Developing Action Policies with Q-Learning and Shallow Neural Networks on Reconfigurable Embedded Devices

ALWYN BURGER and GREGOR SCHIELE, University of Duisburg-Essen, Germany
DAVID W. KING, Air Force Institute of Technology, USA

The size of sensor networks supporting smart cities is ever increasing. Sensor network resiliency becomes vital for critical networks such as emergency response and waste water treatment. One approach is to engineer “self-aware” sensors that can proactively change their component composition in response to changes in work load when critical devices fail. By extension, these devices could anticipate their own termination, such as battery depletion, and offload current tasks onto connected devices. These neighboring devices can then reconfigure themselves to process these tasks, thus avoiding catastrophic network failure. In this article, we compare and contrast two types of self-aware sensors. One set uses Q-learning to develop a policy that guides device reaction to various environmental stimuli, whereas the others use a set of shallow neural networks to select an appropriate reaction. The novelty lies in the use of field programmable gate arrays embedded on the sensors that take into account internal system state, configuration, and learned state-action pairs, which guide device decisions to meet system demands. Experiments show that even relatively simple reward functions develop both Q-learning policies and shallow neural networks that yield positive device behaviors in dynamic environments.

CCS Concepts: • **Computing methodologies** → **Reinforcement learning**; **Q-learning**; *Neural networks*; • **Hardware** → **Hardware accelerators**; • **Computer systems organization** → *Embedded systems*;

Additional Key Words and Phrases: Embedded devices, Q-learning, shallow neural networks, self-aware agents, FPGA

ACM Reference format:

Alwyn Burger, Gregor Schiele, and David W. King. 2021. Developing Action Policies with Q-Learning and Shallow Neural Networks on Reconfigurable Embedded Devices. *ACM Trans. Auton. Adapt. Syst.* 15, 4, Article 14 (December 2021), 25 pages.

<https://doi.org/10.1145/3487920>

1 INTRODUCTION

Future smart cities will contain millions of interconnected systems dedicated to providing everything from critical services such as emergency response and traffic management to the more mundane such as wifi. One example of such a system is a set of interconnected traffic signals that must autonomously collaborate to manage traffic flow through the city both during peak traffic

The authors acknowledge the financial support by the Federal Ministry of Education and Research of Germany in the KI-Sprung LUTNet project (project number 16ES1125).

Authors' addresses: A. Burger and G. Schiele, Embedded Systems Group, University of Duisburg-Essen, 47057 Duisburg, Germany; emails: {alwyn.burger, gregor.schiele}@uni-due.de; D. W. King, Dept of Electrical and Computer Engineering, Air Force Institute of Technology, Dayton, Ohio, USA; email: david.king@afit.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2021 Copyright held by the owner/author(s).

1556-4665/2021/12-ART14 \$15.00

<https://doi.org/10.1145/3487920>

hours, such as the beginning and ending of the work day, as well as off peak hours such as 2 am. These sensors will have lots of connections to similar systems and other systems, such as devices on their shared network, as well. If these components can communicate, then it is possible to use free processing time on these devices to help complete tasks that other devices may need help accomplishing. Enabling these devices to change their onboard configuration would allow them to better assist their neighbors. We argue that if we build such reconfigurable devices, and add “self-awareness” (i.e., they know their internal state and how other components perceive them [22]), then it is possible to construct systems that can proactively change their component composition, providing assistance to connected critical systems that may be experiencing high work loads or catastrophic loss.

Adaptive hardware embedded systems using **field programmable gate arrays (FPGAs)** present a possible solution to this device reconfiguration. FPGAs are programmable circuits that can be customized through programming logic to perform specific tasks using hardware acceleration. Devices could use an onboard FPGA to change which tasks they can accomplish, creating a task-flexible agent. The challenge for the device then becomes when to switch its configuration (and thereby which tasks it can perform), as any changes will affect neighboring devices and, by extension, the system as a whole. In other words, the device must balance its current workload and life expectancy with system demands. This work compares two different methods—Q-learning and **shallow neural networks (SNNs)**—to create self-aware agents that learn “optimal” behavior patterns. Here and throughout the rest of this article, *optimal* refers to the selection of what the agent considers the best action for its current state. These methods allow the agent to respond to shifting system demands, such as the failure of other devices in the network, to keep the system running as optimally as possible. In this work, the agents are not optimizing over a specific set of tasks but are optimizing their collective response to changes in network demands. It is important to note the devices are acting autonomously and are learning responses individually, free from centralized control.

Our methodology for developing autonomous, decentralized learning agents is inspired by natural systems such as ant [14] and bee [18] colonies. Individual insects that compose these colonies are autonomous and self-aware, meaning that they make action choices based on their perception of the world around them in conjunction with any learned behaviors both from their personal histories and other colony members. Foraging bees, for example, can independently learn areas of high predation and pollen sources. Once they return to the hive, these bees share this information with other bees in the colony. As information flows across the colony, bees make action choices, such as avoiding the area with a large number of predators, without having to visit the location themselves—saving colony resources. The pooling and sharing of intelligence across the hive, which can be defined as collective intelligence, creates a series of information-feedback loops across the entire colony. Although the bees make choices independently (i.e., autonomously), these independent action choices aggregate to meet global system requirements, such as resource collection, brood sorting, and so on, creating a balanced, self-maintained system free from centralized control. This type of decentralized-control system scales extremely well, overcoming limitations of centralized control methods. For our proposed system, desired system goals are accomplished through the emergence of **division of labor (DOL)** produced by the decisions and interactions of individual devices, which, like bees and ants, autonomously learn what actions to select over time.

This approach is different from previous work where agents have a predefined decision function [21], devices learn optimal behaviors through the use of Q-learning [39] and SNNs, enabling devices to explore and find optimal responses to dynamic system change. This work uses the off-loading decision problem found in **mobile edge computing (MEC)** [24] as a case study to show

the power of coupling reconfigurable hardware agents with **reinforcement learning (RL)** techniques implemented in a decentralized manner. In the offloading decision problem, a device must decide whether to process a received job or offload it to another device connected to it. For example, in the smart cities problem mentioned earlier, a smart traffic light may record video of an event that it needs processed using computer vision or **artificial intelligence (AI)**. Solving the offloading problem then tells the device whether to process it itself, or in the case where its resources are better spent collecting further data or conserving energy it may choose to offload to a neighboring device. Our methodology leverages autonomous agents (i.e., devices) independently learning how to react and make effective decisions in a highly dynamic environment created by the offloading decision problem.

The Elastic Node hardware platform [5, 7, 36] provides the testing environment for this work. The nodes (i.e., self-aware devices) include a low-powered **micro-controller unit (MCU)** and a small, locally controlled, embedded FPGA. These devices can deploy various hardware accelerators onto the FPGAs at runtime, allowing them to solve advanced mathematical tasks. Each task requires a specific FPGA configuration. As reconfiguring an FPGA costs considerable energy, devices must decide when to process tasks, reconfigure their FPGA, or pass tasks to other connected devices. In this manner, we model a dynamic sensor network composed of self-aware, autonomous devices.

The rest of the article is structured as follows. Section 2 introduces the problem. Section 3 covers current embedded device methodologies for learning optimal behaviors. Section 4 describes our implementation of Q-learning with respect to reconfigurable embedded devices. Section 5 outlines the performed experiments. Finally, Section 6 concludes the article.

2 PROBLEM STATEMENT

The binary offloading decision problem is popular in MEC and **mobile cloud computing (MCC)** communities [12, 25, 27]. In these types of edge or cloud networks, when a device encounters a new task it must decide whether to perform it locally or offload it (usually to a connected server). The device tries to maximize some cost function based on carefully defined system performance criteria that balances the device's energy characteristics while considering the processing loads of both the device and the device's targeted server.

This work extends the binary offloading decision problem by allowing devices a third option: batching. A device may batch a job (i.e., place it into a queue) for later processing. Additionally, devices may decide to process all batched jobs at once, creating a different energy dynamic than offloading tasks to the computationally limitless cloud. Devices developed this way can decide to (1) remain in one configuration to batch jobs that match its current configuration, (2) offload jobs that do not match, or (3) process jobs in its queue. This approach has multiple benefits: first, devices can conserve energy by not reconfiguring their FPGA for each job; second, devices overcome greedy policy methods as batching produces a device capable of long-term planning; and third, this self-aware approach creates a responsive and adaptable system where teams of devices dynamically form and dissipate in response to system requirements.

