DeepWiERL: Bringing Deep Reinforcement Learning to the Internet of Self-Adaptive Things

Francesco Restuccia and Tommaso Melodia
Institute for the Wireless Internet of Things
Northeastern University, Boston, MA 02115 USA
Email: {frestuc, melodia}@northeastern.edu

Abstract*—Recent work has demonstrated that cutting-edge advances in deep reinforcement learning (DRL) may be leveraged to empower wireless devices with the much-needed ability to "sense" current spectrum and network conditions and "react" in real time by either exploiting known optimal actions or exploring new actions. Yet, understanding whether real-time DRL can be at all applied in the resource-challenged embedded IoT domain, as well as designing IoT-tailored DRL systems and architectures, still remains mostly uncharted territory. This paper bridges the existing gap between the extensive theoretical research on wireless DRL and its system-level applications by presenting *Deep Wireless Embedded Reinforcement Learning* (*DeepWiERL*), a general-purpose, hybrid software/hardware DRL framework specifically tailored for embedded IoT wireless devices. *DeepWiERL* provides abstractions, circuits, software structures and drivers to support the training and real-time execution of state-of-the-art DRL algorithms on the device's hardware. Moreover, *DeepWiERL* includes a novel supervised DRL model selection and bootstrap (S-DMSB) technique that leverages transfer learning and high-level synthesis (HLS) circuit design to orchestrate a neural network architecture that satisfies hardware and application throughput constraints and speeds up the DRL algorithm convergence. Experimental evaluation on a fully-custom software-defined radio testbed (i) proves for the first time the feasibility of real-time DRL-based algorithms on a real-world wireless platform with multiple channel conditions; (ii) shows that *DeepWiERL* supports 16x data rate and consumes 14x less energy than a software-based implementation, and (iii) indicates that S-DMSB may improve the DRL convergence time by 6x and increase the obtained reward by 45% if prior channel knowledge is available.

I. Introduction

It is now widely accepted that the impressive scale of the Internet of Things (IoT) – expected to reach 18B devices by 2022 [1] – will impose a never-before-seen burden on today's wireless infrastructure [2]. To make matters worse, existing IoT wireless protocols such as WiFi and Bluetooth are deeply rooted in inflexible, cradle-to-grave designs, and thus unable to address the demands of the next-generation IoT. If unaddressed, this "perfect storm" may lead to severe delays in the much-needed IoT's global development – if not its downfall. As such, it is today more important than ever to shift toward a vision where self-optimization and adaptation to unpredictable – perhaps, adversarial – spectrum conditions is not a "nice-to-have" feature, but is deeply intertwined with the IoT device's software and hardware fabric.

Simply put, it has now become crucial to re-engineer IoT devices, protocols and architectures to dynamically self-adapt to different spectrum circumstances [3]. Bleeding-edge advances in deep reinforcement learning (DRL) have recently stirred up the wireless research community, now rushing to apply DRL to address a variety of critical issues, such as handover and power management in cellular networks [4, 5], dynamic spectrum access [6–9], resource allocation/slicing/caching [10–14], video streaming [15–17], and modulation/coding scheme selection [18], just to name a few [19]. This comes with no surprise; DRL has shown to provide near-human capabilities in a multitude of complex tasks, from playing dozens of Atari video games [20] to single-handedly beating world-class Go champions [21]. In a nutshell, DRL algorithms solve partially-observable Markov decision process (POMDP)-based problems without any prior knowledge of the system's dynamics [22]. Therefore, DRL may be the ideal choice to design wireless protocols that **(i)** optimally choose among a set of known network actions (*e.g.*, modulation, coding, medium access, routing, and transport parameters) according to the current wireless environment and optimization objective; and **(ii)** adapt the IoT platform's software and hardware.

Why is Bringing DRL to the IoT Challenging?

Despite the ever-increasing interest in DRL from the wireless research community, existing algorithms (discussed in details in Section VII) have only been evaluated through simulations or theoretical analysis – which has substantially left the investigation of several key system-level issues uncharted territory. Clearly, this is not without a reason; the resource-constrained nature of IoT devices brings forth a number of core research challenges – both from the hardware and learning standpoints – that are practically absent in traditional DRL domains and that we summarize below.

The first key aspect is that DRL is based on a *training phase*, where the agent learns the best action to be executed given a state, and an *execution phase*, where the agent selects the best action according to the current state through a deep neural network (DNN) trained during the training phase. Traditionally, DRL training and execution phases are implemented with GPU-based software and run together in an *asynchronous manner*, *i.e.*, without any latency constraints. On the other hand, in the embedded wireless domain, the DRL execution phase must run in a *synchronous* manner – meaning with *low, fixed latency* and with *extremely low energy consumption* – features that only a *hardware implementation* can provide. This is because (i) the wireless channel changes in a matter of a few milliseconds and is subject to severe noise and interference [23], and (ii) RF components operate according to strict timing

constraints. For example, if the channel's coherence time is approximately 20ms, the DNN must run with latency much less than 20ms to (i) run the DNN several times to select the best action despite of noise/interference; and (ii) reconfigure the hardware/software wireless protocol stack to implement the chosen action – all without disrupting the flow of I/Q samples from application to RF interface. As far as we know, existing work does not consider the critical aspect of real-time DRL execution in the wireless domain, which is instead considered and carefully analyzed for the first time in Section IV.

To further complicate matters, the strict latency and computational constraints necessarily imposed by the embedded IoT wireless domain *should not not come to the detriment of the DRL performance*. Indeed, usually DRL algorithms are trained in very powerful machines located up in the cloud, where we can afford computationally-heavy DRL algorithms and DNNs with hundreds of thousands of parameters [20]. This is obviously not affordable in the IoT domain, where devices are battery-powered, their CPUs run at few hundreds of megahertz, and possess a handful of megabytes of memory *at best*. Therefore, a core challenge is how to design a DNN "small" enough to provide low latency and energy consumption, yet also "big" enough to provide a good approximation of the state-action function. This is particularly crucial in the wireless domain, since the RF spectrum is a very complex phenomenon that can only be estimated and/or approximated on-the-fly. This implies that the stationarity and uniformity assumptions [24, 25] usually made in traditional learning domains may not necessarily apply in the wireless domain. All this considered, it is necessary to provide a design methodology to explore which DNN architectures are appropriate (in terms of latency and convergence) to approximate state-action functions in the embedded IoT domain.

Real-Time Embedded Deep Reinforcement Learning

We believe that the above core challenges can only be addressed through novel, IoT-specific DRL designs and architectures. As such, this paper proposes *Deep Wireless Embedded Reinforcement Learning* (*DeepWiERL*), the first DRL system carefully designed to bridge the existing gap between theoretical and system-level aspects of wireless DRL in the IoT landscape. The key and critical innovation behind *DeepWiERL* is to bring to the IoT research community what has been missing so far – a general-purpose framework to design, implement and evaluate the performance of IoT-tailored real-time DRL algorithms on embedded devices.

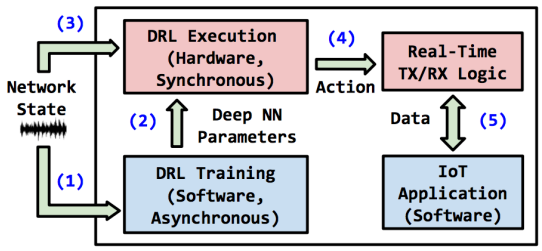


Fig. 1: The *DeepWiERL* approach to DRL in the IoT.

