Towards Safe Online Reinforcement Learning in Computer Systems

Hongzi Mao, Malte Schwarzkopf, Hao He, Mohammad Alizadeh
MIT Computer Science and Artificial Intelligence Laboratory
{hongzi,malte,haohe,alizadeh}@csail.mit.edu

Abstract

Models trained using reinforcement learning (RL) are a powerful replacement for hand-written heuristics in many networked systems, but their potentially unpredictable behavior makes them unsafe for production deployment. We propose an approach to applying RL to networked systems that allows for safe deployment of models that rapidly adapt in the face of changing conditions. The key idea is to train the RL model online, in the real system, but to fall back on a simple, known-safe fallback policy if the system enters an unsafe region of the state space, while still providing sufficient feedback for RL to learn a good policy. Augmenting RL with such "training wheels" for safety helps a proof-of-concept request load balancer faced with sudden workload changes to rapidly adapt without violating response time objectives. Training wheels are effective, and they motivate research into general interfaces for developers to identify and specify unsafe states and fallback policies, as well as questions on how to integrate them deeper with RL methods.

1 Introduction

Reinforcement learning (RL) has recently emerged as a powerful technique to train control policies that outperform human-engineered heuristics for a wide range of tasks in computer systems and networking, such as routing [23], congestion control [10], video streaming [16], and cluster scheduling [15, 17]. Practical deployment of these RL policies, however, faces justified skepticism about their robustness when they face unusual situations. For example, an RL model for scheduling requests to servers may "misbehave" on a sudden workload change: it may make poor scheduling decisions that violate applications' service level objectives (SLOs), or worse, it may accumulate a request backlog that it can never clear.

Therefore, robust RL for control problems in computer systems and networks requires the ability to adapt a model's learned policy *in situ*, on a live system and in real time. However, online RL is unsafe in today's systems because RL fundamentally must explore and make mistakes in order to learn from them. While these necessary mistakes are no problem in an offline simulator, they are dangerous to a running system and may themselves inhibit further learning (e.g., by creating an insurmountable queue of requests). Our idea is to make online RL feasible by making these mistakes harmless. In particular, we constrain the RL model to a safe subset of possible behaviors while still allowing it to learn from its exploration.

As an analogy, consider how humans learn to ride a bicycle: there is a good chance your first bicycle had "training wheels", small auxiliary wheels attached to ensure that even if you lost balance, the bicycle did not fall over. The training wheels constrain how far the bicycle can lean — i.e., they constrain the possible states of the bicycle to those that are safe. Over time, the rider learns to keep their balance and the training wheels engage increasingly rarely.

To apply this concept to RL for computer systems, we observe that it is typically easy to identify a safe (but not necessarily performant) fallback policy in many systems. We rely on such a guaranteed-safe policy as our "training wheels" to recover if the RL policy takes the system into unsafe states. With this approach, the RL model never performs worse than the safe fallback policy, but as it trains, it can still learn data-dependent policies that improve over the fallback policy.

The challenge, however, is to ensure that despite engaging the training wheels the RL model still successfully adapts — i.e., that it ultimately finds a better new policy. This is difficult because the training wheels constrain the search space for exploration. The key insight that helps us overcome this challenge is to merge feedback from engaging training wheels into the reward function, and thus into the RL optimization objective. Our RL setting therefore optimizes for the original objective, but at the same time also learns to engage the training wheels as rarely as possible. Through a theoretical analysis, we show that the optimal policy under the training wheel setting translates to the same optimal policy for the original MDP.

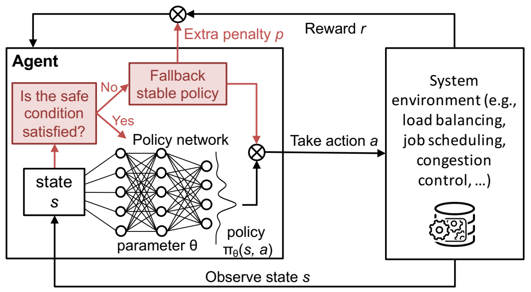

Extra penalty p
Reward r
Agent
Fallback stable policy
Is the safe condition satisfied?
No
Yes
Policy network
state s
parameter θ
policy $\pi_\theta(s, a)$
Take action a
System environment (e.g., load balancing, job scheduling, congestion control, ...)
Observe state s

Figure 1: Reinforcement learning with training wheels, new components highlighted in red. A stable backup policy guards the system against dangerous actions by the policy neural network during training and deployment.

We evaluate our training wheel technique with the case study of a web service request load balancer. To maintain low tail latency, the load balancer sends "hedged" backup requests to replicas after a timeout expires, as is common practice [5]. While RL can learn a policy that outperforms human-configured heuristics for a fixed workload, the model starts to perform poorly if workload characteristics change and deviate from the training workload. Augmenting with the training wheel technique enables the load balancer to adapt to changes in request distribution, cluster load, and server performance.