Formally, given a set of devices $\mathbb{D} = \{1, \dots, D\}$ in which each device, d_i , includes a local FPGA, an MCU, and wireless transceiver. A device, d_i , can complete any job it receives from the set of jobs, $\mathbb{J} = \{1, \dots, J\}$ if the device is in the correct configuration. Each job is represented by the tuple (j, d_i, s_n, T_n) , where j is the time the job was received, d_i is the device that created the job, s_n is the input data size, and T_n represents the computational load of the job.

Initially, all jobs are homogeneous, and any job can be solved by any device. This maps neatly to a system where the jobs are primarily computational, as the local FPGA can instantiate any required

processing architecture. When using more varied jobs (or ones that require special additional hardware), the system becomes more heterogeneous as different devices may specialize in different jobs.

For improved accuracy, we model the completion of a job J as a collection of subtasks $\mathbb{S}(J)$. These include MCU-based tasks (e.g., data preparation, communication tasks) and FPGA-based processing tasks. Each requires time to complete—for example, offloading a job from one device to another requires both devices to perform a task of duration $\delta_n = \frac{s_n}{\tau}$, where τ is the transmission rate between them.

For each subtask S_i , each device can dynamically change the power state of its components between three different states: $\mathbb{P} = \{\text{active}, \text{idle}, \text{sleep}\}$. This creates a total instantaneous power at timestamp j for device i that has N components of $P_i^j = \sum_{n=1}^N P_n^j$, where P_n^j is the component power based on the power state required by the current subtask. For example, when offloading a job to another device, the MCU and the communication device are active, whereas the FPGA sleeps to preserve energy.

Components, such as the MCU, have simple power consumption that is only dependent on the current power state, assuming the absence of frequency scaling. However, FPGA power consumption is more complex, dependent on the complexity C_k of the computational task being performed (i.e., the amount of FPGA resource being utilized). Additionally switching from one power state to another can be very costly—for example, changing an FPGA out of sleep requires a reconfiguration.

Energy efficiency is nearly a universal optimization goal for embedded applications. Therefore, the energy consumed by device d_i to complete a job J_n is given as

$$E(J_n, d_i) = \sum_{S \in \mathbb{S}(J_n)} P_i^j \times \delta(S), \quad (1)$$

where $\delta(S)$ is the duration of subtask S . This is an important metric that captures the cost to the device of computing a job, considering all subtasks involved.

Each duration $\delta(S)$ needs to be modeled for simulation, based on which subtask is being performed and what computational task is associated with it. For example, FPGA computational time can be calculated using

$$\delta_n = \frac{C_k s_n}{f_{i,FPGA}}, \quad (2)$$

where $f_{i,FPGA}$ is the frequency of the FPGA and C_k is the complexity of the task. This complexity translates directly to the number of clock cycles required to compute for a data size $s_n = 1$, which can be directly gleaned from the accelerator design.

The goal of this work is to create autonomous, decentralized agents that can accomplish high-level system goals, such as maximizing completed jobs or maintaining system uptime. In this article, we mainly focus on maximizing the number of jobs within an episode—which represents a system lifetime—by balancing $E(J_n, d_i)$ with completing workloads. The agents use Q-tables or SNNs to learn a decision policy through a training episode which is then used as a starting point in the next episode. Training continues until device actions aggregate into system goals: completing the maximum number of jobs possible within an episode.

3 RELATED WORK

Recent work on learning optimal embedded device behaviors view it as either a binary or partial computation offloading problem [27]. In the binary offloading approach, certain tasks cannot be broken down and either must be completed by the device or offloaded to a more capable device, whereas in partial computation, pieces of the task can be broken apart and sent to multiple devices to accomplish such tasks as parallel execution [27].

Within the MEC and MCC communities, interest in the binary offloading problem has grown recently as MEC and MCC adoption increases and applications create greater computational loads on small power-limited devices. Approaches to solving the dilemma of maximizing resources vary in complexity and use different AI techniques to teach the devices the most optimal behavior.

Some systems [10, 25] are mostly concerned with a greedy approach that models the expected costs of offloading versus local computation. In more complex approaches, Liu et al. [25] and Mao et al. [28] use Lyapunov optimization. However, Lyapunov approaches seek to balance a requirement system-wide, meaning that it is a centralized approach requiring global knowledge of sensor states that is unrealistic, as the solution will not scale with the size of the system. Zhang et al. [42] use Markov decision processes. Although unique, Zhang et al. [42] kept the task queue size small to limit the exponential growth in the possible decision space, limiting the flexibility of the solution. The same critique can be levied at graph based modeling approaches, such as Guo et al. [16], where the growth of tasks can make the search space intractable. Others such as Ma et al. [26] and Chen [11] use game-theoretic approaches to formalize the competing goals of users and devices, creating a decentralized computation offloading game, but these experiments, again, must keep the search space small for computational reasons.

Others move beyond the binary offloading approach. Xu et al. [41] created a system for regularly allocating data streams to one of a set of servers. The primary objective of this is to load balance the servers. As an extension to this, Chen et al. [9] offer the edge and servers as a multi-layered offloading problem. However, it appears that the local device is still limited to a binary offloading decision.

RL and **Deep-Q network (DQN)** methods have gained traction in the MEC and MCC communities. Xu et al. [41] and Huang et al. [17] used a deep RL agent in the cloud to perform offloading decisions on behalf of all users in the system. They placed emphasis on the communications overhead, and the computation was simplified to offloading streams to make the problem easier to solve. Similarly, Li et al. [24] and Chen et al. [12] used DQN to optimize a cost function that manages server load and task deadlines. However, one can debate how “deep” these networks were, as researchers appeared to stick to one hidden layer in their networks. Salmani et al. [35] trained a **deep neural network (DNN)** to maximize channel gains while meeting user requirements. Their DNN consisted of more than 24,000 hidden nodes (eight hidden layers with 3,000 nodes per layer), and although it outperformed other methods in CPU time, one could argue that the network is overfitting the domain and would produce irregular results if the number of candidates from their experiment exceeded two. In other words, the trained DNN would not be robust to dynamic changes in requirements.

A set of more modestly sized neural networks is used by Xiao et al. [40] in their actor-critic design, controlling numerous parameters such as transmission power and offloading rate in an MEC system. They optimize a number of characteristics to overcome jamming and other signal interference, utilizing two convolutional and four fully connected layers totaling thousands of parameters.

In traditional Q-learning (originally developed by Watkins [39]), an agent learns a mapping of states to actions based upon a learned policy π . The policy uses the Bellman equation (Equation (3)) [37] to update Q-values, for particular state-action pairs, $Q(s_t, a_t)$, with perceived rewards, r_t :

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]. \quad (3)$$

Typically, the agent stores state-action pairs in a table (i.e., a Q-table), which is updated as the agent gains rewards from taking actions in the environment. The agent ends up learning a policy,

π , that guides future decisions. The maintenance and size of the Q-table is the major drawback to this approach, as one may not be able to enumerate all possible states the agent will encounter. Furthermore, the table grows exponentially as states and actions are added, making the Q-table approach infeasible in many real-world cases. A similar argument could be made against the XCS classifier approach developed by Butz and Wilson [8] with respect to the population sizes used by the underlying genetic algorithm portion of XCS to find a solution. To overcome the limitation of Q-table sizes, Mnih et al. [31] developed **Deep Q-learning (DQL)**, where a neural network substitutes for the Q-table.

In DQL, a DNN replaces the policy selection action, removing the requirement for a Q-table. The agent develops an estimate of the policy function as the neural network updates its component weight matrices learning a mapping between states and actions. DQL updates the state-action value mapping (i.e., updates its weights) at every timestep, just like Q-learning. However, to overcome catastrophic forgetfulness [33], such as erasing weights of previously learned state-action pairs, the network retrains old data during updates. In addition to catastrophic forgetfulness, DQL architectures could also overcome concept drift in a similar manner.

According to Wang et al. [38] and Minku and Yao [30], concept drift occurs when the learned space (i.e., the environment the network learned in) changes over time. For example, let us say we built an agent in the 1950s that tells a user how many nations exist in the **European Union (EU)**. There were six founding nations for the EU in the 1950s [38] and has increased, and decreased, over the years. The addition and subtraction of nations effectively change the state of the environment, and the agent would necessarily need to update its view of the world or it would give a user bad information. In this example, changes happen over years where agent updates could be slow with relatively little harm to system performance—a 6- to 12-month update cycle could effectively capture EU updates. However, as pointed out by Minku and Yao [30], for online systems, such changes can occur rapidly, requiring systems to sense changes and adapt in real time. As DQN-based agents retrain during updates, these agents could change their weights and adapt as the environment shifts, in real time if necessary, allowing them to combat both catastrophic forgetfulness and concept drift.