Figure 1 provides an eagle's-eye overview of the approach taken by *DeepWiERL*. Specifically, the system's core operations are divided into two different yet tightly intertwined tasks: **(i)** an asynchronous, software-based *DRL training* (step 1), where the algorithm learns to select the best policy according to the specific task at hand; and **(ii)** a synchronous, hardware-based *DRL execution*, where the results of the training, *i.e.*, the deep neural network (DNN) parameters, are periodically sent (step 2) to the IoT platform to enforce the execution of the policy. The IoT device, in turn, runs the hardware-based DNN to select an action from the current network state (step 3) and then enforce the action in the transmission/reception logic (step 4). The IoT application can then transmit/receive its data according to the new wireless logic decided by the DNN (step 5).

DeepWiERL is fundamentally different from existing DRL systems and frameworks, since it physically separates two traditionally interconnected steps – DRL training and execution – by **(a)** placing the DNN on the hardware portion of the IoT platform to guarantee real-time constraints; and **(b)** interconnecting the DNN both to the DRL training phase and to the RF components of the platform to enforce the real-time application of the action selected by the hardware-based DNN. This allows at once to **(i)** guarantee real-time and low-power requirements and **(ii)** make the general-purpose and applicable to a multitude of software-based DRL training algorithms.

Novel Technical Contributions

The design of *DeepWiERL* and its implementation on a practical testbed was a daunting task, as it required us to intertwine concepts from the fields of real-time embedded systems and deep reinforcement learning – which, before this paper, have been treated separately. As a result, this paper demonstrates for the first time the system-level applicability of DRL algorithms to the embedded real-time IoT landscape.

Specifically, we make the following technical advances:

- We propose *DeepWiERL*, an IoT-tailored framework providing real-time DRL execution coupled with tight integration with DRL training and RF circuitry (Section III). Specifically, we (i) design a DRL framework for system-on-chip (SoC) architectures integrating RF circuits, DNN circuits, low-level Linux drivers and low-latency network primitives to support the real-time training and execution of DRL algorithms on IoT devices (Section IV); and (ii) propose a new *Supervised DRL Model Selection and Bootstrap* (S-DMSB) technique (Section V) that combines concepts from transfer learning and high-level synthesis (HLS) circuit design to select a deep neural network architecture that concurrently (a) satisfies hardware and application throughput constraints and (b) improves the DRL algorithm convergence;

- We prototype *DeepWiERL* on a customized software-defined radio testbed (Section VI), and demonstrate *DeepWiERL* on a concrete physical-layer rate maximization problem, where the transmitter uses a customized variation of the low-computation cross-entropy DRL algorithm [26] to

optimize the modulation scheme in real-time. We also compare the latency performance of *DeepWiERL* with a software-based Pytorch implementation, and compare S-DMSB with a traditional "clean-slate" approach where the DNN weights are initialized at random. Experimental results obtained with our testbed indicate that (i) *DeepWiERL* supports 16x more data rate and 14x less energy consumption than a software-based implementation; and that (ii) S-DMSB may speed up the convergence by 6x and obtains 45% better reward than a clean-slate DRL algorithm if prior state information is available.

II. Background on Deep Reinforcement Learning

Reinforcement learning (RL) can be broadly defined as a class of algorithms providing an optimal control policy for a Markov decision process (MDP). There are four elements that together uniquely identify an MDP, *i.e.*, (i) an action space $\mathcal{A}$, (ii) a state space $\mathcal{S}$, (iii) an immediate reward function $r(s,a)$, and (iv) a transition function $p(s,s',a)$, with $s,s' \in S$ and $a \in A$). The core challenge in MDPs is to find an optimal policy $\pi^*(s,a)$, such that the discounted reward $R = \sum_{t=0}^{\infty} \gamma^t r(s_t,a_t)$, $s_t \in S$ and $a_t \in A$ is maximized, where $0 \leq \gamma \leq 1$ is a discount factor and actions are selected from policy π^*. We refer the interested reader to [22] for a detailed introduction to MDPs and RL.

Different from dynamic programming (DP) strategies, RL can provide an optimal MDP policy also in cases when the transition and reward functions are unknown to the learning agent. Thanks to its simplicity and effectiveness, Q-Learning is one of the most widely used RL algorithm today. Q-Learning is named after its $Q(s,a)$ function, which iteratively estimates the "value" of a state-action combination as follows. First, Q is initialized to a possibly arbitrary fixed value. Then, at each time t the agent selects an action a_t, observes a reward r_t, enters a new state s_{t+1}, and Q is updated. The core of Q-Learning is a simple value iteration update rule:

$$Q(s_t,a_t) = (1-\alpha)\cdot \underbrace{Q(s_t,a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1},a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}} \quad (1)$$

where r_t is the reward received when moving from the state s_t to the state s_{t+1}, and $0 < \alpha \leq 1$ is the learning rate. An "episode" of the algorithm ends either when state s_{t+1} is a "terminal state" or after a certain number of iterations.

The main issue with traditional RL is the so-called *"state-space explosion"* problem, in other words, explicitly representing the Q-values in real-world problems is prohibitive. Let's assume, for example, that we use a vector of 64 complex elements to represent the channel state in a WiFi transmission (*i.e.*, number of WiFi subcarriers). Therefore, we would (potentially) need to store all possible vectors $S \in \mathbb{R}^{128}$ in memory – which is obviously intractable, especially for the limited memory available in embedded IoT devices.

Deep reinforcement learning (DRL) addresses the state-space explosion issue by using a deep neural network (DNN), also called Q-Network, to "lump" similar states together by using an non-explicit, non-linear representation of the Q-values, *i.e.*, a deep Q-network (DQN). This way, we (i) use Equation (1) to compute the Q-values, and then (ii) stochastic gradient descent (SGD) to make the DQN approximate the Q-function. Therefore, DRL trades off precision in state-action representation for a reduced storage requirement (which ultimately becomes $\mathcal{O}(1)$ complexity).

III. DeepWiERL: An Overview

Figure 2 provides a very high-level overview of the *DeepWiERL* system. At its core, we can describe *DeepWiERL* as a self-contained software-defined radio (SDR) platform where the platform's hardware and software protocol stack is continuously and seamlessly reconfigured based on the inference of a DRL algorithm.

A. Key Features

Due to the generalization and extrapolation abilities of neural networks, we cannot afford to use a single deep neural network (DNN) to both retrieve and learn the optimal Q-values. Critically, this is well known to lead to a slow or even unstable learning process [27–30]. Moreover, Q-values tend to be overestimated due to the max operator [31].

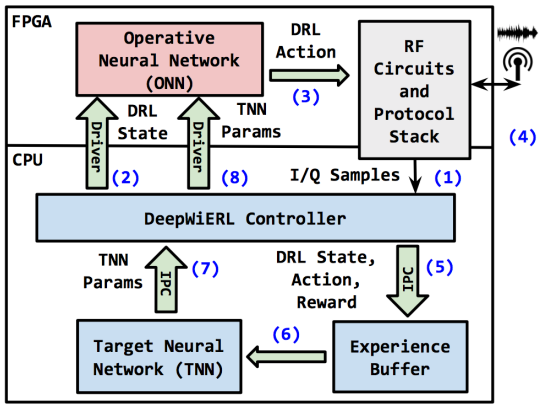


Fig. 2: An overview of the main components of DeepWiERL.