Our preliminary results illustrate the promise of RL with training wheels, but they raise several questions for future research. For example, generalizing the concept will require interfaces and tooling to specify safety conditions and fallback policies for a broad variety of systems, auto-tuning of key thresholds, and support for "off-policy" learning methods to improve sample efficiency.

2 Load Balancing: A Case Study

Consider the example of a request load balancer. A frontend application (e.g., a web search engine) generates requests to backend servers (e.g., caches, database servers), which replicate data. Each backend server processes requests from its queue. Request service times can vary across servers due to interference from other workloads or faulty hardware. The goal of the load balancer is to minimize the request response time, i.e., the time between initial request arrival and the earliest finish time of the (replicated) request from any server. The load balancer may base its server choice on the current queue length at each server; and when a request runs the risk violating the SLO, it may choose to resend the request to another server replicating the data. Such "hedged requests" after a timeout [5] alleviate head-of-line blocking and slow servers, at the expense of generating extra load on the backend.

The right server selection, timeout value and request replication factor depend on properties of the workload. A fixed policy for these choices can destabilize the system (i.e., cause request response time to grow without bound) when the workload changes. For example, when the system load is light, a sensible policy may set an aggressive timeout and send many request replicas in order to mitigate head-of-line blocking by slow requests. However, as load increases, aggressive replication overwhelms the backend's capacity. Therefore, workload changes require the load balancing policy to adapt online.

RL formulation. At each step, the load balancing agent observes the system *state*, which contains the queue length at each server and a one-hot vector that encodes which servers the request has already been sent. The agent then takes an *action* to choose a server and a timeout for the request. The action is two-dimensional and consist of $(a^j, a^l) \in \{1, 2, \cdots, n\} \times \{l_1, l_2, \cdots, l_k\}$, where n is the number of servers and k is the number of discrete timeout choices. This action assigns the request to server a^j and sets the timeout for sending a hedged request to a^l. The reward r_i at each step i is $-(t_i - t_{i-1})M_i$, where t_i is the wall time at step i, and M_i is the number of unfinished requests (across all replicas) in the system. Thus, the penalty amounts to the total request response time of all requests over a series of action.

3 Training Wheels Design

To introduce training wheels, we extend the standard RL setting with three components (Figure 1). The *safety condition* is a function over the state that indicates whether the system is in a safe or an unsafe state. If the safety condition is met, the agent relies on the agent's policy to take action. If the safety condition is violated, by contrast, the agent uses a guaranteed-safe *fallback policy* to choose the action. The fallback policy is designed to take the system back into a safe state to meet the safe condition in the future. Finally, activation of the fallback policy adds an *extra penalty* term into the agent's reward to encourage it to adapt its policy. In the following, we explain the implementation of these three components in the context of the load balancer and a theoretical analysis of this approach.

Safety condition. For the load balancer, the safety condition must ensure that the long-term request rate to the backend does not exceed the backend's service rate. Such a condition is difficult to directly measure and is unavailable as part of the state. However, a conservative proxy safety condition can measure the maximum length of server queues. If the request rate exceeds the service rate, server queues will grow indefinitely, triggering this safety condition. In practice, we further strengthen the safety condition and detect problems earlier by setting a threshold on the *longest* queue.

Fallback policy. A fallback policy must pick actions that move the system back into a safe state regardless of the (arbitrarily messed-up) state it inherits. Importantly, the fallback policy needs not be a strong policy with regards to the RL problem's objective function; it only needs to keep the system working and move it towards safety. For load balancing, the fallback policy should drain server queues back below the safety threshold to meet the system's request response time SLO. We implement the fallback policy that sends requests to the shortest queue, which is stable and monotonically improves safety: it guarantees to (eventually) drain the queues as long as the system is not overloaded.

Modified objective. To guide the RL agent to improve its policy, we add an extra penalty term when the fallback policy is active at each step. Specifically, the RL objective becomes

$$\underset{\pi}{\text{minimize}} \sum_{i=1}^{N} (t_i - t_{i-1}) M_i + \beta \sum_{i=1}^{N} (t_i - t_{i-1}) \mathbb{1}^{\pi}_{i,\text{fallback}}, \tag{1}$$

where the indicator function $\mathbb{1}^{\pi}_{i,\text{fallback}}$ is 1 when the fallback policy is active and β is the extra penalty factor. Here, the first term corresponds to the original objective and the second term applies the penalty. Intuitively, with a large enough β, the extra penalty steers the agent to avoid triggering the fallback policy. Adding the penalty term is necessary because the agent would otherwise learn to quickly violate the safety condition in order to activate the fallback policy, which leads to better rewards than the initial random policy in training. For neural networks, this traps the learned policy in a local minimum [9]. As the agent learns to avoid the fallback policy, the indicator returns zero, eliminating the extra penalty, and the remaining reward signal guides RL to optimize for the original request response time objective.