Although deep RL techniques are popular, they rely on large networks where the number of trainable parameters (i.e., neural network weights) far outnumber the number of training data samples [32]. This “overparameterization” is what theoretically allows DNNs to learn more complex features spaces than their SNN counterparts [2]. However, an argument can be made that if the learning space is not overly complex, then deep networks are overkill, placing an unnecessary demand on device memory and energy use, as well as the accompanying large training dataset requirement. It is then reasonable to turn to smaller, more shallow, neural networks (i.e., networks whose architectures contain one or two hidden layers) to accomplish the same learning task. For the rest of this work, we refer to agents using SNNs as **shallow reinforcement learning (SRL)** agents.

In supporting literature, McDonnell et al. [29] show that SNNs, consisting of one hidden layer and using their extreme learning machine (ELM) approach, can perform just as well as deeper networks on MNIST classification test sets. Similarly, Jiang and Crookes [19] present and test small unorganized neural networks (SUNNs) on early image recognition tasks. Their small networks boast a 2D structure that use both adaptive neurons and random interconnections to learn tasks. These small networks were able to learn early edge detection faster than other state-of-the-art techniques that used DNNs, experimentally proving that SNNs can outperform deeper neural networks in less-complex environments.

This work expands on previous work by directly comparing the performance of agents using Q-learning and SRL models as their decision-making processes. Our approach also has distinct

differences from other work. First, local FPGAs can reconfigure their internal structure to meet task demands. In other words, the device array can dynamically respond to changes in its environment. Second, instead of optimizing a complex cost function, our approach simplifies agent development. We present a way of intuitively building a device reward function based on system priorities and architecture and show how even simple reward functions can create “smart” agents who quickly learn state-action mappings. Third, we show that SRL agents provide additional advantages over Q-learning agents. This is accomplished by translating the learning problem from one of Q-values to classification. The restructuring of the problem decreases the complexity of the decision state-space, making it ideal for SRL agents. Furthermore, we show that SRL agents prove to be more robust in dynamic environments despite their relatively small size. Finally, our approach creates a fully decentralized, autonomous system of self-aware devices that can operate without user or server access, making a vital contribution to the field that eschews some of the problem simplification techniques described earlier, creating a more realistic system.

4 METHODOLOGY

In this section, we provide a more detailed overview of developing and employing our approach. We first state that our approach is fundamentally centered on being able to model different, real-world scenarios. In other words, the goal is to make the approach generalizable. For example, networks of our devices can represent an autonomous drone swarm exploring a city, capturing images and LIDAR information. Instead of offloading these images to the cloud for processing, drones could self-organize into teams where one team switches to a computer vision mode using principal component analysis (PCA) [7] or a convolutional neural network (CNN) [5] to identify potential items of interest while other drones continue to patrol and capture information. As reconfiguring onboard FPGAs comes with an energy cost, drones could collect inputs from their peers to improve the accelerator’s efficiency as it dedicates itself to one particular type of workload, only switching when a certain threshold, or a higher priority task, is encountered. This is analogous to ant behaviors found in nature where ants switch between tasks based on input stimuli received from local neighbors (i.e., other ants passing by) [3]. As shown in previous work [21], internal agent decision functions heavily impact task-division outcomes across the system; therefore, careful consideration of how devices make decisions must be accomplished and tested before real-world employment. Finally, our approach was guided by a rule of simplicity. Building a Q-table agent, as well as a DQN agent, is programatically straightforward, without having to rely on other algorithmic techniques, such as genetic algorithms or rule construction and matching, to achieve a competent agent.

4.1 System Model and Assumptions

We have certain assumptions about the adaptive hardware devices as well as the corresponding infrastructure. First, we assume that devices have local access to the required FPGA configurations. This usually involves over-the-air (OTA) updates for sending and updating the relevant program code and bit files. In the case of the Elastic Node, it can freely choose which of the available configurations to load onto the embedded FPGA from local flash memory, and fetch missing ones from a nearby computer.

Second, we assume that each device includes a wireless device for exchanging jobs with peers. On our platform, this is generally a low-energy local communication protocol (e.g., 802.15.4), but the solution presented in this article can be exchanged for any other local or wide-area communication protocol. We further assume that all devices can communicate with each other (i.e., a cooperative model).

Fig. 1. Agent overview showing the relationship between the agent, its knowledge, and its environment (adopted from Russell and Norvig [34]).

Third, a basic requirement of creating self-aware devices is introspection. Based on the definition provided by Lewis et al. [22], they should at least be able to monitor themselves and their environment. One example of providing this is the fine-grained passive power monitoring available on the Elastic Node [36]. It provides per-component details on the consumption of the system (MCU, FPGA, etc.), giving insight into the energy cost of a specific action.

4.2 Proposed Solution

The foundation of our approach is a control learning system that is decentralized and device-facing. Therefore, we require that each agent can operate using information it would realistically have access to. This mostly pertains to the system state, which can conceptually contain any information pertaining to the agent, the device, or their surroundings.

The goal-based agents of Russell and Norvig [34] (Figure 1) best describe the agents built in our experiments. Goal-based agents have sensors that perceive the environment in some manner and through the use of condition-action rules select an action to take. These actions can influence the environment as well as the internal conditions of the agent.

Our agents receive jobs from other agents they are connected to in their network. A Q-table, consisting of state-action pairs, provides the agent with a mechanism for action selection. The *state* of the world consists of the job (and corresponding computational task) received, the internal state of the agent (battery life, current configuration, etc.), and a rough estimate of how much battery life will be depleted if certain actions are selected. The agent uses this information, and the values stored in the Q-table, to select the action that maximizes the number of jobs it can complete before its life is over (i.e., its battery depletes).

Figure 2 provides a more detailed view of how the device decision and learning process works. The agent builds the current system state from the device's internal status (e.g., battery life and batched tasks), as well as the environment, which, in our experiments, consists of tasks received from other devices in the network. The device information may contain static configuration values, such as its physical or contextual location, or more dynamic measures, such as current battery level or wireless connection quality. The complete system state is mapped in a Q-table to possible actions, such as offloading a job, batching a job, or reconfiguring the FPGA. Each state-action mapping in the Q-table contains a learned reward value based upon previous state-action selection outcomes. The agent uses a greedy- ϵ policy [37] to select what action to take, ensuring that if the

Fig. 2. Single iteration of the training phase for our Q-based agent, including selecting an action, enacting it, and updating its policy based on the reward received.

environment changes at a later stage, it can adapt its learned behavior. After taking the action, it updates its internal Q-table using Equation (3).

The difficulty an engineer faces for Q-learning is threefold. First, reward functions often need to be developed and refined over time to generate desired outcomes. This means deploying reward functions, testing and training agents, and measuring outcomes in a plan-test-execute cycle that can be time consuming. Second, Q-tables can grow exponentially if state-action tuples are not pruned. Pruning of states can be done through generalization (e.g., multiple states or actions are viewed as synonymous) or some heuristic guidance where non-reachable states are dropped from the table entirely. However, these methods still cannot overcome the final limitation in Q-table approaches—what if a new state is encountered? This leads some researchers, as described in Section 3, to look at DQL; however, we take a slightly different tact. We investigate this problem by introducing SRL agents into our experimental scenarios. These agents use smaller neural networks than those found in the current DQL research, keeping them more in line with the smaller Q-table agents when it comes to energy consumption and memory usage. This allows us to compare the performance of agents employing Q-tables to ones that use neural networks on small devices. Specifically, we are interested in introducing new states to trained agents and studying how quickly and effectively each agent can adapt. We hypothesize that SRL agents should be robust to these changes while Q-table-based agents will necessarily need to first expand their Q-tables to accommodate the new state-action pair as well as requiring time to learn the proper Q-values related to these new state-action pairs. We expect this to result in Q-table-based agents taking longer to converge to a new steady state when compared to SRL ones.

4.3 Agent Design

In this section, we show how to set up Q-learning and SRL agents and apply them to a specific domain. One point of this work is to make AI techniques such as Q-learning more accessible to engineers who may not have a background in AI. What follows is the methodology for designing Q-learning and SRL agents, along with supporting experiments showing how these agents “learn” optimal decisions over time. We begin with a description of Q-learning agents, their action space, state space, and reward functions. The section then transitions to SRL agent setup and training.