For this reason, *DeepWiERL* uses a *target neural network* (TNN), implemented in the edge/cloud, and an *operative neural network* (ONN), implemented in the FPGA portion of the platform. The ONN is updated with the TNN's parameters once every C DRL iterations to prevent instabilities. While the usage of two DNNs is (to some extent) straightforward in the software domain, achieving the same architecture in the embedded IoT domain is extremely challenging. Critically, this is because the TNN training will be performed over the course of minutes (or hours, in some cases), yet as explained in Section V-A, the ONN has to work in the scale of microseconds. Therefore, we took particular care in designing a hybrid synchronous/asynchronous architecture able to handle the different time scales, as explained in Section III-B. According to recent advances in DRL, we also leverage

an *experience buffer* [32] to store <state, action, reward> tuples for N time steps. The updates to the TNN are then made on a subset of tuples (called *mini-batch*) selected randomly within the replay memory. This technique allows for updates that cover a wide range of the state-action space.

B. A Walkthrough

Figure 2 shows *DeepWiERL*'s main operations. Throughout the paper, we will use red, blue and grey color to distinguish between hardware, software and hybrid software/hardware components. First, the wireless protocol stack – which includes both RF components and physical-layer operations – receives I/Q samples from the RF interface, which are then fed to the controller module (step 1). The controller module is tasked with creating a DRL state out of the I/Q samples, according to the application under consideration. The DRL state is then used sent by the controller to the *Operative Neural Network* (ONN), which resides in the FPGA and thus can only be accessed through drivers. The ONN provides with fixed latency a DRL *action* (step 3), which is then used to reconfigure in real-time the wireless protocol stack of the platform (step 4). This action can be anything from the physical layer ("change modulation to BPSK") to the MAC layer and above ("increase packet size to 1024 symbols, use a different CSMA parameters" and so on). Steps 2-4 are continually performed in a loop fashion, which reuses the previous state if a newer one is not available; this is done to avoid disrupting the I/Q flow to the RF interface. Section V-A will provide details on how the ONN has been implemented and the tradeoffs between accuracy and performance that must be achieved by the system.

Once the DRL state has been constructed, it is sent by the controller to the training module (step 5), which is located in another host outside of the platform (on the edge or up in the cloud). Thus, sockets are used to asynchronously communicate to/from the platform from/to the training module. The target of the training module is to (i) receive the <state, action, reward> tuples corresponding to the previous step of the DRL algorithm; store the tuples in the *experience buffer*; and (iii) utilize the tuples in the experience buffer to train the TNN according to the specific DRL algorithm being used (*e.g.*, cross-entropy, deep Q-learning, and so on). The ONN's parameters are updated with the TNN's parameters by the training module after each epoch of training (step 7). Finally, the ONN's parameters are updated through driver by the controller inside the platform (step 8).

IV. DeepWiERL: Hardware Design

Our design ethos was to keep *DeepWiERL* as general-purpose as possible yet also provide a concrete implementation on a real-world IoT platform. To accomplish the above, we choose a system-on-chip (SoC) architecture for our designs [33, 34], as SoCs (i) integrate CPU, RAM, FPGA, and I/O circuits all on a single substrate; and (ii) are low-power and highly reconfigurable, as the FPGA can be reprogrammed according to the desired design.

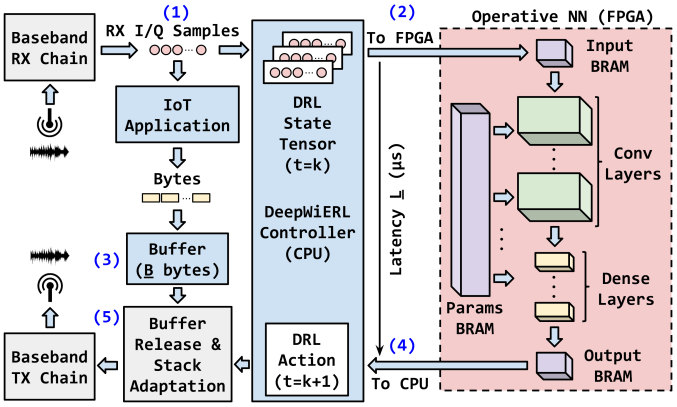


Fig. 3: The ONN and its main interactions with *DeepWiERL*.

A. Hardware Design Constraints

There are several key design constraints, both on time and memory, that need to be addressed in *DeepWiERL*, which we illustrate with the help of Figure 3. Once the I/Q samples have been received and processed by the RX chain (step 1), and the controller has created the ONN's input (*i.e.*, the DRL state tensor) as explain in Section V-A, the input is thus sent to the ONN through driver (step 2), which will provide the DRL action after a latency of L seconds (step 3). At the same time, the IoT application generates bytes, which have to be temporarily stored in a buffer of size B bytes, since the stack has to be reconfigured according to the selected DRL action.

Let us suppose that the RF interface is receiving samples at T million samples/sec. Usually, a digital-to-analog converter takes as input I/Q samples that are 4 bytes long in total. Therefore, $4 \cdot T$ MB worth of data must be processed each second to achieve the necessary throughput. Critically, since spectrum data is significantly time-varying, we need to assume that the ONN has be run S times each each second to retrieve the DRL action on fresh spectrum data. Furthermore, we cannot assume that the memory of the platform is unlimited. For the sake of generality, we assume the memory of the platform allows for maximum of B bytes of data to be buffered.

To summarize, in $1/S$ seconds, *DeepWiERL* needs to (i) insert $4 \cdot T/S$ bytes into a buffer (either in the DRAM or in the FPGA); (ii) send the DRL state tensor to the input BRAM of the ONN through driver; (iii) wait for the ONN to complete its execution after L seconds; (iv) read the DRL action from the output BRAM, (v) reconfigure the protocol stack and release the buffer. By experimental evaluation, we know that (i), (ii), (iv) and (v) are negligible with respect to L, therefore, we approximate those delays to zero for simplicity. Therefore, to respect the constraints, the following must hold:

$$\begin{cases} S \cdot L \leq 1 \text{ (time constraint)} & (2) \\ \dfrac{4 \cdot T}{S} \leq B \text{ (memory constraint)} & (3) \end{cases}$$

To give an example of the magnitude of the above contraints in real-world systems, let's assume that T = 20 MS/S (e.g.,

WiFi transmission) and that we want to sample the spectrum every millisecond $S = 1000$. To sustain these requirements, the ONN's latency L must be less than 1 millisecond, and the buffer B must be greater than 80 KB. We point out that the sampling rate T and the buffer size B are hard constraints imposed by the platform hardware/RF circuitry, and can be hardly relaxed in real-world applications. Thus, at a system design level, the only things that can be leveraged to meet performance requirements are L and S. Notice, moreover, that increasing S can help meet the second constraint (memory) but may fail the first constraints (time). On the other hand, decreasing S could lead to poor system/learning performance as spectrum data could be stale when the ONN is run. In other words, *we need to decrease the latency L as much as possible*, which in turn will (i) help us increase S (learning reliability) and thus (ii) help meet the memory constraint.

V. DeepWiERL: Deep Learning Design

In this section, we introduce the TNN and ONN design in Section V-A, followed by our novel supervised DRL model selection and bootstrap (S-DMSB) technique in Section V-B.

