# Real-Time Channel Management in WLANs: Deep Reinforcement Learning versus Heuristics

Ovidiu Iacoboaiea, Jonatan Krolikowski, Zied Ben Houidi, Dario Rossi
Paris Research Center Huawei Technologies Co. Ltd
{ovidiu.iacoboaiea, jonatan.krolikowski, zied.ben.houidi, dario.rossi}@huawei.com

***Abstract*—Today's WLANs rely on a centralized Access Controller (AC) entity for managing distributed wireless Access Points (APs) to which user devices connect. The availability of real-time analytics at the AC opens the possibility to automate the allocation of scarce radio resources, continuously adapting to changes in traffic demands. Often, the allocation problem is formulated in terms of weighted graph coloring, which is NP-hard, and custom heuristics are used to find satisfactory solutions. In this paper, we contrast solutions that are based on (and even improve) state of the art heuristics to a data-driven solution that leverages Deep Reinforcement Learning (DRL). Based on both simulation results as well as experiments in a real deployment, we show that our DRL-based scheme not only learns to solve the complex combinatorial problem in bounded time, outperforming heuristics, but it also exhibits appealing generalization properties, e.g. to different network sizes and densities.**

## I. Introduction

IEEE 802.11 Wireless Local Area Networks (WLANs) are the preferred communication medium for a wide variety of large-scale corporate and campus networks. To manage a large fleet of distributed wireless Access Points (APs), modern WLANs adopt a centralized architecture, where an Access Controller (AC) is responsible for both real-time monitoring and control of the APs. For many medium to large scale deployments, high density puts significant stress on scarce radio resources, making channel and bandwidth allocation across APs difficult. In recent times, scalable BigData analytics complemented the AC with a computing backend to collect, analyze and display several Key Performance Indicators (KPIs), assisting network administrators in operation and management. This opened the way towards centralised continuous resources reallocation to cope with sudden changes in traffic demands (e.g. reacting to a flash crowd) [1]–[3].

Yet, most academic research in the last decades targeted distributed and cooperative radio resource allocation [4]–[8] for wireless mesh networks, that however did not yield to successful and pervasive deployments. A significant body of literature also exists for the centralized version of the problem [3], [9]–[17] and has been with almost no exception formulated as an NP-hard weighted graph coloring problem, generally approached with heuristics.

In this paper, we deal with a classical control problem that can be tackled (i) building on state of the art heuristics, or (ii) leveraging recent advances in Deep Reinforcement Learning (DRL). We set out to assess the limits and advantages of each. As for a heuristic solution to weighted graph coloring, we provide an innovative dynamic local search (*dynLS*) solution that improves over the state of the art *TurboCA* [16] algorithm currently implemented in the Meraki product series. As for DRL-based solutions, we address several methodological challenges and design an architecture (*net2seq*) capable of not only tackling a hard combinatorial optimization problem, but also generalizing to networks of variable size and density (transfer learning) with a furthermore bounded inference time (below 1 second). Particularly, our results testify that a clear advantage of data-driven techniques is their intrinsic ability to learn: e.g., by feeding a DRL-based agent with traffic-related KPIs, it allows for learning to predict future demands and adapt its solution as a consequence. This gives DRL approaches an intrinsic advantage compared to plain heuristics, where forecasting mechanisms should be explicitly designed and accounted for. Summarizing our main contributions:

- We design *dynLS* (a novel real-time local-search based solution that improves over the state of the art *TurboCA* [16]) and introduce *net2seq* (a novel sequence-based neural network architecture that learns a sequential policy built as a set of sub-actions). In particular, our DRL design augments the traditional dual-network actor-critic architecture with a *selector* network, that learns an approximation of the $Q$ function and picks the best action from a restrained set of possible actions proposed by the Actor (as opposed to the entire space, which would explode the inference time).
- We systematically compare *net2seq*, *dynLS* and *TurboCA*, by both *simulation* as well as *real deployment*. By simulation, we discover that while *dynLS* performance decreases with increasing problem size, *net2seq* is instead not impacted by the size of the network, which suggests DRL-based methods to be interesting for scaling up the solutions of combinatorial problems. Furthermore, we deploy the algorithms in a real network of 34 APs for about one month, whose preliminary benchmark confirms the results gathered via simulation.