4.3.1 Q-Learning. For Q-learning agents, Q-table values are updated after action selection via a reward function. Reward functions for agents can be difficult to develop. However, simple, heuristic guided reward functions can often yield optimized behaviors. The main goal of our agents is currently to complete as many jobs as possible before their battery life is depleted, meaning that they receive a positive reward for jobs completed and negative rewards for unfinished ones.

4.3.2 Action Space. Using the Q-Learning State-Action diagram, we illustrate the functionality of a typical system of peer-based reconfigurable embedded devices that loosely represents the scenario described in Section 1. Shown in Figure 3 is some sample behavior of a device that starts

Fig. 3. Example state-action space for experimental agent when the “Batch” action is chosen.

with no jobs in its queue and then creates a new job by sensing (e.g., capturing an image) its environment. It can either decide to not compute this job and offload it to a peer, or reconfigure its FPGA and start processing it. Instead, it decides to batch it for later and then creates a second job. Again the same action choices as before are available to it, this time with one existing job in its queue.

This forms the basis for our first experimental agent, which will be expanded upon to show how flexible our solution is to different agents and scenarios. The agent’s decision to offload, batch, or reconfigure is mostly guided by its current energy levels and configuration.

If a job does not match the device’s current configuration or its energy level is too low, the device may choose to *offload* the job to a connected peer. The peer device may proceed with the job or continue to offload the job to yet another device. The important part to note here is the decision is entirely local—that is, up to the agent, not the system as a whole.

If the current configuration does not match the received job, the device may *batch* the job into a queue. Batching a job requires very little energy. Theoretically, batching jobs can improve efficiency as the device can queue a large batch of jobs and then execute (i.e., complete them) in one execution run. This also saves the device energy, as it is not reconfiguring for each job received as those are offloaded to peer devices. However, batching jobs can delay the completion of jobs, so the agent must learn how to balance queuing too many jobs and executing them.

Finally, the device may begin performing the local jobs. This can include reconfiguring the FPGA if it was in a sleep state or reconfiguring it to address the jobs in question. If these jobs addressed a variety of different computational tasks, the FPGA would need to be reconfigured multiple times to perform all the tasks. However, all jobs that require the same computational task can be performed with only a single FPGA reconfiguration (i.e., in a *batch*). Reconfiguring the FPGA incurs the highest energy cost but is a requirement for local job processing. Once the device begins processing jobs, it will continue until all jobs are completed.

An important concept to consider when designing the agent’s action space is *heuristic behavior*. This refers here to predefined actions or procedures that the user dictates. In some cases, this can provide the user powerful control over the agent’s behavior by removing unwanted freedom. Instead of learning either *unwanted* behavior or wasting time learning *obvious* behavior, the user can nudge the agent in the right direction. This type of guidance is usually seen in AI game design where search algorithms can be given information about certain states, such as a board score, which indicates if the position is “good” or “bad” based on “expert” opinion. However, this limits the ability of an agent to truly explore the search space where, sometimes, a very “good” outcome may lie just a few decision steps away from an apparent “bad” state.

The only heuristic embedded in our devices is the batch process where a device will perform all batched jobs once it starts processing its queue where they are stored. Up to this point, the device freely chooses to batch or offload jobs. This can be altered in the future, but larger batches generally lead to improved energy efficiency, as it avoids additional future reconfigurations. Additionally, the scenario being tested refers to critical tasks that must be done at all costs. Therefore, when the device reaches a critical energy state (in current experiments, the minimum measurable battery level), it enters *graceful death* and offloads all its queued jobs randomly to peers in its environment. This ensures that all jobs get solved but can cause a strain on the system if it is suddenly flooded with a high number of jobs.

4.3.3 System State. As decentralized control is at the core of this work, it is important that the system state is (1) limited to information available to the device in the real world and (2) relative to the device so the agent can be migrated from one device to another. The basic system state for our devices consists of its energy state, number of batched jobs, and FPGA configuration. The *energy state* of the battery powering the device can be measured or estimated locally, and provides the primary indication of expected remaining lifetime. Commonly, this is represented by a discretized battery indicator.

The *number of batched jobs* is limited by the available memory of the devices. Batching involves locally storing the input data and configuration of each job, so it can be completed at a later stage. The local state reflects the number and types of each task on the device. As switching between types of tasks involves reconfiguring the FPGA, they are considered independent batches, and the device only considers the relevant batch.

The current *FPGA configuration* can greatly affect the chosen action. Therefore, its configuration state is included as a binary variable that indicates whether the correct one is currently available on the FPGA. Keeping this variable binary also reduces the number of resultant state-action pairs reducing the possible exponential growth of the Q-table, a known limitation of Q-learning approaches.

4.3.4 Reward Functions. The primary control mechanism for application developers is through the reward function. As a first step, we developed a few different agents that each have their own objectives. Although simple agents such as *local only* or *random* can be created, the reward function distinguishes normal agents from each other.

As a demonstration, we created two distinct agents that we refer to as *basic agent* and *lazy agent*. While the basic agent attempts to balance saving device energy and processing jobs, the lazy agent only aims to survive for as long as possible. This provides an indication of the control available via the abstractions provided by the reward function.

The reward function of the basic agent is given by

$$r = R_j + R_e + R_d, \quad (4)$$

where R_j is the job reward given by $R_j = J_{batch}$ —that is, the number of jobs computed so far in the batch. The energy reward R_e is simply $R_e = -\frac{\sum P_i^j}{E_i}$, where E_i is the total energy of device i 's battery—normalizing the used energy based on the total amount available.

Note that the device does not require any energy prediction abilities to choose the optimal action. By rewarding based on passed (measured) energy usage, the expected reward (Q) of a state-action pair reflects the likely energy costs. In this way, we do not need *a priori* knowledge on the energy usage of the different actions. As mentioned in Section 2, each action results in a number of subtasks that one or more devices have to perform (e.g., offloading a job to a peer requires them to send/receive the sampled data). This makes it impractical to predict accurately,

which can be additionally difficult beforehand due to unpredictable environmental effects such as wireless interference. Instead of requiring the devices to perform complex and possibly inaccurate predictions, they simply measure their own energy costs.

Last, R_d is the death reward given by $R_d = \{-10 \text{ if battery critical}(d_i) \text{ else } 0\}$, where the -10 is chosen empirically to numerically balance the job and energy rewards so one part of the reward function does not overpower the rest. The death reward refers here to the *graceful death* state described before, effectively dissuading agents from reaching a low battery state. The reward function of the lazy agent is then simplified to $r = R_e + R_d$, entirely disregarding the reward for finishing a job and focusing exclusively on low energy consumption and avoiding death.

The reward function shown in Equation (4) shows how simple it is to control device intent. By adding or removing a reward based on the device's current state, the job being processed, or the task to be done, the user can influence the agent's objectives and goals. When fine-tuning their agent during the simulation phase of the development cycle, the balance between the terms of this equation offers a simple way to emphasize one over the other.

Since each device can be given a different reward function, they can be directed toward different goals. Some may be given a more aggressive task-solving agent, whereas others are aimed toward prolonging the life of the group. Again, parallels can be drawn to swarm behavior where subgroups are created in heterogeneous applications. In the domain of computational devices, this may be instantiated with cooperating drones that must balance individual goals such as energy saving with group goals such as performing a high-level task.

4.3.5 Shallow Reinforcement Learning. Neural network-based RL agents differ greatly from Q-table-based agents. Instead of maintaining a Q-table, these agents rely on a trained neural network to make decisions. The neural network learns a classification policy (π') that is different from the Q-table agents. This reduces the state-space the neural network has to learn. Instead of having to learn approximate values for m state-action pairs, the neural network directly learns how to map states to three actions: offload, batch, or local. The input to the neural network is the same as the Q-table agent, as it receives a vector containing the agent's current state. The vector's composition changes depending on the experiment, but the goal is the same: choose to offload a job, save a job in the queue, or begin processing all jobs in the queue.

Designing neural networks is part art and part science. By the universal approximation theorem, we know that one can design a neural network to approximate any function [13]. However, one usually does not know what the architecture of the neural network should look like. The process of building a neural network usually consists of trial and error: design a topology, assign random weights, train, and test. If the network does not reliably converge to satisfactory performance, the engineer can try a few different techniques.