RL training. With these modifications, we focus on policy gradient methods [22] for RL training. To control the exploration for policy adaptation (e.g., when the workload shifts), we adjust the entropy factor [20]. Triggering the fallback policy should increase the entropy factor to randomize the policy network's actions. However, while increasing the entropy factor helps exploration, it must also decrease again to actually converge to an adapted policy. Our design uses the frequency of safety condition violations as a *control signal* for the entropy factor. The entropy factor increases if (*i*) the policy was previously stable, i.e., the safety condition has not been violated for a long time, and (*ii*) the safety condition is repeatedly violated over a short time. These conditions correlate with systematic environment changes, warranting exploration of different policies. Once the entropy factor has increased, we then slowly anneal it to converge the policy. Determining the right magnitude of increase for entropy factor is challenging, but might, e.g., rely on techniques from control theory (§6).

Theoretical Analysis. The training wheels essentially restrict the MDP into a smaller transition space. We analyze the optimal policy on the restricted MDP with a large penalty on violating the safe condition.

Definition 1 (State-constrained MDP)**.** *For an MDP* $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$ *and a set of "bad" states* $\mathcal{B} \subset \mathcal{S}$*, we define a new MDP* $\phi(\mathcal{M}, \mathcal{B}) = (\mathcal{S}, \mathcal{A}, \mathcal{P}', \mathcal{R}')$ *which is encouraged to never visit the states in* $\mathcal{B}$*. Formally,* $\mathcal{P}'(s,a,s') = \mathbb{1}_{[s=s']}, \forall s \in \mathcal{B}, a \in \mathcal{A}$ *and* $\mathcal{R}'(s,a,s') = -\infty, \forall s \in \mathcal{B}$*.*

Theorem 1. *Let* $\mathcal{M}$ *be an MDP with an optimal policy* π^**. Consider the constrained MDP* $\phi(\mathcal{M}, \mathcal{B})$ *and its optimal policy* $\tilde{\pi}$*. If* π^* *never visit states in* $\mathcal{B}$*, i.e.,* $\mathbb{P}_{\pi^*}(\exists t, s_t \in \mathcal{B}) = 0$*, then the constrained MDP has the same optimal policy as the original MDP's, i.e.* $\tilde{\pi} = \pi^*$*.*

The theorem states that as long as the optimal policy does not break the safety condition, the optimal policy obtained from the restricted MDP is also optimal in the original MDP Intuitively, this is because the trajectory is identical in the Bellman equation for any safe state of the two MDPs (taking the max in the values eliminates the path to unsafe regions). The uniqueness of Bellman equation solution [21, §3.5] then leads to the identical optimal policy. We refer the readers to the details in Appendix A.

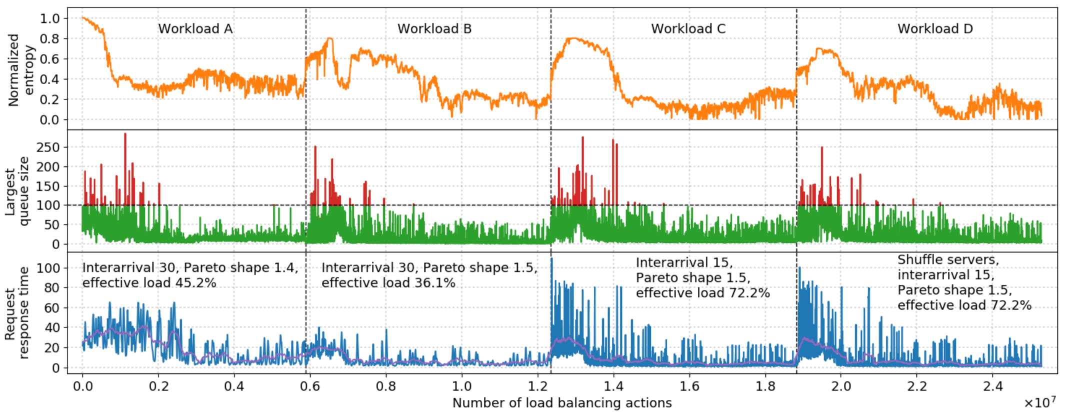


Figure 2: A load balancer agent with training wheels is robust to workload changes, which occur at each vertical dashed line on the time series. When the largest queue exceeds 100 requests (horizontal dashed line), the fallback policy engages and drains the overloaded queue (red spikes). With this safety guarantee, request response time remains below the 100ms and the agent gradually converges to a new policy (entropy decreases) for all workloads.

4 Experiments

We evaluate the effectiveness of the training wheels with a load balancing environment. The environment consists of ten heterogenous servers serving requests from a heavy-tailed Pareto distribution in size and Poisson process for inter-arrival time. The details of the setup are described in Appendix B. In this experiment, we change the workload three times while continuously training the RL agent with the training wheels. First, we change the Pareto skew of request durations from 1.4 to 1.5, reflecting an increase in small requests. Second, we double the request frequency (i.e., halve interarrival time), reflecting a load spike. Third, we shuffle the processing rates of the servers, reflecting changing interference from other workloads. All of these changes are sudden, unanticipated transitions that cause the workload-specific policy that the agent has learned to mismatch the environment. An agent without training wheels will misbehave on these changes and trigger system collapse (see Appendix C).

