Robust Deep Reinforcement Learning Scheduling via Weight Anchoring

Steffen Gracla, Edgar Beck, Carsten Bockelmann, and Armin Dekorsy

Abstract

Questions remain on the robustness of data-driven learning methods when crossing the gap from simulation to reality. We utilize weight anchoring, a method known from continual learning, to cultivate and fixate desired behavior in Neural Networks. Weight anchoring may be used to find a solution to a learning problem that is nearby the solution of another learning problem. Thereby, learning can be carried out in optimal environments without neglecting or unlearning desired behavior. We demonstrate this approach on the example of learning mixed QoS-efficient discrete resource scheduling with infrequent priority messages. Results show that this method provides performance comparable to the state of the art of augmenting a simulation environment, alongside significantly increased robustness and steerability.

Index Terms

Resource Management, Learning Systems, Robustness

I. Introduction

In the field of communication systems, deep Reinforcement Learning (RL) has stirred interest with its ability to learn approximately optimal strategies with limited model assumptions. This is an auspicious promise for challenges that are complex to model or to solve in real time, and early research has shown that deep RL strategies are indeed applicable to problems such as intelligent resource scheduling [1], [2]. Ultra-reliability, low latency, and heterogeneous QoS-constraints are cornerstones of future communication systems, particularly in fields such as medical communications, and flexible and fast learned schedulers may aid in achieving required performance goals. However, to answer whether these learned strategies can truly uplift modern communication systems, key questions regarding the *reliability* of these learned strategies in highly demanding scenarios must be addressed. Robustness and sample efficiency are among the central issues of Deep Learning (DL) that may cast doubt on its viability in the near future. For example, the commonly used deep deterministic policy gradient (DDPG) algorithm is known to have issues with sparse events [3], [4].

This work was partly funded by the German Ministry of Education and Research (BMBF) under grant 16KIS1028 (MOMENTUM).

All authors are with the Department of Communications Engineering, University of Bremen, Germany. Email: {gracla, beck, bockelmann, dekorsy}@ant.uni-bremen.de

This work was accepted for publication in IEEE Communications Letters.

These issues are commonly addressed by making a virtue of the sample-hunger of DL. Training data are often generated synthetically, so one may *augment* the simulation environment that generates the data by, e.g., increasing the occurrence of rare events like emergency signals [5] or splicing relevant examples into the training data [6]. While this approach is usable, highly parametrized training environments may require time-intensive tweaking to produce desired behavior on the learned algorithm. Worse, this approach widens the gap between simulation environment and reality, which is known to impede Machine Learning [7]. Finally, this approach does not inherently prevent a phenomenon known as “catastrophic forgetting” [8], where already learned behavior may be overwritten if examples are not encountered frequently.

In this paper, we regard the problem of discrete resource scheduling with rare but important URLLC priority messages. We make use of generic deep RL algorithms, but propose to cast the handling of priority signals and the handling of normal scheduling as two distinct tasks. By doing this, we enable the use of a method from the continual learning domain, *weight anchoring*, as discussed in [8]. Motivated by information theory, this method uses the Fisher information to focus the optimization landscape on solutions in the vicinity of the anchored solution via an elastic penalty. Hence, we design a two-stage learning process. First, our approach learns to handle priority messages exclusively. Next, we apply weight anchoring to this solution, and learn to optimize for overall system performance.

Such a split-task approach promises certain procedural advantages: Both priority handling and normal scheduling can be learned without distraction on environments that offer a low reality gap for optimal, sample-efficient learning. A single scaling factor controls the anchored tasks pull on the main optimization objective. Additionally, approximations of the Fisher information are readily available at no additional computation cost, as modern gradient descent optimizers already make use of it. In this work, we compare the commonly used “augmented simulation” approach with our two-stage process. We show comparable results in both overall system performance and handling of priority messages. We further highlight that our process is affected significantly less by catastrophic forgetting.

II. Setup & Notations

In this section, we introduce the general setup of the scheduling task and its metrics. We give detail on the standard actor-critic deep RL algorithm that is used to find efficient allocation solutions, and the elastic weight anchoring which encodes priority task handling.