The rest of the paper is organized as follows. Related work is covered in Sec. II, and our MDP-based problem formulation is presented in Sec. III. Design of *net2seq* and *dynLS* are introduced in Sec. IV and Sec. V, respectively. Sec. VI evaluates the algorithms in simulation settings and via real deployment, and Sec. VII summarizes our findings.

## II. Related work

We first examine existing WLAN centralized allocation schemes (Sec. II-A), to identify the state of the art reference baseline, upon which our *dynLS* improves. Then we review a recent trend in the machine learning community that we leverage in this work, namely the use of

ISBN 978-3-903176-39-3

TABLE I: Overview of centralized allocation schemes

| Ref | CA† | LA† | B† | Metric† | Method◇ |
|---|---|---|---|---|---|
| [9] | | | | $\sum$ interf | nnI+ns |
| [10] | ✓ | | | Conflict free clients | nnI |
| [11] | ✓ | ✓ | | Weighted channel separation | nnSA |
| [12] | ✓ | ✓ | | Delay | SDP |
| [13] | ✓ | ⋆ | | $\sum$ interf | C+nnI |
| [14] | ✓ | ✓ | | Throughput redux (fairness) | nnI |
| [15] | | ✓ | | Free AT (sum, min, fairness) | nnI |
| [16] | ✓ | ✓ | ✓ | Fair AT - reconfig cost | nnI+NC |
| [17] | | | | $\sum$ channel util. | ILP |

† *CA/LA*: client/load-aware, ⋆: active clients, *B*: bonding, *AT*: air time
◇ *nnI*: node-by-node improvement, *NS*: neighbor swap, *C*: clustering, *ILP*: integer linear programming, *SDP*: semi-definite programming, *nnSA*: node-by-node simulated annealing, *NC*: neighborhood clearance

Deep Reinforcement Learning (DRL) techniques to solve combinatorial optimization problems (Sec. II-B).

### A. WLAN Channel allocation and bonding

A substantial body of literature has targeted the problem of centralized channel (and to a lesser extent bandwidth) allocation, as surveyed in [18]. We compactly present the literature landscape in Tab. I, that reports relevant aspects (such as: topology, load-awareness, support of bonding, objective metric, and optimization method) on which literature differs.

The problem is often formulated as weighted graph coloring [9]–[14]: the color represents the channel, and the weight some property such as the number of active clients [13], extra transmission delays incurred due to interference between each pair of APs [12], or the reduction in throughput caused by interference between vertices [14]. As illustrated in Tab. I, some work tackles the problem leveraging simulated annealing [11], Integer Linear Programming [17] or Semi Definite Programming (SDP) relaxation [12], while the majority of existing solutions are local-search (LS) based heuristics, that iteratively improve node-by-node (nnI) [9], [10], [13]–[16]. To circumvent node-by-node iteration, additional procedures are often necessary to escape local minima: neighbor swapping (NS) has been introduced in seminal work in the late 90s [9] and neighborhood clearance (NC) is used in the *TurboCA* [16] state of the art solution implemented by Meraki products nowadays.

In line with prior work, we assume knowledge of the topology, our problem is by definition load-aware, and our solutions support bonding as in [16]. Our objective (regret) function depends on the overall network interference as common in the literature [9], [13], [17] and explicitly takes into account the reconfiguration cost, which is less popular [16] but equally important. In addition, the edge-by-edge design of our *dynLS* algorithm avoids local minima by design, eliminating the need for additional node- [9] or neighborhood [16] level procedures.