Figure 2 shows the request response time, maximum queue length, and entropy monitored at each action. At every workload change (vertical line), the agent's current policy quickly drives the system to violate the safety condition (red queue size in excess of 100), triggering the fallback policy and an entropy increase. The fallback policy bounds the response time while the queue is long, and the RL agent gradually adapts its policy to the new workload, and entropy drops again. Note that the fallback policy drains the queues fast enough that the maximum queue length is always below 300 to keep the response time below 100ms. Additionally, as time progresses without workload changes, the agent learns to further reduce the request response time (its primary reward signal).

Compare these results to offline training with a simulator: while it may be possible to train on changing workloads to obtain a robust policy, no offline training can ever cover *all* possible workload changes. Moreover, the policy learned by an agent trained offline over mixed workloads will be general across these workloads, falling short of ideal performance for *specific* workload settings. The RL agent with training wheels, however, is robust to workload changes because it adapts its policy online, and it can learn high-performance policies specific to the current workload (more details in Appendix D).

5 Related Work

A number of prior work has studied safety constraints in RL training [7]. The work closest to ours is "shielded" RL [2], which blocks out certain actions when a safety condition is violated. However, shielded RL only applies to simple problems with tabular states and treats the shield as part of the environment, without modifying the RL optimization objective. Therefore, the shield is only a protection mechanism to constrain the action set, not a technique for adapting the agent's policy. Xie et al. [24] proposed an "assisted" RL framework for robotics to inject experience data from expert's control into the replay buffer for off-policy RL methods such as DQN [19] or DDPG [14]. Unlike our training wheels that guarantee safety by design, this approach relies on the learned Q value as opposed to a strict safe condition to switch the control between the RL agent and the expert.

Another line of prior work focuses on adjusting RL algorithms to avoid exploring the unsafe states. This includes techniques such as bootstrapping from existing policy [1, 6], modeling the risk in state transitions [13], and conservative exploration [8]. These methods effectively incorporate the fallback policy within neural networks, but as these networks are unpredictable function approximators, they cannot guarantee that the system eventually recovers to a safe state. Training wheels, on the other hand, prioritize meeting the safety condition and rely on deterministic fallback policies whose properties and guarantees can be reasoned about.

6 Discussion and Open Questions

Training wheels are a general technique for online RL, but making them easy to use and apply to new settings will require further research. Many problems in computer systems and networks have clear safety conditions and heuristics that can serve as fallback policies (unlike, say, self-driving vehicles, which have no guaranteed-safe fallback policy). Practical online adaptation also requires a high action frequency to achieve rapid policy change (due to RL's sample complexity), and a strong reward signal that rapidly reflects the agent's actions. These criteria make RL with training wheels a good fit for low-latency and high-throughput settings like network packet routing [4] or request load balancing [5].

Expressing safety conditions and fallback policies. In general, RL with training wheels fits for the applications that resembles the dynamics of a queuing system [12]. For these applications, one may express the safety conditions through system measurements that are highly correlated with the performance metric (e.g., service throughput, request latency, system load, etc.). As long as the effect of an action is diminished (and eventually erased) over time, a work-conservative fallback policy can bring the system back to a safe state. More broadly, the safety condition can be indirectly expressed in terms of performance relative to a fixed heuristic. For example, a large enterprise optimizing video bitrate adaptation [16, 18] might run continuous A/B testing over video streams in the same network, helping highlight unsafe situations when the RL agent severely underperforms a heuristic.

We believe that it is an important research direction to develop a general framework that helps developers identify, specify, and validate safety conditions and fallback policies. This may include a safety condition language and an API that supports expressing a wide variety of fallback policies for different applications. The framework might ideally also support automated analyses to prove that a fallback policy monotonically moves the system towards safe states.

Automatic exploration. In our experiments, we hard-coded entropy factor increase and decrease rules to encourage exploration in the face of workload changes (§3). This scheme is unlikely to generalize to other applications, as the exact relation between exploration and safety violations (which the exploration may itself trigger) is often unknown. Applying ideas from control theory might help avoid brittle, hand-tuned thresholds and help automatically configure the entropy factor. The challenge is to balance the trade-off between agile adaptation (which requires a rapid entropy increase) and quick policy convergence (entropy annealing) using a function of, e.g., the rate of safety condition violations, the policy's past stability, and other factors.

Learning from the fallback policy. Standard policy gradient training methods are "on-policy" [21], meaning that bias-free training forces us to omit the data (i.e., experiences) collected from the fallback policy's actions. This can waste a substantial portion of training data, especially when the fallback policy is triggered frequently and remains active during workload changes. We intend to investigate combining off-policy RL training techniques such as apprenticeship learning [1] and importance sampling [11] with training wheels to use the fallback policy's experience without introducing bias. This would substantially increase sample efficiency.