A. The Allocation Task

Many current medium access schemes, such as Orthogonal Frequency-Division Multiplexing (OFDM), assume division of the total available resource bandwidth into discrete blocks. Hence, we introduce a scenario as depicted in Fig. 1, where in each discrete time step $t \in \mathbb{N}$ a scheduler is tasked with assigning the limited number $U \in \mathbb{N}$ of discrete resource blocks to jobs $j \in \mathbb{N}$. Jobs j come in two types, *normal* and *priority*, to be explained subsequently, and have two attributes: 1) A request size $u_{j,\text{req},t} \in \mathbb{N}$ in resource blocks; 2) A delay counter $d_{j,t} \in \mathbb{N}$ in time steps t since the job j has been generated. Job generation occurs at the beginning of each time step t at a probability p_{job} for each of the $N \in \mathbb{N}$ connected users. When generated, a job j is assigned an initial

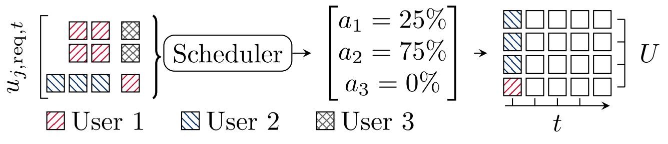

Fig. 1. The Resource allocation scenario tasks a scheduler with assigning a fraction of given discrete resources U to each user n at each time step t. It schedules based on the current state of a request pool, with request jobs of varying sizes and time-outs assigned to one particular user.

size $u_{j,\text{req},t} \leftarrow u_{n,\text{init},t} \sim \mathbb{U}\,[1,\, u_{\text{max}}]$ drawn from a discrete uniform distribution, and the delay is initialized as $d_{j,t} \leftarrow 0$. In a time step t, the scheduler outputs allocations $\mathbf{A}_t$ that distribute a ratio of the total resources U to each of the users n,

$$\mathbf{A}_t = [a_1, \quad a_2, \quad \ldots, \quad a_N]\,, \quad \text{with} \sum_{n=1}^{N} a_n = 1. \tag{1}$$

According to this allocation vector, the discrete blocks are then distributed to jobs currently in queue, starting from the oldest job j of each user n. The job's requested blocks $u_{j,\text{req},t}$ are decreased accordingly. Once all resources have been distributed following an allocation $\mathbf{A}_t$, the delay $d_{j,t}$ of all jobs j with requests remaining, i.e., $u_{j,\text{req},t} > 0$, is incremented. Jobs j with a delay $d_{j,t} > d_{\text{max}}$ greater than a maximum allowed delay d_{max} at the end of a time step t are removed from the queue, and a count $r_{d,t,\text{normal}}$ of timed-out jobs in that time step t is incremented. Further, at the beginning of each time step t there is a probability p_{prio} that one existing job j is designated *priority* status. If a priority job has not been fully scheduled within one time step t, it is removed from the job queue regardless of its delay $d_{j,t}$, incrementing a count $r_{d,t,\text{prio}}$ of timed-out priority jobs in time step t.

In addition to minimizing time-outs $r_{d,t,\text{normal}}$ and, in particular, $r_{d,t,\text{prio}}$, the scheduler will also be tasked with maximizing the sum-capacity of communication. The connection between base-station scheduler and users n is described by a Rayleigh-fading channel with a fading amplitude $|h_{n,t}| \sim \text{Rayleigh}(\sigma_{\text{R}})$ drawn from a Rayleigh distribution with scale σ_{R} in each time step t. Unlike in usual OFDM systems, we assume the power fading to be independent of the specific resource selected, to reduce computation complexity within the scope of this work. With $u_{n,\text{sx},t}$ being the resources scheduled to a user n in a time step t, the sum capacity under Gaussian code books calculates as

$$r_{c,t} = \sum_{n=1}^{N} u_{n,\text{sx},t} \cdot \log\left(1 + |h_{n,t}|^2 \frac{P}{\sigma^2_{\text{noise}}}\right), \tag{2}$$

where we assume the ratio of expected signal power P and expected noise power σ^2_{noise} as constant for all users. We aim for the scheduler to balance these three goals $r_{d,t,\text{normal}}$, $r_{d,t,\text{prio}}$, and $r_{c,t}$ with a weighting of our choosing. Therefore, we collect all metrics in a reward sum

$$r_t = w_c\, r_{c,t} - w_{d,\text{normal}}\, r_{d,t,\text{normal}} - w_{d,\text{prio}}\, r_{d,t,\text{prio}} \tag{3}$$

to be maximized, with tunable weights w_c, $w_{d,\text{normal}}$, $w_{d,\text{prio}}$ that control the relative importance of each term.