A. TNN and ONN Design

As explained in Section III-B, the TNN and ONNs are at the core of *DeepWiERL*. The ONN is entirely located in the FPGA portion of the platform while the TNN resides in the cloud/edge. This challenging yet crucial design choice allows for real-time (*i.e.*, known a priori and fixed) latency DRL action selection yet scalable DRL training.

The crucial target of the ONN is to approximate the state-action function of the DRL algorithm being trained in the TNN. On the other hand, differently from the computer vision domain, the neural networks involved in deep spectrum learning should be lower complexity and learn directly from I/Q data [34]. To address these challenges, and in accordance to recent work [35, 36], we decided to implement the TNN/ONN with a *one-dimensional convolutional neural network* (in short, Conv1D). We chose Conv1D over two-dimensional convolutional networks because they are significantly less resource- and computation-intensive than Conv2D networks, and because they work well for identifying shorter patterns where the location of the feature within the segment is not of high relevance. Similarly to Conv2D, a Conv1D layer has a set of N filters $F_n \in \mathbb{R}^{D \times W}, 1 \leq n \leq N$, where W and D are the width of the filter and the depth of the layer, respectively. By defining as S the length of the input, each filter generates a mapping $O^n \in \mathbb{R}^{S-W+1}$ from an input $I \in \mathbb{R}^{D \times S}$ as follows:

$$O^n_j = \sum_{\ell=0}^{S-W} F^n_{j-\ell} \cdot I_{n,\ell} \tag{4}$$

Creating the ONN Input. We now discuss how the *DeepWiERL* controller creates an input to the first Conv1D layer of the ONN from the I/Q samples received from the RF interface. Let us consider a complex-valued I/Q sequence $s[k]$, with $k \geq 0$. The w-th element of the d-th depth of the input, defined as $I_{w,d}$, is constructed as

$$\begin{aligned} I_{d,w} &= \text{Re}\{s[d \cdot \delta + w \cdot (\sigma - 1)]\} \\ I_{d,w+1} &= \text{Im}\{s[d \cdot \delta + w \cdot (\sigma - 1)]]\} \\ &\text{where } 0 \leq d \in D, 0 \leq w < W \end{aligned} \tag{5}$$

where we introduce σ and δ as intra- and inter-dimensional stride, respectively. Therefore, (i) the real and imaginary part of an I/Q sample will be placed consecutively in each depth; (ii) we take one I/Q sample every σ sample; and (iii) we start each depth once every δ I/Q samples. The stride parameters are application-dependent and are related to the learning vs resource tradeoff tolerable by the system.

B. Supervised DRL Model Selection and Bootstrap (S-DMSB)

(1) *Motivation.* A crucial problem for the embedded IoT domain is *selecting the "right" architecture for the TNN/ONN*. We know from Section IV-A that the ONN needs to be "small" enough to satisfy hard constraints on latency. At the same time, we need the ONN to possess the necessary depth to approximate well the current network state. It is intuitive that to allow DRL convergence, the TNN/ONN architecture should be "large enough" to distinguish between different spectrum states. A major challenge here is to verify constraints that are very different in nature – on one side, we have classification accuracy (software), on the other we have latency/space constraints (hardware). Therefore, we need a design methodology able to (i) evaluate those constraints and (ii) automatically transition from Python/PyTorch models to the FPGA bitfile.

On top of this, it is well known that DRL's weakest point is its slow convergence time [20]. Canonical approaches, indeed, start from a "clean-slate" neural network (*i.e.*, random weights) and explore the state space hoping the algorithm will converge. Existing work has attempted to tackle this problem through a variety of approaches, for example, exploring Q-values in parallel. However, these solutions are hardly applicable to the IoT domain, where resources are extremely limited and the wireless channel changes continuously. What we need in the wireless domain, instead, is a *bootstrapping* procedure where the TNN/ONN start from a "good" parameter set that will help speed up the convergence of the overarching DRL algorithm.

(2) *Our Approach.* We propose *Supervised DRL Model Selection and Bootstrap (S-DMSB)* to address the above issues at once through *transfer learning* [37]. Transfer learning allows the knowledge developed for a classification task to be "transferred" and used as the starting point for a second learning task to speed up the learning process. To explain why this process works in practice, we make the following example. Consider two people who want to learn to play the guitar. One person has never played music, while the other person has extensive music background through playing the violin. Intuitively, the person with an extensive music knowledge will be able to learn the piano faster and more effectively, simply by transferring previously learned music knowledge to the task of learning the guitar. Similarly, in the wireless domain, we can train a model to recognize different spectrum states, and let the

DRL algorithm figure out which ones yield the greater reward. This will at once help (i) select the right DNN architecture for the TNN/ONN to ensure convergence and (ii) speed up the DRL learning process when *DeepWiERL* is actually deployed.

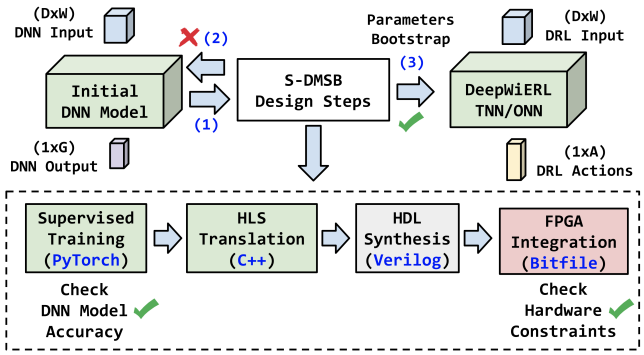


Fig. 4: Supervised Model Selection and Bootstrap (S-DMSB).

Figure 4 summarizes the S-DMSB technique, which is based on high-level synthesis [34] (HLS). In short, HLS translates a software-defined neural network to an FPGA-compliant circuit, by creating Verilog/VHDL code from code written in C++ [38]. The first step in S-DMSB is to train a DNN to classify among G spectrum states – for example, different SNR levels – low, medium, and high SNR, as in Figure 10. Once high accuracy (*e.g.*, 95%) is reached through hyper-parameter exploration, the model is translated with a customized HLS library that generates an HDL description of the DNN in Verilog language. Finally, the HDL is integrated with the other circuits in the FPGA and the DNN delay is checked against the requirements. In other words, if the model does not satisfy the latency constraint in (3) or the model occupies too much space in hardware, the model's number of parameters are decreased until the constraints are satisfied. Once the latency/accuracy trade off has been reached, the parameters are transferred to the TNN/ONN networks and used as a starting point (*i.e.*, "bootstrap") for the DRL algorithm.

VI. Experimental Evaluation

In this section we extensively evaluate the performance of *DeepWiERL* and compare its latency performance with a traditional software-based implementation in PyTorch. Furthermore, we demonstrate the performance of *DeepWiERL* on a concrete rate adaptation problem.

A. Software-defined Radio Testbed

Figure 5 shows our experimental testbed. The transmitter has been implemented on a customized software-defined radio (SDR) platform, made up by (i) a Xilinx ZC706 evaluation board, which contains a Xilinx Zynq-7000 system-on-chip (SoC) equipped with two ACM Cortex CPU and a Kintex-7 FPGA; and (ii) an Analog Device FMCOMMS2 evaluation board equipped with a fully-reconfigurable AD9361 RF transceiver and VERT2450 antennas. The receiver is implemented on a less-powerful Zedboard, which is also equipped with an AD9361 transceiver and a Zynq-7000 with a smaller FPGA. In both cases, the platform's software, drivers and data structures have been implemented in the C language, running on top of an embedded Linux kernel. The receiver's side of the OFDM scheme has been implemented on Gnuradio, while the *DeepWiERL Controller* has been implemented in C language for maximum performance and for easy FPGA access through drivers. To measure power consumption, we have used the PMBUS controller from Texas Instrument.