One option is for the engineer to reinitialize the neural network with different starting weights. It has been shown that certain starting weights can prevent a neural network from converging [13], so a reinitialization may be all that is necessary for the neural network to learn its primary function. Second, one can train an ensemble of neural networks, either averaging their overall performance or simply selecting the best-performing models during training. In this manner, one can be fairly confident that one or more models should converge, and these networks are then used during experimentation or production. Another popular method is to add more hidden layers to the network, creating a "deeper" network. The increase in hidden layers increases the model's tunable parameters, which increases the likelihood of modeling the desired function correctly. However, increasing the size of a network comes at a cost.

First, larger networks usually require more training data that may not always be available. Second, larger networks lead to longer training times, as it is more computationally intense. Third,

these larger networks could grow to a size that exceeds the memory size of small, portable devices. For example, the GPT-3 (Generative Pre-trained Transformer-3) system (OpenAI's latest natural language processing model) contains 174 billion tunable parameters [4]. GPT-3's memory footprint is on the order of 350 GB, far exceeding the capabilities of mobile devices. Although one can argue that deeper networks generalize better [13], there is a hard memory limit for mobile devices, thus limiting the available "depth" one can practically use in their neural network design.

A similar argument can be made against expanding Q-table agents. As the number of states increases, so does the size of the table, meaning that there is an absolute limit to the number of state-action pairs a Q-agent can learn. Here is where neural networks may provide added flexibility. If one can design and train a neural network to generalize to a variety of states, then it may be able to perform the same as—if not better than—a similarly sized Q-table agent. Furthermore, it can estimate new states it has not seen before with the same memory requirement—that is, the neural network does not expand like the Q-table is required to do when encountering a new state. Removing the expansion requirement would greatly benefit small mobile devices whose memory footprint is severely limited, as they could show added flexibility in dynamic environments without having to change their internal structures.

We decided to test this last hypothesis with our SRL agents. To do so, we had to derive a network architecture that could perform as well as a Q-table agent in different scenarios. Once we found a SRL agent that performed as well as a Q-table agent, we then set to test both in an environment where previously unseen tasks (i.e., new states) were encountered. The Q-table agents would expand their Q-tables while the neural network was held to the same size. The purpose of these experiments was twofold: show that SRL agents can perform as well as Q-table agents in baseline experiments, and that generalized neural networks provide a reasonable way forward for decision-making models when memory and resources are at a premium.

Figure 4 shows the SRL architecture developed for our experiments. During design, we specifically focused on keeping the neural network small so as to not exceed memory requirements.¹ Surprisingly, we ended up with a neural network with only one hidden layer with either 8 or 10 nodes, depending on the experiment. However, the SRL agent's goal was always the same: provide a suggested agent action based upon the agent's current state.

Our ability to minimize the size of our neural network is directly related to the task of classification versus learning individual Q-values for every state-action pair. One can view this as a way to generalize the decision-making process. Instead of learning a policy and accompanying Q-value for each state-action pair, $Q(s_t, a_t)$, the neural network is learning to map states to actions, which is a different agent policy, $\pi' = A(s_t)$. We found that the task transformation from derivation of values to classification significantly reduced the number of required neural network parameters, and improved stability when Q-values for different actions were very similar. Since the network is not being expected to distinguish between unimportant losing actions (as a Q-table agent needs to), it can instead focus on choosing a winning action. It also forces the network to generalize since one cannot guarantee that it will see every possible state-action pairing during training. Theoretically, this will allow the network to yield a "best guess" for what action an agent should take when encountering a new state.

To train these SRL agents, we need to provide "corrected" classifications results based on the reward function created earlier. Akin to Q-learning, we use the Bellman equation (Equation (3)) to incorporate the received reward—either increasing or decreasing the previously inferred

¹We settled on **rectified linear units (ReLU)** in the hidden layer through early experimentation where these nodes appeared to converge quicker than other types. This highlights part of the art and science struggle with developing neural network architectures that we alluded to earlier.

Fig. 4. Neural network architecture for SRL agents. The input vector consists of current battery status, queue size, FPGA configuration status, and the type of task passed to the agent. The neural network produces a classification output that chooses the opportune action.

classification weight. Over time, this pushes the neural network to choose the action that receives the highest reward for each provided state. As they use the same reward update function, the differences in the underlying complexities lie primarily from the memory requirements. Using a small network for the SRL agent suggests that its memory footprint should be overtaken by an expanding Q-table.

5 EVALUATION

Although evaluating the absolute performance of the agents designed in Section 4.3 offers some insight into what can be accomplished with our solution, our primary objective is to evaluate the system itself. This focuses on whether or not the system works—that is, “can a meaningful agent be created by defining the *system state*, *action space*, and *reward function*?” Therefore, we performed some experiments using the created agents to show how well they accomplish their intrinsically defined goals.

In the scenario simulated here, the group of devices creates jobs at a regular interval T . The job rate is shared for the entire group, meaning that a group of two devices would create the same number of jobs in a given time frame as a group of four. Every T seconds, one random device in the system will sample a new set of data (i.e., create a job) and its agent will assign the job an action. This action is then performed until the job is finished being processed. If the job is passed to a new device, that device’s agent assigns it a new action.

The following experiments were run with a small simulated initial device battery energy of 1.33 J for each episode, and all power consumption is characterized from an Elastic Node v4 that sports the 8-bit Atmel AT90USB128 and Xilinx Spartan 7 S50. This device has been shown to efficiently compute neural networks [6] and provide highly accurate per-component power measurements in real time [36]. A Gaussian distribution is fit for the usage of each component (separately for multi-voltage components) to accurately represent real-world power usage. The relevant distributions are then sampled for each device at each timestep and fed to its system state and reward function, as the middleware-based device software of the Elastic Node can provide this information.

5.1 Experiment 1: Basic Agent Comparison

As a first step, we compare the performance of the “basic” and “lazy” agents as described in Section 4.3.4. Their system estimate is based on the simplest version of the system state, which is

Fig. 5. Comparison of average jobs per episode between basic and lazy Q-table agents and a random agent. Includes centralized (C) and decentralized (D) versions.

based only on the current queue and the energy state. The primary goal is to show that learning optimal outcomes can be accomplished even with simple reward functions. For comparison, a *random agent* is also introduced that chooses each action at random, without regard for learning.

The performance of each agent is evaluated by studying how many jobs a team of two devices can achieve in an episode (defined as ending when all devices run out of energy). Five different teams were created. The first team introduced a shared Q-table between devices. This effectively models a centralized learning and control approach that reduces its real-world applicability but is used to show how learning affects system outcomes. In the second team of devices, each learn individually without sharing a Q-table. This models the preferred network of autonomous devices in a decentralized control approach. Basic agents composed these first two teams. Teams three and four were exactly the same; however, lazy agents made up the teams. Finally, a team of random agents composed the final team.

Figure 5 clearly shows that a team of basic agents performs considerably more jobs after approximately 50 episodes for both centralized and decentralized approaches. Interestingly, individual lazy agents learn to survive better without sharing a Q-table, possibly reflecting a pure “self-greedy” approach to survival. No statistical difference in the number of jobs completed by teams of basic agents employing either centralized or decentralized methods was produced with both approaches quickly converging to the same values. We hypothesize that this is because the system is designed to utilize the same device-centric information in both the decentralized and centralized variations.

In general, basic agents accomplish more jobs on average per episode than the lazy agents, driven by their reward for performing jobs. Next are the lazy agents, performing generally fewer jobs than the basic agents but still outperforming the random agents. Note that this graph has datapoints for each episode and includes error bars, making it infeasible for the lines to be dashed or dotted.

This is a positive result, as it shows that decentralized approaches, which we argue are a better approach for systems with large populations, learn optimal behaviors just as quickly as centralized

Fig. 6. Impact of a larger job queue on agent effectiveness, allowing agents to create larger batches without processing jobs.

methods with respect to Q-learning. All teams outperformed the random agent team, which makes sense as random agents were not equipped with a learning capability. After episode 1,000, the centralized and decentralized lazy agents slowly converge in the same fashion as the basic agent.

5.2 Experiment 2: System State Expansion

From early experiments, we discovered that having a maximum queue size has a strong impact on agent performance. This is primarily caused by the agent having to reconfigure once the queue size is filled. In this experiment, we tested the impact of queue size on agent performance. We also decided to implement a “learning” phase and a “production” phase for the agents that models real-world employment of learning agents where policies are first learned in a simulation setting before deploying into a real-world network.

A number of similar agents “learned” individual policies for the first 1,000 episodes. After the 1,000th episode, the agents switched to “production” mode for 10 episodes. Here, all learning is halted and agents follow a pure greedy policy by selecting the available action with the highest Q-table score. Results are shown in Figure 6.