B. Deep Reinforcement Learning-Allocation

We implement a DDPG-based scheduler, as used in our previous work [2], to learn to output allocations $\mathbf{A}_t$ that approximately optimize the sum reward r_t in (3). The scheduler uses a pre-processor, two neural networks (*actor* and *critic*), a memory module, an exploration module, and a learning module. In the following we will briefly describe these components and the scheduler's process flow in training and inference. This scheduler has no explicit way of dealing with potential sparsity of priority job events.

We begin the design of our scheduler with a pre-processor that summarizes the information relevant to making allocation decisions into a vector of fixed size, palatable for the neural networks that are used subsequently. This state vector $\mathbf{S}_t$ holds four features per user n:

1) Resources requested in the set $\mathbb{J}_{n,t}$ of jobs assigned to user n at time step t, normalized by total resources U: $S_{n,1,t} = \sum_{j \in \mathbb{J}_{n,t}} u_{j,\text{req},t} \,/\, U$
2) Priority resources requested, normalized by total resources U: $S_{n,2,t} = \sum_{j \in \{\mathbb{J}_{n,t},\, j \text{ is prio}\}} u_{j,\text{req},t} \,/\, U$
3) Instantaneous channel power fading: $S_{n,3,t} = |h_{n,t}|^2$. We assume the power fading to be perfectly estimated; in a real system, an error or time delay may interfere.
4) Maximum delay, normalized by maximum allowed delay: $S_{n,4,t} = \max_{j \in \mathbb{J}_{n,t}} d_{j,t} \,/\, d_{\text{max}}$

The state is therefore directly influenced by the allocation action $\mathbf{A}_t$ taken by the scheduler in the previous time step t. Observed states $\mathbf{S}_t$ are saved in the memory module to be used by the learning module later, and fed into the *actor*-network $\boldsymbol{\mu}(\mathbf{S}_t, \boldsymbol{\theta}_{\mu,t})$. Based on the state and its current parameters $\boldsymbol{\theta}_{\mu,t}$, the actor outputs an allocation $\mathbf{A}_t$. In order to experience a wide variety of state-action combinations, an exploration module then introduces noise to the action $\mathbf{A}_t$ as follows. Using a momentum parameter ϵ_{expl} that is decayed to zero over the course of training, the actor allocation $\mathbf{A}_t$ is mixed with a normalized vector $\tilde{\mathbf{A}}_t$ of same length with entries drawn from a random uniform distribution $U(0,1)$, as $\mathbf{A}_t \leftarrow \epsilon_{\text{expl}} \tilde{\mathbf{A}}_t + (1 - \epsilon_{\text{expl}}) \mathbf{A}_t$. The noisy action is then re-normalized and propagated to the greater communication system, where success metrics are calculated according to Section II-A. Both noisy action and the resulting reward r_t are saved into the memory buffer alongside their state $\mathbf{S}_t$, and a new, updated state is forwarded to the pre-processor, restarting the loop.

In order to learn how to output good allocations from the experiences made, the learning algorithm must answer the question: What is a *good* mapping of state vector $\mathbf{S}_t$ to allocation $\mathbf{A}_t$? Standard Deep Deterministic Policy Gradient (DDPG) [9] algorithms decompose this question into two separate, fully connected neural networks with learnable parameters $\boldsymbol{\theta}_{\hat{Q},t}, \boldsymbol{\theta}_{\mu,t}$, respectively. As described previously, the actor-network handles the direct mapping of a state to an allocation. Meanwhile, the *critic*-network $\hat{Q}(\mathbf{S}_t, \mathbf{A}_t, \boldsymbol{\theta}_{\hat{Q},t}) = \hat{r}_t$ identifies what makes an allocation "good" by estimating the expected success r_t, i.e., (3), of performing an allocation $\mathbf{A}_t$ in state $\mathbf{S}_t$. It therefore approximates the unknown dynamics of the system that lead from allocation $\mathbf{A}_t$ to metric r_t, i.e., in this case: 1) the timeout mechanic; 2) the sum capacity formula in (2); and 3) the relative weightings w in (3). While in the learning phase, once per time step t, both networks learn from the data set of experiences collected in the memory module as follows. A mini-batch of experiences $(\mathbf{S}_t, \mathbf{A}_t, r_t)$ is sampled from the memory module and used to evaluate a loss function for critic and actor each. The critic loss provides the squared error between critic estimate $\hat{Q}(\mathbf{S}_t, \mathbf{A}_t, \boldsymbol{\theta}_{\hat{Q},t})$ and experienced reward r_t:

$$L_{\hat{Q},t} = \left(\hat{Q}(\mathbf{S}_t, \mathbf{A}_t, \boldsymbol{\theta}_{\hat{Q},t}) - r_t \right)^2 . \tag{4}$$

Thus, a lower loss $L_{\hat{Q},t}$ corresponds to a better reward estimate. Meanwhile, the *actor*-loss evaluates the critics reaction to the actors allocation:

$$L_{\mu,t} = -\hat{Q}\left(\mathbf{S}_t,\, \boldsymbol{\mu}(\mathbf{S}_t, \boldsymbol{\theta}_{\mu,t}),\, \boldsymbol{\theta}_{\hat{Q},t}\right). \tag{5}$$

A lower loss $L_{\mu,t}$ indicates greater estimated rewards. Each networks' parameters are then tuned using variants of stochastic gradient descent (SGD), leveraging the gradients $\nabla_{\boldsymbol{\theta}} L$ to minimize the respective loss functions. The actor's allocation strategy is therefore adjusted in a way that maximizes rewards $\hat{r}_t$, as estimated by the critic. These parameter updates can be performed decoupled from inference and sample gathering. In run-time inference, the process flow simplifies to only the pre-processor and the actor network.

C. Weight Anchoring

In our split-task approach, we are looking to first train the scheduler to deal with priority messages, then preserve this behavior as we learn the greater objective of QoS-efficient scheduling. A neural network's behavior is determined by its parameters $\boldsymbol{\theta}$. Thus, after the scheduler has learned to deal with priority messages to satisfaction in learning stage one, we follow [8] and record two factors for each parameter i: 1) The final values θ_i^{Anchor} describe a parametrized network that deals well with priority messages; 2) the Fisher information F_i^{Anchor} measures how sensitive the network behavior reacts to local changes on parameter i. For a more in depth analysis of Fisher information in the context of neural networks we refer to [10]. To save on computation cost, we extract an approximation of the Fisher information used by the SGD-optimizer. In our case, the Adam optimizer estimates the Fisher information as a moving average of the squared gradient [11],

$$F_{i,t} \leftarrow \frac{\beta_2 \cdot F_{i,t-1} + (1-\beta_2)\left(\nabla_{\boldsymbol{\theta}_\mu} L_{\mu,t}\right)^2}{(1-\beta_2)}, \tag{6}$$

with β_2 controlling the momentum of the rolling average in the optimizer.

Now, in learning stage two, we would like to learn QoS-efficient scheduling on the same network or any network of the same dimensions without straying from the critical message handling learned in stage one. To do this, we can use θ_i^{Anchor} and F_i^{Anchor} to construct a term $L_{\theta_i^{\text{Anchor}},t}$ that measures how far this network's current behavior, represented by its current parameter values $\theta_{i,t}$, has moved from the recorded behavior:

$$L_{\theta_i^{\text{Anchor}},t} = w_{\theta_i^{\text{Anchor}}} \sum_{i=1}^{I} F_i^{\text{Anchor}} \left(\theta_{i,t} - \theta_i^{\text{Anchor}}\right)^2. \tag{7}$$

This term grows proportional to the distance to the recorded parameter values θ_i^{Anchor}, weighted by their sensitivity F_i^{Anchor}, and is scaled by an anchoring weight $w_{\theta_i^{\text{Anchor}}}$.

By adding this term $L_{\theta_i^{\text{Anchor}},t}$ to the actor loss (5), we can incentivize the actor to learn a parametrization $\boldsymbol{\theta}_\mu$ that not only maximizes the approximated rewards, but also minimizes the distance to the recorded parameters θ_i^{Anchor}. The anchoring weight $w_{\theta_i^{\text{Anchor}}}$ controls how strongly the term (7) pulls the learning process to solutions in the vicinity.

