[62] R. Caruana, “Multitask learning,” Mach. Learn., vol. 28, no. 1, pp. 41–75, 1997.
[63] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, “Soft actor-critic: Off-
policy maximum entropy deep reinforcement learning with a stochastic
actor,” in Proc. Int. Conf. Mach. Learn., 2018, pp. 1861–1870.
[64] S. Fujimoto, H. Hoof, and D. Meger, “Addressing function approxima-
tion error in actor-critic methods,” in Proc. Int. Conf. Mach. Learn., 2018, pp. 1587–1596.

Mohammed Sharafath Abdul Hameed received the M.Sc. degree from the South Westphalia University of Applied Sciences, Soest, Germany, in 2019, where he is currently pursuing the Ph.D. degree. He is currently a Research Assistant with the Department of Automation Technology, South Westphalia University of Applied Sciences. His research interests include deep reinforcement learning, automation, production planning, and smart manufacturing.

Gavneet Singh Chaudha received the M.Sc. degree from the South Westphalia University of Applied Sciences, Soest, Germany, in 2016, where he is currently pursuing the Ph.D. degree. He is currently a Research Assistant with the Department of Automation Technology, South Westphalia University of Applied Sciences. His research interests include model-based control, networked automation systems, and intelligent data analytics with applications in the manufacturing and process industry.

Andreas Schwung received the Ph.D. degree in electrical engineering from the Technische Universität Darmstadt, Darmstadt, Germany, in 2011. From 2011 to 2015, he was a Research and Development Engineer with MAN Diesel & Turbo SE, Oberhausen, Germany. Since 2015, he has been a Professor of automation technology with the South Westphalia University of Applied Sciences, Soest, Germany. His research interests include model-based control, networked automation systems, and intelligent data analytics with applications in the manufacturing and process industry.

Steven X. Ding received the Ph.D. degree in electrical engineering from the Gerhard Mercator University of Duisburg, Duisburg, Germany, in 1992. From 1992 to 1994, he was a Research and Development Engineer with Rheinmetall GmbH, Munich, Germany. From 1995 to 2001, he was a Professor of control engineering with the University of Duisburg-Essen, Duisburg. His research interests are model-based and data-driven fault diagnosis and fault-tolerant systems and their applications in industry, with a focus on automotive systems and mechatronic and chemical processes.