One interesting takeaway from this experiment is that the performance difference between the two agents is more distinct for larger job queue sizes. This can be attributed to the agent having more control over the actions of the device, as the reconfigure action from having a full queue is less frequent. Instead, the agent can freely choose when to change the configuration of the device.

In this case, the lazy agent learns that with large job queue sizes (>200), it can simply keep all jobs in its queue and not process any. This behavior follows logically when one considers that the lazy agent’s reward function does not consider any job-based reward, and only focuses on saving energy. In this case, batching is always the more energy-efficient option over offloading or processing locally.

A well-known limitation of the Q-table is its size expanding exponentially with larger state spaces. Each individual state needs to be explored and learned, and its expected reward stored in

the Q-table. Especially when the agent should be deployed to a memory-limited embedded device such as the Elastic Node, this is obviously impractical. This is where neural network-based agents can be highly beneficial, as they can interpolate between known states, find similarities between different interchangeable states, and possibly represent the full Q-table with fewer parameters.

5.3 Experiment 3: Heterogeneous Agents Sharing an Environment

So far, we have shown the impact of using different agents to control a team of devices. Based on their defined system state and reward function, a user can encourage various high-level objectives. While considering both centralized and decentralized learning, these teams have all been uniform and homogeneous. This means that they all try to accomplish the same outcome, and therefore one would expect them to tend to learning similar Q-tables.

However, this is not necessarily the most realistic or optimal setup for a real-world scenario. Agent heterogeneity (and the unpredictability of one device to another) has been shown to create unexpected and interesting behaviors. For example, King et al. [20] showed that heterogeneous teams could outperform homogeneous ones in task coverage scenarios. However, the composition of the team had a direct impact on its efficacy. In other words, heterogeneous teams were not always the best at performing the task at hand in all scenarios.

To investigate this, we created different heterogeneous teams that include the basic and random agents from Experiment 1. A team of 10 agents are deployed together in the same environment (where again any device can communicate with any other device). For each team, a different percentage of them are basic agents (varying from 10% to 100%), whereas the rest are random agents that do not learn. Our goal here is to see what impact these unpredictable and different agents have on the learning and performance of our basic agents. Therefore, our chosen metric for each team is the average number of jobs completed by the deployed basic agents.

The results of this is shown in Figure 7, where the performance for each team is shown over the first 200 episodes. It is clear that the homogeneous team consisting only of basic agents (100% Basic Agents) performs considerably better than the mixed teams, with performance decreasing with more random agents being introduced. This culminates when there is only a single basic agent in a team of otherwise random agents, where the average number of jobs is decreased by 57%. Initially, each of the basic agents explore the entirely unknown policy (Q-table) causing their average jobs to decrease for the first 10 episodes. Once more states become known and each option has been tried (steady state Q-values tend to be lower than initialized values to encourage initial exploration), the performance of all teams begin to improve.

This shows that although the basic agent still manages to improve its performance in this scenario, the rate of learning and relative performance is heavily affected by having these unpredictable and different agents in its environment. Reminiscent of socially situated agents [1], the agents each need to manage their own objectives. In this case, the ability of the basic agents to learn optimal behaviors is hindered by interactions with agents that have different goals. The number of jobs the basic agent is exposed to diminishes as more lazy agents are introduced into the system, reducing both the number of jobs the basic agents can complete but also winnowing down the parts of the search space they encounter. This results in agent teams that do not work well together. Future work could explore this further by connecting more teams dedicated to multiple competing tasks and measure the impact of competition of agent learning.

5.4 Experiment 4: Division of Labor Investigation

Our final experiment ties back to our introductory problem where devices may fail, forcing other devices to pick up the extra workload. This scenario can quickly destroy interconnected networks if devices are not adaptable. Our main hypothesis argues that adaptable devices should be able to

Fig. 7. Performance of different compositions of heterogeneous teams of basic and random agents, showing that higher concentrations of basic agents perform more jobs on average per episode.

handle such failures, keeping the overall system viable for a longer period of time. To measure the adaptability of our devices, we introduce the idea of device specialization. In natural systems, agents such as bees and ants can change the job they are doing as the environment changes. In ant colonies, there can be an increase in specialization (i.e., more ants dedicated to one particular job) for short periods of time to meet colony needs. For example, when food is discovered, many ants change to food recovery over other jobs such as nest cleaning to quickly harvest the food resource. The key is measuring this change in agent behavior. Gorelick et al. [15] developed a DOL metric that yields the current DOL score of a population. The DOL measurement is calculated via a combination of individual agent entropy and the mutual entropy of the agent and the entire population. First, an $n \times m$ matrix is developed where rows represent agents and columns represent tasks. Agents selecting a task increase the associated agent-task (i.e., row-column) value. After an episode, the matrix is normalized and Shannon's entropy is used to calculate each agent's entropy score across all tasks. The mutual entropy is

$$I(X, Y) = \sum_{x \in X, y \in Y} p(x, y) \log \frac{p(x, y)}{p(x)p(y)} \quad (5)$$

across all agents. Dividing by Shannon's index,

$$D_{y|x} = \frac{I(X, Y)}{H(X)}, \quad (6)$$

yields the DOL score along the interval $[0, 1]$. Using this metric, we placed our devices through a 1,000-episode training scenario. The Q-table agent is henceforth represented by the Basic Agent from previous experiments. We leverage previous experiments in which King and Peterson [21] showed that DOL scores increased when agents underwent dramatic shifts in the environment, leading to stronger system resiliency. To test our network devices, we remove half of the device

Fig. 8. Average number of jobs completed per episode for different-sized teams under catastrophic failure of half the team.

Fig. 9. DOL scores indicating the change in specialisation for different sized teams under catastrophic failure of half the team.

population at the 500th training episode, emulating a catastrophic loss of device sensors in a real-world environment.

Figures 8 shows the number of jobs completed by the devices, whereas Figure 9 shows the DOL scores. The larger teams perform more jobs on average per episode, with the team of 12 devices doing roughly twice as much as the team of 4. Note that they are not creating more jobs within the

same time period (as the job creation is global to the experiment), but instead collaborating better and having more energy stored in total in their batteries. Interestingly, initially all teams have a fairly low DOL—indicating that all teams are showing low specialization.

We see that the DOL increases after the 500th episode. The population becomes more specialized when reduced by half. Surprisingly, this specialization results in more jobs being completed, on average, after the catastrophic loss. One possible explanation for this behavior is that early on there are too many devices per job resulting in higher idle times—that is, more devices not completing jobs resulting in inefficient use of resources. Devices then respond to efficiently complete jobs as they become specialized in response to the loss of half the networked devices. These results support our hypothesis that adaptable agents, armed with a learning algorithm, can successfully navigate large-scale system failures.

5.5 Experiment 5: Q-Table Versus Shallow-Learning Agent Baseline Comparison

The recent push in both academia and industry to use neural networks to solve various problems naturally leads to the following question: are neural networks better than traditional approaches? The overhead associated with the architecture design, training, and test methodology (combined with the associated data requirement) places a strict requirement on neural networks: they must perform at least as well as—and hopefully better than—traditional approaches to justify the up-front time and energy cost. The next two experiments compare the performance of our shallow-RL agents against the Q-table agent developed for earlier experiments. In this experiment, we compare the baseline performance of both pre-trained and untrained agents.

Pre-trained agents go through their learning process in an offline supervisory manner. Each agent is trained on a large pool of state-action pairs with their weights updated according. More specifically, the corresponding Q-table values are filled in while the weight adjacency matrices are updated using gradient descent. Once the agents converged and reached a sufficient accuracy rating, they were deployed into the production simulation environment.

Untrained agents started with a fresh Q-table of zero values, and SRL agents started with randomized adjacency matrices. As mentioned, it has been shown [13] that initial weights can prevent a neural network agent from converging to a desired accuracy rating. To overcome this, we imposed an ensemble-like methodology where we collected the average performance of the top 10 producing neural networks. This allowed us to see the expected performance of a shallow-learning agent and reduced the variance associated with random weight initialization. However, this does point out one of the potential pitfalls with using neural networks. One could have the correct architecture and a proper training dataset; however, due to the random generation of weights, the neural network may not learn the desired behavior.

Our first shallow-RL experiment in Figure 10 compares the performance of both pre-trained and untrained models. The goal of the experiment is to see if shallow-learning agents perform as well as their Q-table counterparts. The results are largely positive. First, pre-trained agents achieve a steady state quickly after deployment onto the production system, which they should as they have “seen” the possible state-action pairs they encounter on the production system. Both the untrained agents also manage to learn their new environment, whereas the Q-table agents converge slightly slower to the same performance as their pre-trained counterparts.