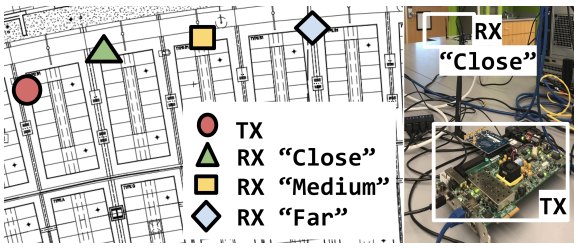


Fig. 5: Experimental testbed for *DeepWiERL*.

B. Randomized Cross-Entropy with Fixed Episodes (RCEF)

To maximize the state-action function, we use a customized variation of the cross entropy (CE) DRL method [39], which we call *randomized CE with fixed episodes* (RCEF). We leverage CE instead of more complex DRL algorithms thanks to its good convergence properties, and because it is well known to perform excellently in problems that do not require complex, multi-step policies and have short episodes with frequent rewards, as in our rate maximization problem. However, we point out that *DeepWiERL* provides a very general framework that can support generalized DRL algorithms, including the more complex Deep Q-Learning.

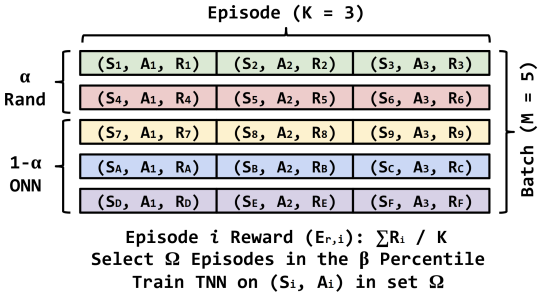


Fig. 6: Randomized CE with Fixed Episodes (RCEF).

Figure 6 summarizes RCEF, which is a model-free, policy-based, on-policy method, meaning that (i) it does not build any model of the wireless transmission; (ii) directly approximates the policy of the wireless node; and (iii) requires fresh spectrum data obtained from the wireless channel. The CE method feeds experience to the wireless node through *episodes*, which is a sequence of spectrum observations obtained from the wireless environment, actions it has issued, and the corresponding rewards. Episodes are of fixed length K and are grouped in a batch of M episodes. At the beginning of each episode, the node can choose to either *explore* the action space with probability α or *exploit* the ONN knowledge with probability $1 - \alpha$ for the duration of the episode. After completion,

episode i is assigned a reward $E_{r,i} = \sum_{i=0}^{K} R_i/K$. After the episodes in the batch are completed, RCEF selects the episodes belonging to the β percentile of rewards and puts them in a set Ω. The TNN is trained on the tuples (S_i, A_i) in Ω.

Notice that since the policy is ultimately a probability distribution over the possible actions, the action decision problem boils down to a classification problem where the amount of classes equals the amount of actions. In other words, after the algorithm has converged, the transmitter only needs to (i) pass a spectrum observation to the ONN, (ii) get the probability distribution from the ONN output, and (iii) select the action to execute using that distribution. Such random sampling adds randomness to the agent, which is especially needed at the beginning to explore the state action space.

C. DeepWiERL vs Software Performance

Our experimental evaluation was specifically targeted at evaluating the latency and energy performance of *DeepWiERL* with respect to a software implementation, and the convergence performance of *S-DMSB* with respect to clean-state approaches. To achieve this goal, we consider a rate adaptation problem where the modulation scheme of an orthogonal frequency division multiplexing (OFDM)-based transmission scheme is optimized based on the current spectrum conditions. Specifically, the spectrum is represented by 128 complex channel taps – corresponding to the FFT size used by the OFDM scheme – which are estimated at the receiver's side using pilots. Therefore, since our CNN accepts only real numbers, we extend each complex number to a set of two real numbers, thus obtaining a state vector of dimension (1, 256). The CNN architecture used for our experimental evaluation includes (i) a Conv-1D layer with N filters of W size, as shown in Equation (4); (ii) a dense layer containing D neurons; and (iii) a softmax layer of three actions, corresponding to BPSK, QPSK and 8PSK modulation schemes, respectively.

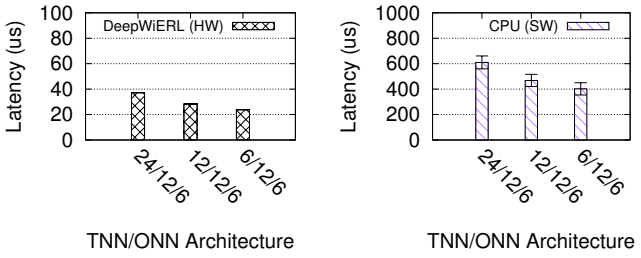


Fig. 7: DeepWiERL (Hardware) vs CPU (Software) DNN latency as a function of Dense Layer Size.

Figure 7 shows the latency comparison between *DeepWiERL* (hardware) and a software implementation of the deep neural network in C++ running on the CPU of our evaluation board. In these experiments, we show results with fixed $N = 12$ and $W = 6$ as we found that D is the parameter that most impacts latency. These results were obtained by setting the FPGA clock frequency to 100 MHz while the RF front-end and the CPU are clocked at 200 MHz and 667 MHz, respectively. The software's latency results were obtained over 100 runs and we also show 90% confidence intervals. The results show that DeepWiERL's latency is about 16x times lower than the software implementation in PyTorch – notice that the scale of the HW graph is 10x less than the SW graph. Notice that according to Equation (3), this implies that *DeepWiERL*'s achievable application rate is 16x greater than PyTorch's, assuming the same buffer is used.

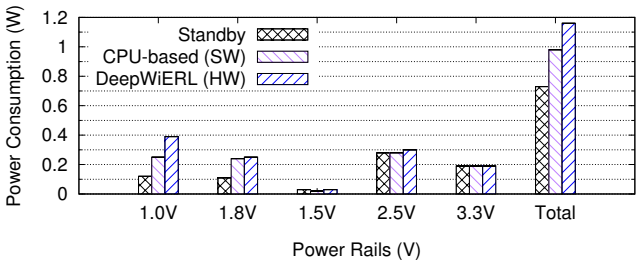


Fig. 8: DeepWiERL (Hardware) vs CPU (Software) Power Consumption for Different Power Rails.

Finally, Figure 8 shows the comparison in terms of power consumption between *DeepWiERL* and the CPU-based implementation, for all the five power rail of the ZC706 evaluation board. The results align with the intuition that *DeepWiERL* should consume more power (1.16W vs 0.98W), due to the involvement of the FPGA. However, as shown in Figure 7, *DeepWiERL* has an order of magnitude less latency than CPU. Therefore, the difference in energy consumption (597.8 μJ vs 42.92 μJ in the case of 24/12/6 model) makes *DeepWiERL* 14x more energy efficient than the CPU-based implementation.

D. System-wide Performance Evaluation

This section shows the experimental results on convergence obtained on the rate maximization problem described earlier. The experiments performed in Figures 9 and 11 were purposely performed in an "in-the-wild" laboratory environment with the presence of several interference coming from nearby wireless devices. Specifically, we choose 2.437G as center frequency, which is the 6th WiFi channel, with a sampling rate of 10 MS/s.