III. Experiments

In an environment with priority messages whose frequency is controlled by a parameter p_{prio}, we present and compare the performance of different DL schedulers, both with and without anchoring. While the true baseline frequency of these priority messages is assumed to be rare ($p_{\text{prio}} = 0.01\,\%$), some schedulers will learn in an *augmented* environment with artificially increased priority message frequency. For our anchored schedulers, we also show the impact for different choices of anchoring weight $w_{\theta_i^{\text{Anchor}}}$.

A. Implementation Details

We implement the simulation using Python and the TensorFlow library using the Adam [11] optimizer. The full implementation code is provided in [12], and the system configuration during network training is listed in Table I. We assume the reward weights w and channel design parameters to be tuned by an expert. All schedulers' performance is evaluated on the baseline environment with rare ($p_{\text{prio}} = 0.01\,\%$) priority messages. In total, we train eight different DL schedulers:

1) The baseline scheduler (BS) is trained exclusively on the same environment that all schedulers will be evaluated on, i.e., priority messages appear infrequently ($p_{\text{prio}} = 0.01\,\%$).

2) Another scheduler (AU20) is trained on an augmented simulation environment where priority messages are encountered much more frequently ($p_{\text{prio}} = 20\,\%$).

3) For the first stage of our anchoring approach we train a scheduler (AU100) to deal well with priority messages ($p_{\text{prio}} = 100\,\%$). We later evaluate a snapshot of the scheduler at this point to highlight the performance gap.

4-6) For the second stage of our anchoring scheduler, we initialize three networks with the weights learned by scheduler no. 3, AU100. We then train these schedulers (AN1, AN2, AN3) on the baseline simulation ($p_{\text{prio}} = 0.01\,\%$), while anchored to AU100's weights as described in Section II-C. AN1, AN2 and AN3 differ in choice of the anchoring parameter $w_{\theta_i^{\text{Anchor}}}$, listed in Table I, to highlight its influence.

As our anchored schedulers are trained in two phases, with $30 \times 10\,000$ training steps each, we double the training episodes for the non-anchored benchmark scheduler AU20 to normalize training time. Further,

7-8) To highlight the effect of forgetting, we initialize two networks (AU20+, AN1+) with the parameters learned by no. 2, AU20, and no. 4, AN1, respectively. Both receive another stage of training on an environment with no ($p_{\text{prio}} = 0\,\%$) priority messages. Scheduler AN1+ remains anchored to the priority scheduling solution AU100.

B. Results

After training, all eight schedulers from Section III-A are frozen and evaluated on the setting with rare priority events ($p_{\text{prio}} = 0.01\,\%$). Testing the schedulers is carried out over 5 episodes at $200\,000$ steps per episode for an expected 10 priority events per episode. The process of training and testing is repeated three times for all schedulers, their average results and variance in overall performance and priority time-outs displayed in Fig. 2. All results are normalized to the performance of scheduler BS, trained exclusively on $p_{\text{prio}} = 0.01\,\%$, which serves as a baseline. Scheduler AU100, which was trained exclusively on priority events, highlights two important points: 1) It is

TABLE I
TRAINING CONFIGURATION

Steps t per episode	10 000	Episodes	30
SNR	10 dB	Rayleigh Scale σ_{R}	0.3
Resource Blocks U	10	Users N	5
Max RB u_{max}	7	Max Delay d_{max}	5
Job Probability p_{job}	50 %	Batch Size	256
F. Mom. β_2	default	NN Layers $\times$ Nodes	3 × 128
Weight w_c	$+1$	Weight $w_{\theta_i^{\text{Anchor}}}$	$1e\,\{5, 6, 7\}$
Weight $w_{d,\text{normal}}$	-1	Weight $w_{d,\text{prio}}$	-5
Init. ϵ_{expl}	1	Episodes $\epsilon_{\text{expl}} \to 0$	50 %

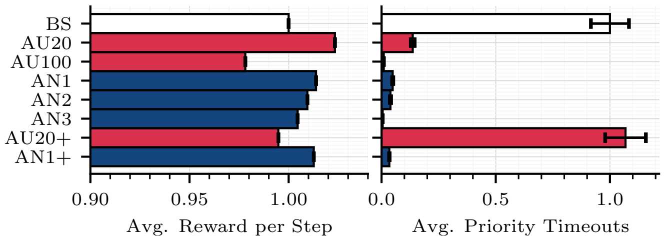

Fig. 2. Results for the eight schedulers introduced in Section III-A. We group schedulers by color, where the baseline scheduler BS is colored white, schedulers from augmented simulations are red, and anchored schedulers are blue. Results are normalized to the baseline scheduler, BS. For rewards, higher results are better, while for timeouts, lower results are better. Black bars represent the variance in results over all simulation runs.