### B. DRL to solve graph-based combinatorial problems

The machine learning community recently started exploring the use of Deep Reinforcement Learning (DRL) as an alternative to classic heuristics for solving combinatorial optimization problems such as the Traveling Salesman Problem (TSP) [19]–[26], Vehicle Routing (VRP) [22], [23] and, to a lesser extent, Graph Colouring (GCP) [27], [28].

One popular approach is to use sequential algorithms: Pointer Networks [19] are first to tackle TSP by leveraging a Long Short Time Memory (LSTM)-based encoder-decoder that works for variable size instances. The encoder extracts features from input city coordinates, and the decoder uses LSTMs to maintain a context, which together with attention-based pointers directed at the inputs, selects the order of node traversal. Bello et al. [20] build on [19], proposing an Actor-Critic RL framework instead of supervised learning. Vinyals et al. [29] reveal that sequence-based approaches are sensitive to input order. Consequently, Nazari et al. [22] extend [20] to propose a VRP solution that drops the sequential encoder step. As sequential operations are time-consuming (most notably for the encoder), Vaswani et al. [30] introduce Transformers (self-attention mechanisms which process all input information in parallel). The latter became the state of the art in sequential Natural Language Processing (NLP) and have also been applied to TSP and VRP [23].

All the above approaches do not use any information on neighbor relations, which is inconvenient for GCP. Graph Neural Networks (GNNs) [31], [32] seem to better fit to process topologically structured data and have been used in [21], [28], [33], [34]. Thus we also resort to GNNs in our work. For instance, DRL is used by Naderializadeh et al. [33] for resource management in 5G networks, but in a distributed multi-agent approach, thus the architecture and application context are rather different. Dai et al. [21] use *struct2vec* [34] graph embeddings to solve TSP, Maximum Cut and Minimum Vertex Cover problems. They rely on Deep Q-learning Networks (DQN), where the Q-function is used as a sequential decoder to decide *sub-actions,* i.e. next visited node. Our *net2seq* approach shares similarities as it uses a sequential decoder, but differs in learning a *stochastic policy* which, evaluating multiple actions through our proposed Actor-Critic-Selector architecture, is helpful in avoiding local minima.

Finally, similarly to our work, Nakashima et al. [28] tackles WLAN channel allocation as a GCP problem. However, they leverage a Double DQN using a Graph Convolutional Network (GCN) [35] to estimate the Q-function – which fails to handle variable-size instances, unlike our *net2seq* proposal.

## III. Problem Description

### A. WLAN channel management and notation

Given an Access Controller (AC) managing a set of Access Points (APs), our goal is to design an algorithm that lets the AC continuously reassign channel and bandwidth resources to the APs in order to best handle the traffic demands. Intuitively, loaded neighbouring APs should be allocated as disjoint radio resources as possible in order to reduce mutual interference. We now formalize the system model, focusing on the 5GHz band for simplicity (though this can also be applied to orthogonal channels in the 2.4GHz band).

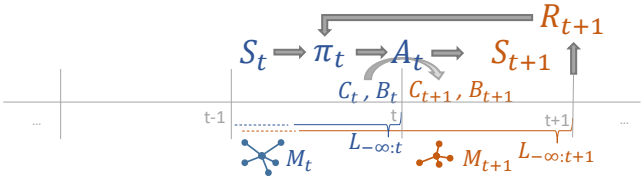


Fig. 1: Temporal view of MDP problem formulation

Consider a WLAN comprising $N_{AP}$ APs, indexed by $i \in \mathcal{N}_{AP} = [1, N_{AP}] \subset \mathbb{N}$. We denote the primary channel allocated to the $i$-th AP as $c_i \in \mathcal{C} = [0, N_C - 1] \subset \mathbb{N}$. Early standards allowed only the use of single 20 MHz-wide channels. Nowadays they further allow for *channel bonding* of up to 8 channels (i.e. 160 MHz). We index the bonding configuration with $b_i \in \mathcal{B} = [0, N_B - 1] \subset \mathbb{N}$, which implies that $2^{b_i}$ channels are aggregated together. We let $\psi(c_i, b_i) : \mathcal{C} \times \mathcal{B} \to \{0,1\}^{N_C}$ represent a bitmask encoding of the set of all channels used, $\psi_k$ is 1 if and only if the $k$-th channel is used when an AP $i$ is configured with $(c_i, b_i)$.