7 Conclusion

The case study on the load balancing environment and the corresponding theoretical analysis show that RL controllers can learn strong policies under the constrained, safe setting. We believe that RL with training wheels is a promising direction that can yield robust, highly adaptive models that are safe to deploy into production systems and networks.

Acknowledgments. We thank the anonymous NeurIPS reviewers for their valuable feedback. This work was funded in part by the NSF grants CNS-1751009, CNS-1617702, a Google Faculty Research Award, an AWS Machine Learning Research Award, a Cisco Research Center Award, an Alfred P. Sloan Research Fellowship, a Microsoft TRAC Award and sponsors of the MIT DSAIL lab.

References

[1] P. Abbeel, A. Coates, and A. Y. Ng. “Autonomous helicopter aerobatics through apprenticeship learning”. In: *The International Journal of Robotics Research* 29.13 (2010), pp. 1608–1639.

[2] M. Alshiekh, R. Bloem, R. Ehlers, B. Könighofer, S. Niekum, and U. Topcu. “Safe Reinforcement Learning via Shielding”. In: *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI)*. New Orleans, Louisiana, USA, Feb. 2018.

[3] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[4] J. A. Boyan and M. L. Littman. “Packet routing in dynamically changing networks: A reinforcement learning approach”. In: *Advances in neural information processing systems* (1994).

[5] J. Dean and L. A. Barroso. “The Tail at Scale”. In: *Communications of the ACM* 56.2 (Feb. 2013), pp. 74–80.

[6] K. Driessens and S. Dvzeroski. “Integrating guidance into relational reinforcement learning”. In: *Machine Learning* 57.3 (2004), pp. 271–304.

[7] J. Garcıa and F. Fernández. “A comprehensive survey on safe reinforcement learning”. In: *Journal of Machine Learning Research* 16.1 (2015), pp. 1437–1480.

[8] C. Gehring and D. Precup. “Smart exploration in reinforcement learning using absolute temporal difference errors”. In: *Proceedings of the 2013 conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*. 2013, pp. 1037–1044.

[9] M. Gori and A. Tesi. “On the problem of local minima in backpropagation”. In: *IEEE Transactions on Pattern Analysis & Machine Intelligence* 1 (1992), pp. 76–86.

[10] N. Jay, N. Rotman, B. Godfrey, M. Schapira, and A. Tamar. “A Deep Reinforcement Learning Perspective on Internet Congestion Control”. In: *Proceedings of the 36th International Conference on Machine Learning (ICML)*. Ed. by K. Chaudhuri and R. Salakhutdinov. Long Beach, California, USA, June 2019, pp. 3050–3059.

[11] T. Jie and P. Abbeel. “On a connection between importance sampling and the likelihood ratio policy gradient”. In: *Advances in Neural Information Processing Systems*. 2010, pp. 1000–1008.

[12] L. Kleinrock. *Queueing systems, volume 2: Computer applications*. Vol. 66. wiley New York, 1976.

[13] E. L. Law, M. Coggan, D. Precup, and B. Ratitch. “Risk-directed Exploration in Reinforcement Learning”. In: *Planning and Learning in A Priori Unknown or Dynamic Domains* (2005), p. 97.

[14] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, et al. “Continuous control with deep reinforcement learning”. In: *arXiv preprint arXiv:1509.02971* (2015).

[15] H. Mao, M. Alizadeh, I. Menache, and S. Kandula. “Resource Management with Deep Reinforcement Learning”. In: *Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets)*. Atlanta, Georgia, USA, 2016.

[16] H. Mao, R. Netravali, and M. Alizadeh. “Neural Adaptive Video Streaming with Pensieve”. In: *Proceedings of the ACM SIGCOMM 2017 Conference (SIGCOMM)*. Aug. 2017.

[17] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh. “Learning Scheduling Algorithms for Data Processing Clusters”. In: *Proceedings of the 2019 ACM SIGCOMM Conference (SIGCOMM)*. Aug. 2019.

[18] H. Mao, S. Chen, D. Dimmery, S. Singh, D. Blaisdell, Y. Tian, M. Alizadeh, et al. “Real-world Video Adaptation with Reinforcement Learning”. In: *Proceedings of the 2019 Reinforcement Learning for Real Life Workshop*. 2019.

[19] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, et al. “Human-level control through deep reinforcement learning”. In: *Nature* 518 (2015), pp. 529–533.

[20] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Harley, T. P. Lillicrap, D. Silver, et al. “Asynchronous methods for deep reinforcement learning”. In: *Proceedings of the 32nd International Conference on Machine Learning (ICML)*. 2016, pp. 1928–1937.

[21] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction, Second Edition*. MIT Press, 2017.

[22] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour. “Policy gradient methods for reinforcement learning with function approximation.” In: *NIPS*. Vol. 99. 1999, pp. 1057–1063.