possible to drastically reduce priority timeouts compared to the baseline BS; 2) There is a QoS-sum performance gap compared to baseline BS. We see that both the augmented scheduler AU20 as well as the anchoring approaches AN are able to fill or even exceed the reward performance gap, while still providing significantly better handling of priority messages. The degree to which these approaches trade-off reward performance and time-out performance is visibly influenced by the anchoring parameter $w_{\theta_i^{\text{Anchor}}}$ for the anchoring approaches, and by the choice of priority message frequency p_{prio} within the augmented simulation for the augmented scheduler. We expect these trade-offs to scale nonlinearly.

As mentioned in Section I, the anchoring approaches AN additionally offer certain desirable benefits over simply augmenting the simulation, as done in AU20, leading to better robustness, sample-efficiency, and a lesser model-gap. We highlight one of these benefits, the increased robustness against forgetting, in the results of schedulers AU20+, AN1+. They continue the training of AU20, AN1, respectively, without encountering any priority events. We find that our anchoring approach is able to retain performance and priority handling, while the comparison method AU20 “forgets” priority handling entirely.

IV. Conclusions

We present a way to transfer weight anchoring, a method from multi-task learning motivated by information theory, to robust deep RL in the presence of significant rare events. The handling of rare priority events is cast as a separate task. Its learnings are subsequently used as a basis to find a overall QoS optimal resource scheduling solution in the neighborhood of handling priority tasks. This method brings procedural advantages and is of particular interest in applications that suffer from high complexity, but require a highly deliberate, sample-efficient handling of rare critical events, which is traditionally a weakness of DL. We demonstrate this on a deep RL discrete resource scheduling task, though the method itself is not limited to this application.

REFERENCES

[1] C. Zhang, P. Patras, and H. Haddadi, "Deep Learning in Mobile and Wireless Networking: A Survey," *IEEE Commun. Surveys Tuts*, vol. 21, no. 3, pp. 2224–2287, 2019.

[2] S. Gracla, E. Beck, C. Bockelmann, and A. Dekorsy, "Learning Resource Scheduling with High Priority Users using Deep Deterministic Policy Gradients," *Proc. IEEE ICC2022*.

[3] G. Matheron, N. Perrin, and O. Sigaud, "The problem with DDPG: understanding failures in deterministic environments with sparse rewards," *arXiv:1911.11679*, 2019.

[4] S. Fujimoto, H. Hoof, and D. Meger, "Addressing Function Approximation Error in Actor-Critic Methods," in *Proc. ICML*, 2018, pp. 1587–1596.

[5] A. T. Z. Kasgari, W. Saad, M. Mozaffari, and H. V. Poor, "Experienced Deep Reinforcement Learning with Generative Adversarial Networks (GANs) for Model-Free Ultra Reliable Low Latency Communication," *IEEE Trans. Commun.*, vol. 69, no. 2, pp. 884–899, 2020.

[6] H. Chae, C. M. Kang, B. Kim, J. Kim, C. C. Chung, and J. W. Choi, "Autonomous Braking System via Deep Reinforcement Learning," in *Proc. IEEE ITSC*, 2017, pp. 1–6.

[7] L. Pinto, J. Davidson, R. Sukthankar, and A. Gupta, "Robust Adversarial Reinforcement Learning," in *Proc. ICML*, 2017, pp. 2817–2826.

[8] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska *et al.*, "Overcoming catastrophic forgetting in neural networks," *Proc. NAS*, vol. 114, no. 13, pp. 3521–3526, 2017.

[9] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv:1509.02971*, 2015.

[10] A. Achille, M. Rovere, and S. Soatto, "Critical Learning Periods in Deep Networks," in *Proc. ICLR*, 2019.

[11] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *arXiv:1412.6980*, 2014.

[12] S. Gracla, "Robust Scheduling via Weight Anchoring," https://github.com/Steffengra/DL_Lottery, 2021.