Define the load as the ratio of time-resources necessary to transmit and receive integrally on one channel. Let $\ell_i \in \mathcal{L} = \mathbb{R}^+$ be the current load of $i$-th AP. E.g. if AP $i$ requires to occupy 70% of two bonded channels, then $\ell_i = 0.7 \cdot 2 = 1.4$.

At network level, the AC builds an asymmetric AP neighborhood map $m = (m_{i,j})_{i,j \in \mathcal{N}_{AP}}$, where $m_{i,j} = 1$ represents the fact that AP $i$ is within the interference range of AP $j$, else $m_{i,j} = 0$. In practice, this is commonly obtained by measuring inter-AP received signal strength and considering a threshold to establish neighborhood (e.g. we use -82 dBm).

An AP $i$ is prevented from transmitting (by medium access control) on the account of an AP $j$ when: (i) AP $i$ is in the interfering range of AP $j$, (ii) AP $j$ is transmitting and (iii) any of their primary or bonded channels overlap. Under the simplifying assumption that traffic is equally split on all bonded channels $\ell_j 2^{-b_j}$, we define the interference to AP $i$ from AP $j$ on channel $k$ as:

$$i_{i,j,k} = m_{i,j} \ \ell_j 2^{-b_j} \ \psi_k(c_j, b_j) \tag{1}$$

The AC objective is to reduce interference and balance the channel utilization over the whole network. We thus define the *maximum channel utilization* $u^+$ as the sum of interference seen by an AP $i$ on the worst channel it uses:

$$u_i^+ = \max_k \psi_k(c_i, b_i) \sum_j i_{i,j,k} \tag{2}$$

While our methods are not bound to any specific objective function, whose detailed definition is deferred to Sec.VI, we point out a reasonable AC goal is to minimize (any convex function of) $u_i^+$.

### B. Markov Decision Processes (MDP)

We model the problem as an MDP, that we describe with the help of Fig. 1. The goal is to learn a *policy* for selecting an *action* given a current *state* aimed to minimizes a *discounted regret*, where system transition to future states is modeled through a *transition kernel*.

*a) State and Action Spaces:* A state written as $s = (m, c, b, (\ell, \ell[1], ..., \ell[N_H]))$, is composed of: the current topology $m$, the current configuration $c$ and $b$, and the current and historical load up to $N_H$ past time instances $\ell[h]$ for $h \in \mathcal{N}_H = \{0, ...N_H\}$ (note $\ell[0] \sim \ell$). The historical load is included to ensure the Markov property. All components are seen as stochastic processes on some probability space. The correspondent state process at time $t$ is $S_t = (M_t, C_t, B_t, (L_t, ..., L_{t-N_H}))$, where each component is a random process in the domains $\mathcal{M}^{N_{AP}N_{AP}}$, $\mathcal{C}^{N_{AP}}$, $\mathcal{B}^{N_{AP}}$ and $\mathcal{L}^{N_{AP}N_H}$, respectively. The combinatorial explosion of the state space is obvious. An action is then written as $a = (c^a, b^a)$ with $c^a = (c_i^a)_{i \in \mathcal{N}_{AP}}$ and $b^a = (b_i^a)_{i \in \mathcal{N}_{AP}}$ representing channel and respectively bonding re-allocations. Let $A_t$ be the action process in the space $\mathcal{A} = \mathcal{C}^{N_{AP}} \times \mathcal{B}^{N_{AP}}$ of size $\text{card}(\mathcal{A}) = (N_C N_B)^{N_{AP}}$.