[23] A. Valadarsky, M. Schapira, D. Shahaf, and A. Tamar. “Learning to Route”. In: *Proceedings of the 16th ACM Workshop on Hot Topics in Networks (HotNets)*. Palo Alto, California, USA, 2017, pp. 185–191.

[24] L. Xie, S. Wang, S. Rosa, A. Markham, and N. Trigoni. “Learning with training wheels: speeding up training with a simple controller for deep reinforcement learning”. In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2018, pp. 6276–6283.

A Proof of Theorem 1

We first introduce an intermediate set of states that leads to the bad states $\mathcal{B}$ in Definition 1.

Definition 2 (Doomed states)**.** *For an MDP $\mathcal{M}=(\mathcal{S},\mathcal{A},\mathcal{P},\mathcal{R})$, we define the doomed states $\mathcal{D}$ corresponding to a given set of bad states $\mathcal{B}$ via $\mathcal{D}=\cup_{i=0}^{\infty}\mathcal{D}_i$ where $\mathcal{D}_{i+1}=\{s\in\mathcal{S}|\forall a\in\mathcal{A},\mathbb{P}(s_{t+1}\in\cup_{j\le i}\mathcal{D}_j|s_t=s,a_t=a)>0\}$ and $\mathcal{D}_0=\mathcal{B}$.*

By definition, the doomed states $\mathcal{D}$ for bad states $\mathcal{B}$ contain all the states that can not avoid visiting state in $\mathcal{B}$ no matter what actions are taken in the future steps.

Lemma 1. *Given a state-constrained MDP $\phi(\mathcal{M},\mathcal{B})$ and $\tilde{V}^{\pi}(s)$ as the value function for a policy π, we have $\tilde{V}^{\pi}(s)=-\infty,\forall s\in\mathcal{D}$, where $\mathcal{D}$ denotes the doomed states for $\mathcal{B}$.*

Proof. We prove the lemma by induction. First, by the definition of $\phi(\mathcal{M},\mathcal{B})$, we have $\tilde{V}^{\pi}(s)=-\infty,\forall s\in\mathcal{D}_0=\mathcal{B}$. Next, we assume $\tilde{V}^{\pi}(s)=-\infty,\forall s\in\cup_{t\le k}\mathcal{D}_t$. Consider a state s in $\mathcal{D}_{k+1}$, by definition, for any action a, the next state s' will be in $\cup_{t\le k}\mathcal{D}_t$ with non-zero probability, i.e. $\forall a\in\mathcal{A},\mathbb{P}(s'\in\cup_{j\le i}\mathcal{D}_j|s,a)>0$. Thus $\tilde{V}^{\pi}(s)=\mathbb{E}_{a\sim\pi}\left[\mathcal{R}(s,a)+\gamma\mathbb{E}_{s'}\tilde{V}^{\pi}(s')\right]=\mathbb{E}_{a\sim\pi}\left[\mathcal{R}(s,a)+\gamma\sum_{s'\notin\cup_{t\le k}\mathcal{D}_t}\mathcal{P}'(s,a,s')\tilde{V}^{\pi}(s')+\gamma\sum_{s'\in\cup_{t\le k}\mathcal{D}_t}\mathcal{P}'(s,a,s')\cdot(-\infty)\right]=-\infty$. Therefore, $\tilde{V}^{\pi}(s)=-\infty,\forall s\in\cup_{t\le k}\mathcal{D}_t$ implies that $\tilde{V}^{\pi}(s)=-\infty,\forall s\in\mathcal{D}_{k+1}$. $\square$

Lemma 2. *For an MDP $\mathcal{M}$ and a policy π, assume π never visits the bad states $\mathcal{B}$. Let $\phi(\mathcal{M},\mathcal{B})$ be the stated-constrained MDP induced by $\mathcal{B}$. Denoting V^{π} and $\tilde{V}^{\pi}$ as the value functions of the policy π for the two MDPs $\mathcal{M}$ and $\phi(\mathcal{M},\mathcal{B})$, we have $\tilde{V}^{\pi}(s)=V^{\pi}(s)$ for all $s\notin\mathcal{D}$ and $\tilde{V}^{\pi}(s)=-\infty$ for all $s\in\mathcal{D}$, where $\mathcal{D}$ is the doomed states for $\mathcal{B}$.*

Proof. We define $v(s)=V^{\pi}(s)$ for all $s\notin\mathcal{D}$ and $v(s)=-\infty$ for all $s\in\mathcal{D}$. It is equivalent to prove that $v(s)$ is the value function for the stated-constrained MDP. For $s\in\mathcal{D}$, we have $v(s)=-\infty=\sum_a\pi(s,a)(\mathcal{R}(s,a)-\infty)=\sum_a\pi(s,a)\left(\mathcal{R}(s,a)+\sum_{s'\in\mathcal{D}}\mathcal{P}(s,a,s')\cdot(-\infty)+\sum_{s'\notin\mathcal{D}}\mathcal{P}(s,a,s')v(s')\right)$. For $s\notin\mathcal{D}$, we have $v(s)=V^{\pi}(s)=\sum_a\pi(s,a)\left(\mathcal{R}(s,a)+\sum_{s'\notin\mathcal{D}}\mathcal{P}(s,a,s')V^{\pi}(s')\right)=\sum_a\pi(s,a)\left(\mathcal{R}(s,a)+\sum_{s'\notin\mathcal{D}}\mathcal{P}(s,a,s')v(s')\right)$. Thus, $v(s)$ matches the value function definition. $\square$

We now use the two lemmas above to prove theorem 1.