The results indicate that if practical, use of pre-training neural networks is the best approach before deploying them onto a system. Online learning (i.e., learning once deployed onto a system) could result in subpar performance when compared to traditional Q-learning agents. As stated earlier, this is why our shallow-learning agents are an ensemble of neural networks with the same architectures but different initial weights. These results also highlight that traditional methods still perform very well in the age of neural networks. This emphasizes one critical component

Fig. 10. Comparison of the job performance for SRL and Q-table agents, both pre-trained and newly initialized to learn from scratch.

of system design: understanding the potential environment the agents will work in and its affect upon their behaviors. The next experiment will explore this idea even further.

Interestingly, this result further suggests that the SRL agents (both pre-trained and not) provide a more stable result than the Q-table variant. We expect that this is due to them being able to provide a better initial guess for unseen states. As the Q-table agent comes across new states, its performance is naturally reduced slightly while it randomly guesses a good action. Later, as it learns the expected value of this state, its performance stabilizes again until the next “new” state is found. In contrast to this, the SRL agent can better generalize between the new unknown state and already trained states—providing a better initial guess.

5.6 Experiment 6: Q-Table Versus Shallow-RL Agent in Dynamic Environments

Following the baseline comparison experiment, we decided to test one known shortfall of Q-table agents, namely that they cannot deal with unknown states. If an unknown state is encountered, the best a Q-table agent can do is select a random action. In addition, if one wants the agent to handle such states in the future, the Q-table agent must increase the size of its Q-table accordingly so it can observe the reward of its action selection and update accordingly. Eventually, the Q-table agent should be able to learn the new state-action pairing. However, this comes at a cost. The first few times the agent encounters a state, its performance will naturally be degraded. Additionally, the agent will require more memory to store its new table rows and columns. It is here that shallow-RL agents could possibly outperform Q-table ones.

Shallow-learning agents learn a pattern of behavior over states and have the capacity to generalize these learned patterns to new states, meaning that their initial “guess” for taking an action in an unknown state is better than random. This generalization power alleviates memory requirements, and these networks do not need to grow or change in the environment. In this experiment, untrained Q-table and shallow-RL agents were deployed onto the production system. At first, agents receive “difficult” jobs to complete. A difficult job is defined as one requiring a

Fig. 11. Job execution performance of SRL and Q-table agents under introduction of new tasks, highlighting the SRL agent's ability to adapt to changing scenarios by avoiding the transient performance reductions.

longer time to process—that is, requires more system resources to complete thus costing more energy. After 1,000 episodes, agents receive slightly easier jobs, “medium-high difficulty,” they have not seen before. At 2,000 episodes, the agents receive medium-low difficulty jobs, and, finally, at 3,000 episodes, they receive easy jobs.

The change in difficulty only means that new difficulty jobs are added as possible inputs, meaning that agents can still receive other jobs of various difficulties they have encountered before. Therefore, we would expect all agents to accomplish more jobs per episode later in the experiment. When Q-table agents encountered a new state, they expanded their Q-tables, filling in the new state-action pairings with a zero (i.e., defaulted to random action selection until the pairing was “learned”). Shallow-RL agents, however, merely updated their weight matrices after giving their answer. In other words, the architectures and memory requirements for shallow-RL agents remained static, whereas Q-table agents had to expand their memory requirements whenever encountering new states.

Figure 11 shows an interesting pattern of behavior. The shallow-learning agents learn the difficult job pairings faster than their Q-table counterparts. The real benefit of shallow-learning agents is easily seen during transition periods. As new jobs of various difficulties are added, Q-table agents invariably suffer a massive degradation in performance as they try and learn how to handle these new states. Meanwhile, shallow-RL agents either hold steady or actually increase the number of jobs they complete during the same time frame. The Q-table agents do eventually converge to the same behavior, which is in line with earlier experiments. However, the performance lag during transition times definitely supports using shallow-learning approaches in highly dynamic environments. However, we must again caution that the performance of the shallow-learning agents is based upon an average of best-performing networks, so one must be vigilant in monitoring agent performance on production systems.

Finally, these experiments show that knowing your problem domain is critical for selecting the appropriate agent decision model. Although the Q-table agent lags behind initially, it does

converge to the same behavior as the shallow-learning models. And even though it requires an increase in memory resources, it does not grow uncontrollably in this particular experiment. Again, understanding the production environment is key: for smaller state-action spaces, Q-table agents are a better choice than shallow-learning agents. They require less “black box” design, training, and test methodology to build and deploy. The main issue with Q-table agents is building a good reward function to help guide them toward correct actions. For shallow-learning agents, the issue is not only in selecting the correct architecture but overcoming the random initialization of weights, meaning that one must deal with more hyper-parameter tuning to ensure desired outcomes. In short, neural networks can be worthwhile, especially in cases where agents must generalize learned knowledge; however, they are not the only way to solve problems.

6 CONCLUSION

Controlling the actions of reconfigurable embedded devices such as the Elastic Node to accomplish high-level objectives is non-trivial. We have shown in this work that this can be done effectively using RL with Q-tables and SRL agents. By defining the system state from locally available information such as FPGA configuration and current energy state, the devices have been shown to effectively decide how to handle incoming computational tasks in the form of jobs. Choosing between changing the local FPGA configuration to perform the job, offloading it to a peer, or batching it for later, the agents have been shown to prioritize long-term job solving efficiency.

The system we showed here provides a start toward truly decentralized intelligent hardware-accelerated devices. By allowing the agent to target the user’s complex long-term goals, sophisticated behavior like proactively mitigating device loss is possible. Since the device’s behavior is not hard-coded or defined directly using cost functions, it can learn how to target more complex user requirements. Furthermore, an SRL agent can generalize between known and unknown states better than a Q-table agent, providing stability and adaptability in dynamic environments.

In the future, it is our hope to extend the action space of the agent to provide more fine-grained control of the FPGA power state. Currently, the FPGA is automatically put to sleep when not used for a short period of time, but this could be extended if the device expects more jobs to be available in the near future. Additionally, other offloading options will be investigated to provide the device further opportunity for optimization. For example, offloading to an edge or centralized server could provide possibilities in some applications that require processing power outside the abilities of embedded hardware. Finally, similar to the work of Lewis et al. [23] with regard to distributed camera network strategies, the impact of agent heterogeneity in the system could be explored further to see if certain groupings of agents lead to greater efficiencies in system performance.

REFERENCES

- [1] Chloe M. Barnes, Anikó Ekárt, and Peter R. Lewis. 2019. Social action in socially situated agents. In *Proceedings of the International Conference on Self-Adaptive and Self-Organizing Systems (SASO’19)*. 97–106. <https://doi.org/10.1109/SASO.2019.00021>
- [2] M. Bianchini and F. Scarselli. 2014. On the complexity of neural network classifiers: A comparison between shallow and deep architectures. *IEEE Transactions on Neural Networks and Learning Systems* 25, 8 (2014), 1553–1565. <https://doi.org/10.1109/TNNLS.2013.2293637>
- [3] Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. 1999. *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press.
- [4] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, et al. 2020. Language models are few-shot learners. arXiv:2005.14165 [cs.CL].
- [5] Alwyn Burger, Chao Qian, Gregor Schiele, and Domenik Helms. 2020. An embedded CNN implementation for on-device ECG analysis. In *Proceedings of the 2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops’20)*.