*b) Transition Kernel:* Let us consider $s' = (m', c', b', (\ell', \ell'[1], ..., \ell'[N_H]))$ the future state, following state $s$ after taking action $a = (c_a, b_a)$. In part, the future state is deterministic and assuming invariable topology, the only random component is the future load, and the transition probability is:

$$\begin{aligned}\mathbb{P}(s'|s,a) = \mathbb{I}\{m' = m\} \ \mathbb{I}\{c' = c^a\} \ \mathbb{I}\{b' = b^a\} \\ \mathbb{I}\{\ell'[1:N_h] = \ell[0:N_h - 1]\} \ \mathbb{P}(\ell'|s,a)\end{aligned} \tag{3}$$

where $\mathbb{I}$ is the indicator function: $\mathbb{I}\{true\} = 1$, $\mathbb{I}\{false\} = 0$.

*c) Regret:* Let $R_t$ be the process of instantaneous regrets and $r(s,a) \in \mathbb{R}$ be the regret of being in state $s$ and taking action $a$, i.e., $r(s,a) = \mathbb{E}[R_{t+1}|S_t = s, A_t = a]$, that is the quantity MDP seeks at minimizing. While we defer the specific regret definitions to Sec.VI, we point out that as in [16] we introduce two regret components: a state-regret $R^s$ taking into account network performance, and a reconfiguration-regret $R^r$ to avoid excessive updates:

$$R_{t+1} = R_{t+1}^s + R_{t+1}^r \tag{4}$$

*d) Policies and Value Functions:* A policy $\pi$ maps states to a probability distributions over the action space $\mathcal{A}$, i.e. $\pi(a|s) = \mathbb{P}_\pi(A_t = a|S_t = s)$. The optimal policy $\pi^*$ is the one preferring actions that minimize the classic value functions [36] which estimate the sum discounted regret given for any starting state and action:

$$V_\pi(s) = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} | S_0 = s\right] \tag{5}$$

$$Q_\pi(s,a) = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} | S_0 = s, A_0 = a\right] \tag{6}$$

where $\gamma \in [0,1]$ is the discount factor and $\mathbb{E}_\pi$ is the expectation given that the followed policy is $\pi$.

## IV. Deep Reinforcement Learning (*net2seq*)

Computing the optimal policy $\pi^*$ in a closed form is not possible. The same applies for tabular methods because the state space $\mathcal{S}$ is not tractable as it scales exponentially with the number of APs. We hence focus on the restricted class of parameterized policies $\pi_\theta$, approximating $\pi^*$, and set out to learn its parameters $\theta$ leveraging Deep Reinforcement Learning (DRL) with Gradient Policy Iteration [36] – which

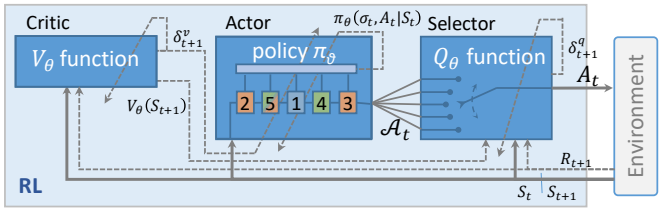


Fig. 2: DRL: Interaction of Actor, Critic and Selector

implies there is no guarantee to find an optimal policy, but this is accepted in practice if results are good and coherent.

We take a novel approach complementing the Actor-Critic architecture [36] by a *Selector network* (Fig. 2) used only at inference time to pick the best action among multiple tryouts proposed by the Actor. This reduces the risk of local minima. The three components are built using Deep Neural Networks (DNNs). We provide the architectural details in the following (illustrated in Fig. 3).