Proof of Theorem 1. Let V^{π^*} and $\tilde{V}^{\pi^*}$ be the value functions of the policy π^* for the two MDPs $\mathcal{M}$ and $\phi(\mathcal{M},\mathcal{B})$. We show that $\tilde{V}^{\pi^*}$ satisfies the Bellman equation to prove the optimality of π^* in the MDP $\phi(\mathcal{M},\mathcal{B})$. First, by lemma 1, we have $\tilde{V}^{\pi^*}(s)=-\infty$ for $s\in\mathcal{D}$. Also, $\mathbb{E}_{s'|s,a}\tilde{V}^{\pi^*}(s')=-\infty$ as a result of definition 2. Thus, $\tilde{V}^{\pi^*}(s)=-\infty=\max_a\mathcal{R}'(s,a)+\gamma\mathbb{E}_{s'|s,a}\tilde{V}^{\pi^*}(s')$. Second, for $s\notin\mathcal{D}$, we have $\tilde{V}^{\pi^*}(s)=V^{\pi^*}(s)$ by lemma 2. Since V^{π^*} satisfies the Bellman equation for MDP $\mathcal{M}$, we have $V^{\pi^*}(s)=\mathcal{R}(s,a^*)+\gamma\mathbb{E}_{s'|s,a^*}V^{\pi^*}(s')$ where $a^*\in\text{argmax}_a\mathcal{R}(s,a)+\gamma\mathbb{E}_{s'|s,a}V^{\pi^*}(s')$. Note that π^* never visits $\mathcal{D}$, taking action a^* at state s does not lead the next state in $\mathcal{D}$, i.e. $s'\notin\mathcal{D}$. Thus, $\tilde{V}^{\pi^*}(s)=V^{\pi^*}(s)=\mathcal{R}(s,a^*)+\gamma\mathbb{E}_{s'|s,a^*}V^{\pi^*}(s')=\mathcal{R}'(s,a^*)+\gamma\mathbb{E}_{s'|s,a^*}\tilde{V}^{\pi^*}(s')$. By the optimallity of a^*, $\forall a\in\mathcal{A},\mathcal{R}(s,a^*)+\gamma\mathbb{E}_{s'|s,a^*}V^{\pi^*}(s')\ge\mathcal{R}(s,a)+\gamma\mathbb{E}_{s'|s,a}V^{\pi^*}(s')$. Consider action $a\in\mathcal{A}$, if $\mathbb{P}(s'\in\mathcal{D}|s,a)=0$ then $V^{\pi^*}(s')=\tilde{V}^{\pi^*}(s')$, otherwise $V^{\pi^*}(s')>\tilde{V}^{\pi^*}(s')=-\infty$. Thus $\forall s\notin\mathcal{D},a\in\mathcal{A},\mathcal{R}'(s,a^*)+\gamma\mathbb{E}_{s'|s,a^*}\tilde{V}^{\pi^*}(s')=\mathcal{R}(s,a^*)+\gamma\mathbb{E}_{s'|s,a^*}V^{\pi^*}(s')\ge\mathcal{R}(s,a)+\gamma\mathbb{E}_{s'|s,a}V^{\pi^*}(s')\ge\mathcal{R}'(s,a)+\gamma\mathbb{E}_{s'|s,a}\tilde{V}^{\pi^*}(s')$, which shows $\tilde{V}^{\pi^*}$ satisfies the Bellman equation. $\square$

B Implementation Details

We follow the load balancer setup described in §2. We sample the size of the requests from a heavy-tailed Pareto distribution and model the request arrivals as a Poisson process. The workload, governed by the Pareto and Poisson parameters, changes during our experiments. We simulate ten heterogeneous servers with processing rates uniformly distributed from 0.15 to 1.05. Each server processes incoming requests in FIFO order. Our setup omits cancellation of replicated requests when the first request returns; this simplifies the setup and makes it more critical for the agent to hedge requests only when necessary.[1]

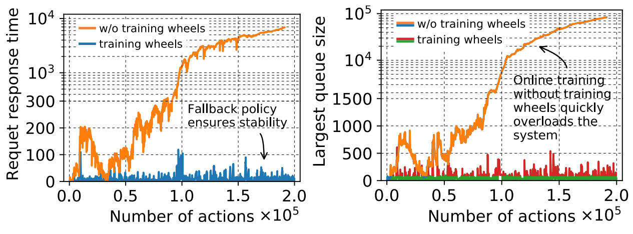
Agent setup. We map the state input to a two-dimensional discrete action of server index and timeout level via a fully-connected policy neural network. The network chooses one of seven discrete timeout levels. For simplicity, we assume the action in each dimension is independent from the other (with separate softmax [3] operations). The neural network has two hidden layers with 32 and 16 neurons, which supply 2,689 parameters. We batch update the policy after each 256 consecutive (state, action, reward) experiences.