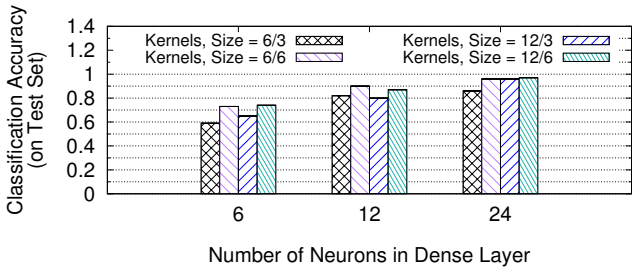
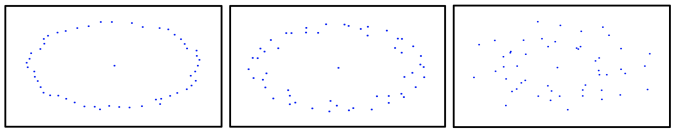


Fig. 9: DNN Accuracy as a function of Dense Layer Size, Number of Kernels and Kernel Size.

Before deployment, we ran S-DMSB to understand which DNN model was appropriate for the problem under consideration. Therefore, we fixed the position of the transmitter and we collected data by changing the position of the receiver to

about 5ft, 15ft, and 40ft distance from the transmitter, so as to have a "*Close*", "*Medium*", and "*Far*" configuration as shown in Figure 5. After data collection, we trained different DNN models to evaluate the accuracy of each model. Figure 10 shows examples of training data obtained using our testbed.



(a) "Close" (b) "Medium" (c) "Far"

Fig. 10: Channel I/Q taps in the "*Close*", "*Medium*" and "*Far*" scenarios.

Figure 9 shows the DNN accuracy as a function of the dense layer size, for different values of N (number of kernels) and W (kernel size). Figure 9 indicates that the dense layer size is the predominant hyper-parameter as it significantly impacts the classification accuracy of the DNN models. This was expected, as more complex features can be learned by the DNN. Furthermore, the number and size of kernels do impact the classification accuracy but to a lesser extent. Since the 24/12/6 model achieves the best performance, we choose this one as our reference DNN model.

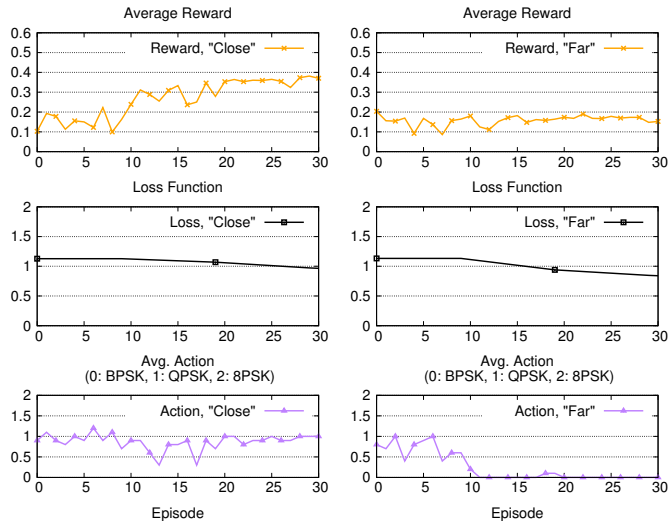


Fig. 11: Reward, Loss and Action Obtained by RCEF as a Function of the Episode.

To evaluate the convergence performance of RCEF in a challenging "in-the-wild" scenario, we consider the *Close* and *Far* configurations and run *DeepWiERL* with the 24/12/6 DNN model starting from a "clean-slate" (*i.e.*, random parameter set). Figure 11 shows the reward, loss function and average action per episode obtained by RCEF, as a function of the number of episodes. The average action is plotted as a real number between 0 and 2, where we assign 0, 1 and 2 to BPSK, QPSK and 8PSK actions, respectively. As far as RCEF is concerned, we fixed K to 10, the batch size B to 10, and α and β to 0.1 and 0.7, respectively.

Intuitively, the expected behavior of the RCEF would be to converge to BPSK and 8PSK modulations in the *Far* and *Close* scenarios, respectively. However, Figure 11 shows that in *Close* the preferred action is to switch to QPSK instead of 8PSK. Indeed, this is due to the fact that in our testbed experiments the QPSK modulation performs better than the other two modulations, and RCEF is able to learn that without human intervention and based just on unprocessed spectrum data. Interestingly, we notice from Figure 11 that RCEF converges faster in *Far* than in *Close*. This is because in the *Far* scenario the 8PSK transmissions always fail and thus the related observations are never recorded at the receiver's side. This, in turn, speeds up convergence significantly – 1 batch vs 2 batches in the *Close* scenario – as also witnessed by the lower loss function values reported in Figure 11.

E. S-DMSB Bootstrapping Performance

The experiments shown in Figure 12 were performed in a controlled environment where the transmitter and receiver are connected through an RF cable and the SINR is changed instantaneously though the introduction of path loss. This was done to (i) explicitly control the RF channel impact and evaluate the convergence performance of RCEF and S-DMSB under repeatable conditions; (ii) determine the optimal reward and action at a given moment in time, which are reported respectively in Figure 12(a) and (b). In these experiments, we changed the SINR level approximately one every 10 episodes, except for the first 20 episodes where we change it every 5 episodes to help RCEF converge better. This was done to emulate highly-dynamic channel conditions between the transmitter and the receiver yet also evaluating the action chosen by the ONN as a function of the SINR.

Figure 12 presents the average reward and action as a function of the episode number obtained by (i) RCEF with a "clean-slate" 6/6/3 DNN in subfigures (c) and (d), (i) RCEF with a "clean-slate" 24/12/6 DNN in subfigures (e) and (f), and (iii) RCEF with a "bootstrapped" 24/12/6 DNN in subfigures (g) and (h), obtained through the S-DMSB method described in Section V-B and with the data collected in Section VI-D.

Figures 12(c) and (d) hint that the 6/6/3 architecture is not able to capture the difference between different spectrum states, as it converges to a fixed QPSK modulation scheme regardless of the SINR levels experienced on the channel. On the other hand, Figures 12(e) and (f) show that the 24/12/6 architecture performs much better in terms of convergence, as it is both able to distinguish between different spectrum states and is able to switch between BPSK and QPSK when the SINR level changes. However, we can see that convergence does not happen until episode 60. Finally, Figures 12(g) and (h) indicates that S-DMSB's bootstrapping procedure is significantly effective in the scenario considered. Indeed, we obtain an increase in average reward of more than 45% with respect to clean-slate RCEF, and 6x speed-up in terms of convergence – indeed, RCEF + S-DMSB converges to the "seesaw" pattern at episode 10, while clean-slate RCEF converges at episode 60.