### A. Sequential Actor Network ($\pi$)

Sampling $\pi_\theta$ is not straightforward, as the action space scales exponentially with the number of APs $(N_C N_B)^{N_{AP}}$. Similarly to recent DRL-based combinatorial optimization [23], we factor the policy $\pi$ to *sample it sequentially*: at each sequence step $\tau \in \mathcal{N}_{AP}$, a sub-action $a_{n_\tau}$ is selected for a not yet reconfigured AP indexed $n_\tau$. Grouping all sub-actions, we get an ordered list of actions $(a_{n_1}, \ldots, a_{n_{N_{AP}}})$ for the corresponding AP sequence $\sigma = (n_1, \ldots, n_{N_{AP}}) \in \mathcal{P}$, where $\mathcal{P}$ is the set of all possible AP permutations of the set $\{1, \ldots, N_{AP}\}$. Since the size $N_{AP} N_C N_B$ of the sub-action space scales linearly with the number of APs, considering all the $N_{AP}$ sequence steps, the sampling complexity scales quadratically as $N_{AP}^2 N_C N_B$. We thus rewrite the policy as:

$$\pi(a|s) = \sum_{\sigma \in \mathcal{P}} \pi\left(n_{N_{AP}}, \ldots, n_1, a_{n_{N_{AP}}}, \ldots, a_{n_1} | s\right) \quad (7)$$

$$= \sum_{\sigma \in \mathcal{P}} \prod_{\tau=1}^{N_{AP}} \pi\left(n_\tau, a_{n_\tau} | s, n_{1:\tau-1}, a_{n_{1:\tau-1}}\right) \quad (8)$$

where, with slight abuse of notation, we reuse $\pi$ to denote also the step-wise policy. Note that the order $\sigma$ does not impact how the action $a$ is applied on the environment. The sequential process is sketched in Fig.2: first AP2 is assigned the *orange* channel, then AP5 is configured with bonding two channels *orange-green* and so on until an action $a$, i.e. a full configuration for all APs, is proposed.

At each step the sub-action is selected by means of a DNN architecture. The DNN produces a probability distribution (of size $N_{AP} N_C N_B$) across possible sub-actions, i.e., the step-wise policy $\pi_\theta$, which is randomly sampled. As a first step towards a solution transferable to any network size, note that the same DNN is re-used sequentially.

Training the DNN with DRL is expected to increase the probability of selecting the best actions, i.e. the actions that have yielded lower regret. As shown in Fig.2, the probabilities of all sampled sub-action are collected and used to update the policy parameters $\theta$. We now delve into the specific DNN architecture with the help of Fig.3.

*a) Node-pair features $f^{(recv)}$ and $f^{(neigh)}$:* The Actor DNN takes two types of features as input. The first input features $f^{(recv)}$ represent the *interference received* by each AP on each channel, as illustrated in Fig. 4 for AP0, where clearly we expect the DNN to learn that the lower the interference, the better. As the interference depends on channel configuration which changes when we advance over the steps of the sequence $\tau$, we provide the DNN with interference matrices that report distinctively on the already reconfigured APs $\mathcal{N}^\tau = \{n_1, \ldots n_\tau\}$ and the remaining ones:

$$f^{<\tau\ (recv)}_{i,j,k,h} = \mathbb{I}_{\{j \in \mathcal{N}^\tau_{-i}\}} m_{i,j} \ell_j[h]\ 2^{-b_j} \psi_k(a^c_j, a^b_j) \quad (9)$$

$$f^{>\tau\ (recv)}_{i,j,k,h} = \mathbb{I}_{\{j \notin \mathcal{N}^\tau_{+i}\}} m_{i,j} \ell_j[h]\ 2^{-b_j} \psi_k(c_j, b_j) \quad (10)$$

where $i, j \in \mathcal{N}_{AP}$, ($j \sim$ the neighbour) $k \in \mathcal{C}$, $h \in \mathcal{N}_H$, $\mathcal{N}^\tau_{-i} = \mathcal{N}^\tau \backslash \{i\}$ and $\mathcal{N}^\tau_{+i} = \mathcal{N}^\tau \cup \{i\}$.