Safety condition, fallback policy, and reward. We define the safety condition as $\max(Q_j) < 100$, where Q_j is the queue size of server j. The fallback policy sends the request to the server with the shortest queue without replicating it. Once the safety condition is violated, the fallback policy overwrites the policy network's actions until all queues are fully drained. The fallback policy is stable (i.e., can fully drain the queues) as long as the request load is below the aggregate server capacity. The penalty factor β for activating the fallback policy is 500. We increase exploration by setting the entropy factor increase rate to 1 (i.e., each safety violation increases the factor by 1) if the fallback policy was inactive for 500k actions. The entropy factor is capped at 10, and we reset the increase rate to zero when it hits this cap. To converge the RL policy, the entropy factor decreases by 10^{-4} per action.

Metrics. We measure the average request response time as the optimization objective. As shown in Figure 2, we also monitor the server queue lengths and the entropy of the policy network (normalized to a uniformly random policy) to observe the convergence behavior.

C Training wheels are necessary for online learning

This microbenchmark tests whether training wheels are necessary for successful online training of a load balancer. We train an agent without training wheels, which always samples its action from the output of the policy network, and compare it to an agent protected by the training wheels. Figure 3 shows the agent's performance for the first 1% of actions in the experiment from §4. As the agent explores initial policies and takes random actions, request response time varies and quickly grows indefinitely as the agent fails to compensate for its earlier mistakes (Figure 3a). The reason for this growth in response time is that the agent without training wheels over-replicates requests and makes poor assignment choices, which results in queues growing without bound (Figure 3b). The RL agent *with* training wheels, by contrast, achieves stable training by relying on the fallback policy and rarely exceeds the 100ms SLO. It still regularly violates the safety condition in these early training stages, but as the fallback policy brings the system back to a safe state, the agent can accumulate the experiences of good and bad actions that improve its policy.



(a) Request response time. **(b)** Maximum queue length.

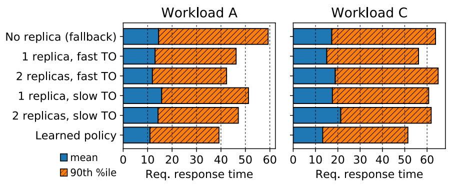
Figure 3: Without training wheels, the load balancer suffers performance collapse early in online training. The x-axis represents the first 1% of the experiment; the upper half of the y-axis is a $\log_{10}$ scale. Red in (b) indicates violations of the safety condition, which are common early in training.

D Policy Adaptation in Changing Environments

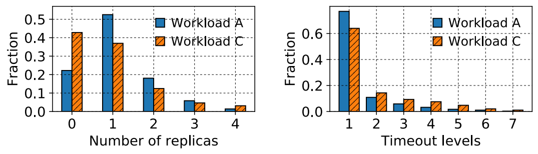
Under the training wheels constraints, we check that the load balancing RL agent can still learn a strong policy. We compare its learned policy after adaptation to the performance of the fallback policy (i.e., no hedged requests), as well as several heuristics with fixed timeouts and replication factors. The fixed timeouts are the two extreme values available to the agent (5 and 50ms).

Figure 4a shows the results for the first and third workloads from Figure 2. The fallback policy performs poorly in the 90th percentile for both workloads as it does not send any hedged requests. No fixed heuristic for request replication works well across both workloads either: for example, two-way

[1]Even with cancellation, hedged requests at high load can cause queues to grow without bound and policy adaptation is needed.



(a) The agent learns workload-specific policies that outperform both the fallback policy (no replication) and fixed heuristics. Blue: average response time; orange: 90th percentile.



(b) RL with training wheels learns to adapt to different workloads. With higher load in workload C, the agent replicates requests less aggressively than on workload A, and it sets more conservative timeouts.

Figure 4: Comparing the learned policy with existing heuristics under different workloads.

replication with a low timeout achieves quick responses with workload A, but performs poorly at higher load in workload C. The RL agent's adapted, workload-specific policies perform well for both workloads, reducing the mean and 90th percentile request response time by 9.8% over the best heuristic for workload A and by 11.5% for workload C. Moreover, as shown in Figure 4b, the learned load balancer exhibits qualitatively different policies under different workloads. This shows that RL agent with training wheels can learn strong, workload-specific policies that outperform the fixed heuristics even when relying on a simple fallback policy with low performance for safety.