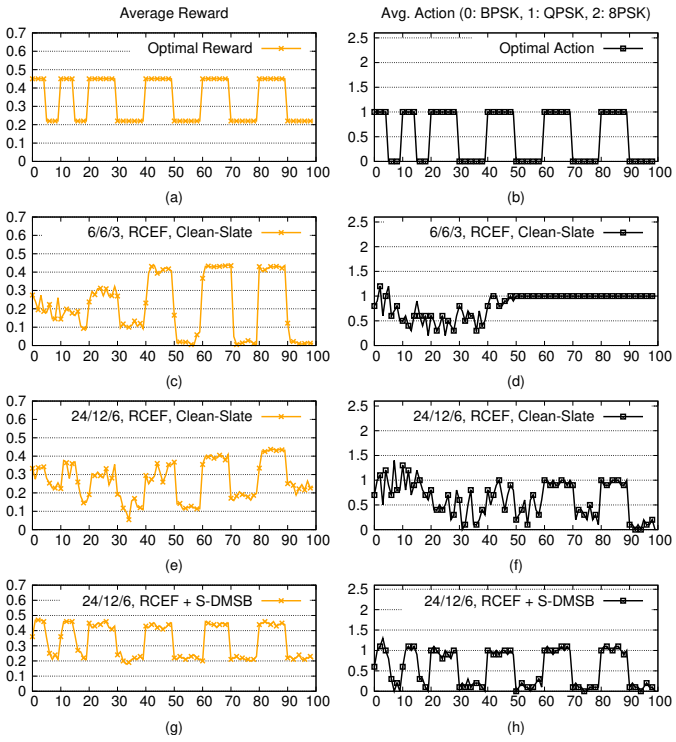


Fig. 12: Reward and Actions obtained by RCEF and S-DMSB.

VII. Concluding Remarks

This section discusses the related work and compares it to ours, and concludes the paper by summarizing the contributions of this paper and the ongoing work.

Related Work and Comparison

Deep reinforcement learning [22] has recently received a remarkable surge of attention. This section summarizes a small part of existing work only – for a recent comprehensive survey on the topic, the reader may refer for example to [40]. As far as cellular networks are concerned, Liu *et al.* [4] proposed *DeepNap*, a DRL- based algorithm for dynamic base station (BS) sleeping control in cellular networks. Wang *et al.* [5] propose a two-layer DRL framework to learn the optimal handover patterns in networks with heterogeneous user mobility. DRL has also been leveraged to improve video quality [15, 17] and to accommodate QoE demands [16].

Much research efforts in DRL for wireless has been devoted to address dynamic spectrum access (DSA) issues [6–9]. Chang *et al.* [9] apply DRL to make secondary users (SUs) in DSA networks learn appropriate spectrum access strategies in a distributed fashion assuming no knowledge of the underlying system statistics. Naparstek and Cohen [8] leverage DRL to formulate a DSA strategy to maximize a given network utility function in a distributed manner without online coordination or message exchanges between users, where at each time slot, each user maps its current state to the spectrum access actions based on a trained DQN. Similarly, the authors in [7] use DRL to learn an access policy from the observed states of all channels when SUs have no knowledge about the channel model. Yu *et al.* [6] design a "universal" Deep-reinforcement Learning Multiple Access (DLMA) protocol, and show that a DRL agent can learn the optimal (*i.e.*, sum throughput or α-fairness) MAC strategy for harmonious co-existence with time-division multiple access (TDMA) and ALOHA nodes.

Recently, the topic of DRL-based resource allocation and management has received significant interest [10–14, 41–43]. Li *et al.* [13] investigate the application of DRL in solving network slicing scenarios and demonstrate the advantage of DRL over several competing schemes through simulations. Sun *et al.* [12] study the problem of resource management in Fog radio access networks. Zhang *et al.* [14] consider the challenges of network heterogeneity, comprehensive QoS goals, and dynamic environments. Xu *et al.* [41] present a DRL-based control framework for traffic engineering, while Feng and Mao [10] tackle the problem of limited backhaul capacity and highly dynamic data rates of users in millimeter-wavelength (mmWave) systems by presenting a DRL scheme that learns blockage patterns and allocate backhaul resources to each user. Jiang *et al.* [42] use DRL to determine the configuration that maximizes the long-term average number of served IoT devices.

The closest work to ours is [34], where the authors proposed a framework to integrate a 2-D convolutional neural network (CNN) in the hardware RF loop. However, our work separates itself from [34] since the authors do not consider DRL and instead strictly focus on supervised learning aspects (*i.e.*, modulation recognition). Indeed, the above work applies deep learning at the receiver's side, while in this paper it is applied at the transmitter's side.

Summary of Contributions and Current Work

This paper's key innovation is *Deep Wireless Embedded Reinforcement Learning* (*DeepWiERL*), a general-purpose hardware-based DRL framework specifically tailored for the IoT and providing support for training and real-time execution of state-of-the-art DRL algorithms. We have (i) modeled the latency/memory design constraints of *DeepWiERL*; (ii) proposed a hardware-based 1-D CNN architecture to infer the current spectrum state; and (iii) proposed a novel DRL bootstrapping technique (S-DMSB) to select a DNN model meeting the hardware/latency requirement and speed-up convergence time. Experimental results have shown that system-level real-time DRL is indeed feasible; that our hardware approach supports up to 16x data rate and 14x less energy consumption than software-based solutions; and that S-DMSB is effective in bringing DRL algorithms to convergence faster and to a better optimal solution.

Acknowledgements

This work is supported by the Office of Naval Research (ONR) and Raytheon Inc. under contracts N00014-18-9-0001 and 4201829283. The views and conclusions contained herein are those of the authors and should not be interpreted as neces-

sarily representing the official policies or endorsements, either expressed or implied, of the ONR or the U.S. Government.

REFERENCES

[1] Ericsson Incorporated, "Internet of Things forecast," https://www.ericsson.com/en/mobility-report/internet-of-things-forecast, 2019.

[2] L. Da Xu, W. He, and S. Li, "Internet of Things in Industries: A Survey," *IEEE Transactions on Industrial Informatics*, vol. 10, no. 4, pp. 2233–2243, 2014.

[3] F. Restuccia, S. D'Oro, and T. Melodia, "Securing the Internet of Things in the Age of Machine Learning and Software-defined Networking," *IEEE Internet of Things Journal*, vol. 5, no. 6, pp. 4829–4842, 2018.

[4] J. Liu, B. Krishnamachari, S. Zhou, and Z. Niu, "DeepNap: Data-Driven Base Station Sleeping Operations Through Deep Reinforcement Learning," *IEEE Internet of Things Journal*, vol. 5, no. 6, pp. 4273–4282, Dec 2018.

[5] Z. Wang, L. Li, Y. Xu, H. Tian, and S. Cui, "Handover Control in Wireless Systems via Asynchronous Multiuser Deep Reinforcement Learning," *IEEE Internet of Things Journal*, vol. 5, no. 6, pp. 4296–4307, 2018.

[6] Y. Yu, T. Wang, and S. C. Liew, "Deep-Reinforcement Learning Multiple Access for Heterogeneous Wireless Networks," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 6, pp. 1277–1290, June 2019.

[7] S. Wang, H. Liu, P. H. Gomes, and B. Krishnamachari, "Deep Reinforcement Learning for Dynamic Multichannel Access in Wireless Networks," *IEEE Transactions on Cognitive Communications and Networking*, vol. 4, no. 2, pp. 257–265, 2018.

[8] O. Naparstek and K. Cohen, "Deep Multi-user Reinforcement Learning for Distributed Dynamic Spectrum Access," *IEEE Transactions on Wireless Communications*, vol. 18, no. 1, pp. 310–323, 2019.

[9] H.-H. Chang, H. Song, Y. Yi, J. Zhang, H. He, and L. Liu, "Distributive Dynamic Spectrum Access through Deep Reinforcement Learning: A Reservoir Computing Based Approach," *IEEE Internet of Things Journal*, 2018.