Selecting a channel configuration will also reversely impact the neighbours. For example, in Fig. 4, if we were to allocate a channel to AP0 based only on the *received* interference above, then we may chose the green one, which would severely damage AP1's performance. Such situations occur primarily due to asymmetry, as in this example AP1 suffers from AP0 interference, but not vice-versa. We hint that when picking the channel for AP0 it is useful to be aware that AP1 is using the green channel cumulating 60% channel usage from all his neighbours except AP0. Thus, the second input feature $f^{(neigh)}$ extends decisions to considering the neighbours interference situation:

$$f^{<\tau\ (neigh)}_{i,j,k,h} = \mathbb{I}_{\{j \in \mathcal{N}^\tau_{-i}\}} m_{j,i} \sum_{x \in \mathcal{N}^\tau_{-i}} f^{<\tau\ (recv)}_{j,x,k,h} \psi_k(a^c_j, a^b_j)$$
$$f^{>\tau\ (neigh)}_{i,j,k,h} = \mathbb{I}_{\{j \notin \mathcal{N}^\tau_{+i}\}} m_{j,i} \sum_{x \notin \mathcal{N}^\tau_{+i}} f^{>\tau\ (recv)}_{j,x,k,h} \psi_k(c_j, b_j) \quad (11)$$

where $i, j \in \mathcal{N}_{AP}$ ($j \sim$ the neighbour), $k \in \mathcal{C}$, $h \in \mathcal{N}_H$.

*b) Aggregation of Variable Size Neighborhood :* The so far obtained feature matrices $f^{>\tau(tag)}$, and $f^{<\tau(tag)}$ with $tag \in \{recv, neigh\}$ have four dimensions each $N_{AP} \times N_{AP} \times N_C \times (1 + N_H)$, specifically: the number of APs, number of channel, number of neighboring APs and length of load history. We remark that as the number of neighbors is a variable-sized input, we need a DNN capable of handling any size. As exemplified in Fig.3 for $f^{<\tau(recv)}$, we first apply a Dense layer with a ReLU activation that will project the information per neighbor on a higher dimensional space of *fixed size* $N_f = 8$ , and sum over neighbors. We obtain matrices with 3 dimensions each, and particularly a third dimension with fixed size $N_g = N_f$. Denote the intermediate node-wise output of this layer as $g^{>\tau(tag)}$and $g^{<\tau(tag)}$ with $tag \in \{recv, neigh\}$, each of shape $N_{AP} \times N_C \times N_g$.

The above information is complemented with AP-level information about the load $\ell$ and the initial channel configuration $(c, b)$, that we denote as by $g^{(comp)}$ which is of shape $N_{AP} \times N_C \times N'_g$ where $N'_g = (1 + N_H) + 1 + N_B$. Concatenating previous elements along the 3rd dimension, we obtain $g_\tau$ with overall shape $N_{AP} \times N_C \times (4N_g + N'_g)$ .