- [6] Alwyn Burger and Gregor Schiele. 2018. Demo abstract: Deep learning on an elastic node for the Internet of Things. In *Proceedings of the 2018 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops'18)*. 555–557.
- [7] Alwyn Burger, Patrick Urban, Jayson Boubin, and Gregor Schiele. 2020. An architecture for solving the eigenvalue problem on embedded FPGAs. In *Proceedings of the 33rd International Conference on Architecture of Computing Systems (ARCS'20)*.
- [8] Martin Butz and Stewart W. Wilson. 2001. An algorithmic description of XCS. *Soft Computing* 6 (2001), 3–4.
- [9] Meng Hsi Chen, Min Dong, and Ben Liang. 2016. Multi-user mobile cloud offloading game with computing access point. In *Proceedings of the 2016 5th IEEE International Conference on Cloud Networking (CloudNet'16)*. 64–69. <https://doi.org/10.1109/CloudNet.2016.52>
- [10] Weiwei Chen, Dong Wang, and Keqin Li. 2018. Multi-user multi-task computation offloading in green mobile edge cloud computing. *IEEE Transactions on Services Computing* 1374, c (2018), 1–13. <https://doi.org/10.1109/TSC.2018.2826544>
- [11] Xu Chen. 2015. Decentralized computation offloading game for mobile cloud computing. *IEEE Transactions on Parallel and Distributed Systems* 26, 4 (April 2015), 974–983. <https://doi.org/10.1109/TPDS.2014.2316834>
- [12] Xianfu Chen, Honggang Zhang, Celimuge Wu, Shiwen Mao, Yusheng Ji, and Mehdi Bennis. 2019. Optimized computation offloading performance in virtual edge computing systems via deep reinforcement learning. *IEEE Internet of Things Journal* 6, 3 (2019), 4005–4018. <https://doi.org/10.1109/JIOT.2018.2876279> arXiv:1805.06146.
- [13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press, Cambridge, MA.
- [14] D. M. Gordon. 2003. The organization of work in social insect colonies. *Complexity* 8, 1 (2003), 43–46.
- [15] Root Gorelick, Susan M. Bertram, Peter R. Killeen, and Jennifer H. Fewell. 2004. Normalized mutual entropy in biology: Quantifying division of labor. *American Naturalist* 164, 5 (2004), 677–682. <https://doi.org/10.1086/424968>
- [16] Songtao Guo, Bin Xiao, Yuanyuan Yang, and Yang Yang. 2016. Energy-efficient dynamic offloading and resource scheduling in mobile cloud computing. *Proceedings of IEEE INFOCOM*. 1–9. <https://doi.org/10.1109/INFOCOM.2016.7524497>
- [17] Liang Huang, Suzhi Bi, and Ying-Jun Angela Zhang. 2018. Deep reinforcement learning for online offloading in wireless powered mobile-edge computing networks. arXiv:1808.01977. <http://arxiv.org/abs/1808.01977>.
- [18] Raphaël Jeanson, Penelope Kukuk, and Jennifer H. Fewell. 2005. Emergence of division of labor in halictine bees: Contributions of social interactions and behavioral variance. *Animal Behavior* 70 (2005), 1183–1193.
- [19] Richard Jiang and Danny Crookes. 2019. Shallow unorganized neural networks using smart neuron model for visual perception. *IEEE Access* 7 (2019), 152701–152714. <https://doi.org/10.1109/ACCESS.2019.2946422>
- [20] David W. King, Lukas Esterle, and Gilbert Peterson. 2019. Entropy-based team self-organization with signal suppression. In *Proceedings of the Artificial Life Conference (ALIFE'19)*.
- [21] David W. King and Gilbert L. Peterson. 2019. The emergence of division of labor in multi-agent systems. In *Proceedings of the IEEE 13th International Conference on Self-Adaptive and Self-Organizing Systems (SASO'19)*. 107–116.
- [22] Peter R. Lewis, Arjun Chandra, Shaun Parsons, Edward Robinson, Kyrre Glette, Rami Bahsoon, Jim Torresen, and Xin Yao. 2011. A survey of self-awareness and its application in computing systems. In *Proceedings of the International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW'11)*. 102–107.
- [23] Peter R. Lewis, Lukas Esterle, Arjun Chandra, Bernhard Rinner, Jim Torresen, and Xin Yao. 2015. Static, dynamic, and adaptive heterogeneity in distributed smart camera networks. *ACM Transactions on Autonomous and Adaptive Systems* 10, 2 (2015), Article 8, 30 pages.
- [24] Ji Li, Hui Gao, Tiejun Lv, and Yueming Lu. 2018. Deep reinforcement learning based computation offloading and resource allocation for MEC. In *Proceedings of the IEEE Wireless Communications and Networking Conference (WCNC'18)*. 1–6. <https://doi.org/10.1109/WCNC.2018.8377343>
- [25] Juan Liu, Yuyi Mao, Jun Zhang, and Khaled B. Letaief. 2016. Delay-optimal computation task scheduling for mobile-edge computing systems. In *Proceedings of the IEEE International Symposium on Information Theory*. 1451–1455. <https://doi.org/10.1109/ISIT.2016.7541539>
- [26] Xiao Ma, Chuang Lin, Xudong Xiang, and Congjie Chen. 2015. Game-theoretic analysis of computation offloading for cloudlet-based mobile cloud computing. In *Proceedings of the 18th ACM International Conference on Modeling, Analysis, and Simulation of Wireless and Mobile Systems (MSWiM'15)*. 271–278. <https://doi.org/10.1145/2811587.2811598>
- [27] Yuyi Mao, Changsheng You, Jun Zhang, Kaibin Huang, and Khaled B. Letaief. 2017. A survey on mobile edge computing: The communication perspective. *IEEE Communications Surveys and Tutorials* 19, 4 (2017), 2322–2358. <https://doi.org/10.1109/COMST.2017.2745201>
- [28] Yuyi Mao, Jun Zhang, and Khaled B. Letaief. 2016. Dynamic computation offloading for mobile-edge computing with energy harvesting devices. *IEEE Journal on Selected Areas in Communications* 34, 12 (2016), 3590–3605. <https://doi.org/10.1109/JSAC.2016.2611964>

- [29] Mark D. McDonnell, Migel D. Tissera, Tony Vladusich, André van Schaik, and Jonathan Tapson. 2015. Fast, simple and accurate handwritten digit classification by training shallow neural network classifiers with the ‘extreme learning machine’ algorithm. *PLoS One* 10, 8 (Aug. 2015), 1–20. <https://doi.org/10.1371/journal.pone.0134254>
- [30] Leandro Minku and Xin Yao. 2011. DDD: A new ensemble approach for dealing with concept drift. *IEEE Transactions on Knowledge and Data Engineering* 24, 4 (2011), 619–633.
- [31] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529–533.
- [32] S. Oymak and M. Soltanolkotabi. 2020. Toward moderate overparameterization: Global convergence guarantees for training shallow neural networks. *IEEE Journal on Selected Areas in Information Theory* 1, 1 (2020), 84–105. <https://doi.org/10.1109/JSAIT.2020.2991332>
- [33] German I. Parisi, Ronald Kemker, Jose L. Part, Christopher Kanan, and Stefan Wermter. 2019. Continual lifelong learning with neural networks: A review. *Neural Networks* 113 (2019), 54–71.
- [34] Stuart Russell and Peter Norvig. 2003. *Artificial Intelligence: A Modern Approach* (3rd ed.). Prentice Hall.
- [35] Mahsa Salmami, Foad Sohrabi, Timothy N. Davidson, and Wei Yu. 2019. Multiple access binary computation offloading via reinforcement learning. In *Proceedings of the 16th Canadian Workshop on Information Theory (CWIT’19)*.
- [36] Gregor Schiele, Alwyn Burger, and Christopher Cichiwskyj. 2019. The elastic node: An experimentation platform for hardware accelerator research in the Internet of Things. In *Proceedings of the 2019 IEEE International Conference on Autonomic Computing (ICAC’19)*. 84–94. <https://doi.org/10.1109/ICAC.2019.00020>
- [37] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA.
- [38] Shengui Wang, Stefan Schlobach, and Michel Klein. 2011. Concept drift and how to identify it. *Journal of Web Semantics* 9, 3 (2011), 247–265.
- [39] Christopher John Cornish Hellaby Watkins. 1989. *Learning from Delayed Rewards*. Ph.D. Dissertation. King’s College.
- [40] Liang Xiao, Xiaozhen Lu, Tangwei Xu, Xiaoyue Wan, Wen Ji, and Yanyong Zhang. 2020. Reinforcement learning-based mobile offloading for edge computing against jamming and interference. *IEEE Transactions on Communications* 68, 10 (2020), 6114–6126. <https://doi.org/10.1109/TCOMM.2020.3007742>
- [41] Zhiyuan Xu, Yanzhi Wang, Jian Tang, Jing Wang, and Mustafa Cenk Gursoy. 2017. A deep reinforcement learning based framework for power-efficient resource allocation in cloud RANs. In *Proceedings of the IEEE International Conference on Communications*. 1–6. <https://doi.org/10.1109/ICC.2017.7997286>
- [42] Yang Zhang, Dusit Niyato, Ping Wang, and Chen Khong Tham. 2014. Dynamic offloading algorithm in intermittently connected mobile cloudlet systems. In *Proceedings of the 2014 IEEE International Conference on Communications (ICC’14)*. 4190–4195. <https://doi.org/10.1109/ICC.2014.6883978>

Received January 2021; revised August 2021; accepted September 2021