[10] M. Feng and S. Mao, "Dealing with Limited Backhaul Capacity in Millimeter-Wave Systems: A Deep Reinforcement Learning Approach," *IEEE Communications Magazine*, vol. 57, no. 3, pp. 50–55, 2019.

[11] Y. He, N. Zhao, and H. Yin, "Integrated Networking, Caching, and Computing for Connected Vehicles: A Deep Reinforcement Learning Approach," *IEEE Transactions on Vehicular Technology*, vol. 67, no. 1, pp. 44–55, Jan 2018.

[12] Y. Sun, M. Peng, and S. Mao, "Deep Reinforcement Learning-Based Mode Selection and Resource Management for Green Fog Radio Access Networks," *IEEE Internet of Things Journal*, vol. 6, no. 2, pp. 1960–1971, April 2019.

[13] R. Li, Z. Zhao, Q. Sun, C. I, C. Yang, X. Chen, M. Zhao, and H. Zhang, "Deep Reinforcement Learning for Resource Management in Network Slicing," *IEEE Access*, vol. 6, pp. 74 429–74 441, 2018.

[14] H. Zhang, W. Li, S. Gao, X. Wang, and B. Ye, "ReLeS: A Neural Adaptive Multipath Scheduler based on Deep Reinforcement Learning," *Proc. of IEEE Conference on Computer Communications (INFOCOM)*, 2019.

[15] H. Pang, C. Zhang, F. Wang, J. Liu, and L. Sun, "Towards Low Latency Multi-viewpoint 360° Interactive Video: A Multimodal Deep Reinforcement Learning Approach," *Proc. of IEEE Conference on Computer Communications (INFOCOM)*, 2019.

[16] F. Wang, C. Zhang, F. Wang, J. Liu, Y. Zhu, H. Pang, and L. Sun, "Intelligent Edge-Assisted Crowdcast with Deep Reinforcement Learning for Personalized QoE," *Proc. of IEEE Conference on Computer Communications (INFOCOM)*, 2019.

[17] Y. Zhang, P. Zhao, K. Bian, Y. Liu, L. Song, and X. Li, "DRL360: 360-degree Video Streaming with Deep Reinforcement Learning," *Proc. of IEEE Conference on Computer Communications (INFOCOM)*, 2019.

[18] L. Zhang, J. Tan, Y. Liang, G. Feng, and D. Niyato, "Deep Reinforcement Learning based Modulation and Coding Scheme Selection in Cognitive Heterogeneous Networks," *IEEE Transactions on Wireless Communications*, pp. 1–1, 2019.

[19] J. Jagannath, N. Polosky, A. Jagannath, F. Restuccia, and T. Melodia, "Machine Learning for Wireless Communications in the Internet of Things: A Comprehensive Survey," *Ad Hoc Networks*, vol. 93, p. 101913, 2019.

[20] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning," *arXiv preprint arXiv:1312.5602*, 2013.

[21] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, "Mastering the Game of Go Without Human Knowledge," *Nature*, vol. 550, no. 7676, p. 354, 2017.

[22] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT press, 2018.

[23] I. E. Telatar and D. N. C. Tse, "Capacity and Mutual Information of Wideband Multipath Fading Channels," *IEEE Transactions on Information Theory*, vol. 46, no. 4, pp. 1384–1400, 2000.

[24] R. S. Sutton, A. G. Barto *et al.*, *Introduction to Reinforcement Learning*. MIT press Cambridge, 1998, vol. 135.

[25] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep Learning*. MIT press Cambridge, 2016, vol. 1.

[26] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel, "Benchmarking Deep Reinforcement Learning for Continuous Control," in *International Conference on Machine Learning*, 2016, pp. 1329–1338.

[27] L. Baird, "Residual Algorithms: Reinforcement Learning With Function Approximation," in *Machine Learning Proceedings 1995*. Elsevier, 1995, pp. 30–37.

[28] "Analysis of temporal-diffference learning with function approximation."

[29] G. J. Gordon, "Stable Function Approximation in Dynamic Programming," in *Machine Learning Proceedings 1995*. Elsevier, 1995, pp. 261–268.

[30] M. Riedmiller, "Neural Fitted Q Iteration–First Experiences With a Data Efficient Neural Reinforcement Learning Method," in *European Conference on Machine Learning*. Springer, 2005, pp. 317–328.

[31] H. Van Hasselt, A. Guez, and D. Silver, "Deep Reinforcement Learning with Double Q-Learning," in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[32] L.-J. Lin, "Self-Improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching," *Machine Learning*, vol. 8, no. 3-4, pp. 293–321, 1992.

[33] R. F. Molanes, J. J. Rodrguez-Andina, and J. Faria, "Performance Characterization and Design Guidelines for Efficient Processor - FPGA Communication in Cyclone V FPSoCs," *IEEE Transactions on Industrial Electronics*, vol. 65, no. 5, pp. 4368–4377, May 2018.

[34] F. Restuccia and T. Melodia, "Big Data Goes Small: Real-Time Spectrum-Driven Embedded Wireless Networking Through Deep Learning in the RF Loop," *Proc. of IEEE Conference on Computer Communications (INFOCOM)*, 2019.

[35] N. E. West and T. O'Shea, "Deep Architectures for Modulation Recognition," in *Proc. of IEEE International Symposium on Dynamic Spectrum Access Networks (DySPAN)*, 2017.

[36] T. J. O'Shea, T. Roy, and T. C. Clancy, "Over-the-Air Deep Learning Based Radio Signal Classification," *IEEE Journal of Selected Topics in Signal Processing*, vol. 12, no. 1, pp. 168–179, 2018.

[37] K. Weiss, T. M. Khoshgoftaar, and D. Wang, "A Survey of Transfer Learning," *Journal of Big Data*, vol. 3, no. 1, p. 9, 2016.

[38] F. Winterstein, S. Bayliss, and G. A. Constantinides, "High-Level Synthesis of Dynamic Data Structures: A Case Study Using Vivado HLS," in *Proc. of International Conference on Field-Programmable Technology (FPT)*, Kyoto, Japan, 2013, pp. 362–365.

[39] I. Szita and A. Lörincz, "Learning Tetris Using the Noisy Cross-entropy Method," *Neural Computation*, vol. 18, no. 12, pp. 2936–2941, 2006.

[40] N. C. Luong, D. T. Hoang, S. Gong, D. Niyato, P. Wang, Y. Liang, and D. I. Kim, "Applications of Deep Reinforcement Learning in Communications and Networking: A Survey," *IEEE Communications Surveys Tutorials*, vol. 21, no. 4, pp. 3133–3174, 2019.

[41] Z. Xu, J. Tang, J. Meng, W. Zhang, Y. Wang, C. H. Liu, and D. Yang, "Experience-driven Networking: A Deep Reinforcement Learning based Approach," in *Proc. of IEEE Conference on Computer Communications (INFOCOM)*, 2018.

[42] N. Jiang, Y. Deng, A. Nallanathan, and J. A. Chambers, "Reinforcement Learning for Real-Time Optimization in NB-IoT Networks," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 6, pp. 1424–1440, June 2019.

[43] Z. Xu, J. Tang, C. Yin, Y. Wang, and G. Xue, "Experience-Driven Congestion Control: When Multi-Path TCP Meets Deep Reinforcement Learning," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 6, pp. 1325–1336, June 2019.