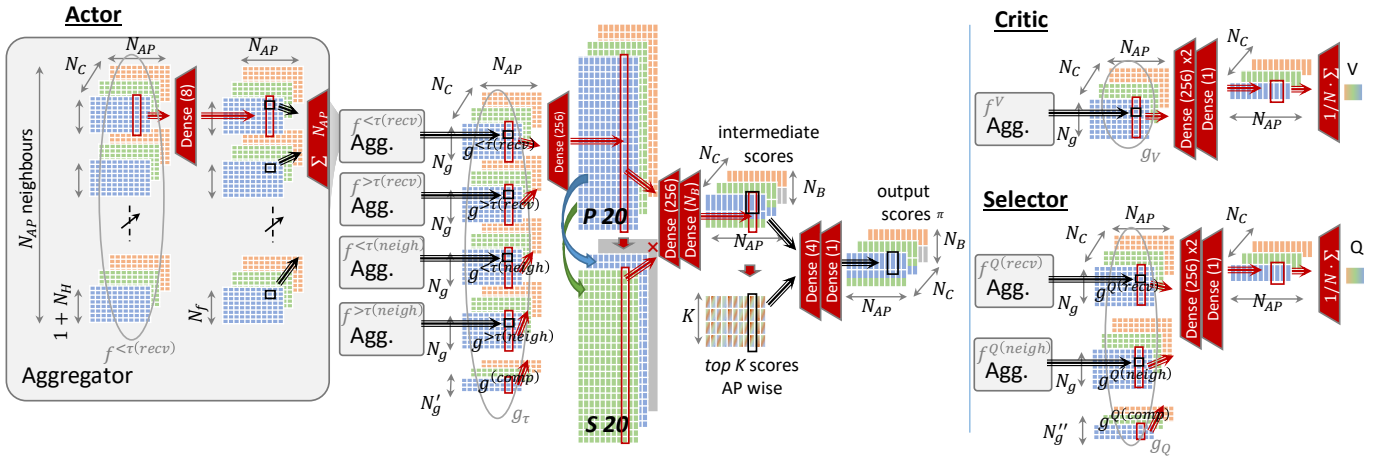


Fig. 3: DNN Architectures: Actor (left, Sec.IV-A), Critic (top-right, Sec.IV-B) and Selector (bottom-right, Sec.IV-C) networks

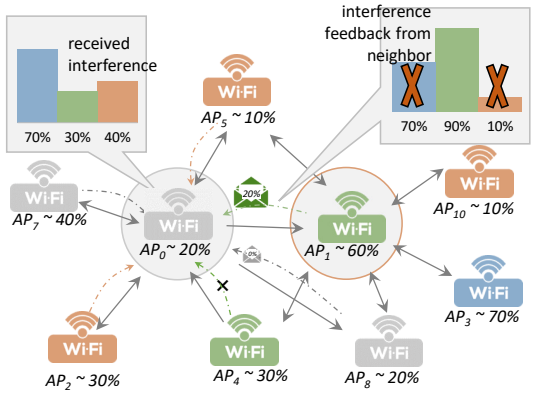


Fig. 4: Illustrative examples for Actor DNN input features. AP0 evaluates the interference received from each neighbour and also their interference situation (e.g. AP1) to estimate the impact it can reversely have on them.

*c) Intermediate and Output Scores:* As illustrated in Fig. 3, the DNN comprises 3 further stages to produce the final output scores. First, the last dimension of $g_\tau$ is input to a Dense layer (of size 256): this produces information per AP and channel, reflecting the primary channel situation (denoted as P20 in Fig. 3) for an output having shape $N_{AP} \times N_C \times 256$. This tensor representing the primary channel is mirrored to reflect the situation on each of the corresponding secondary channels (denoted as S20 in Fig.3). For the sake of illustration, Fig. 3 maps blue (green) primary with green (blue) secondary channels, whereas the orange channels cannot be used for bonding in the example (represented by a grayed out layer). We note that this architecture is designed for bonding up to 2 channels. Similarly it can be augmented for 4 and 8 channels.

The concatenation of the primary and secondary channels (P20 and S20) is input to two dense layers (of size 256 and $N_B$ respectively), applied to the last dimension. Intuitively, the output in this layer gives intermediate scores: per AP, channel and bonding configuration pair. The intermediate score is augmented with a layer pooling the top-K intermediate scores per AP after flattening the channel and bandwidth dimensions. Intuitively, this DNN layer reveals the number of alternative configurations for each AP, which is helpful in selecting an order of allocation for the sequence.

Finally, two dense layers (size 4 and 1) reduce this information to a set of scores per AP and channel configurations. This is followed by a sofmax layer to produce the probability distribution.

### *B. The Critic DNN ($V$)*

The role of the Critic is to learn the state-value function $V$, so that it can assist the Actor's training. As for the Actor, we resort to DNNs to approximate it. A simplified view of the Critic DNN architecture is shown Fig. 3. Using as input feature the current state $(m, \ell, c, b)$ the critic estimates the function by explicitly modeling the interference:

$$f^V_{i,j,k,h} = m_{i,j}\ell_j[h]\ 2^{-b_j}\psi_k(c_j, b_j) \tag{12}$$

After, $f_V$ is aggregated (as in the Actor network) to $g_V$, then reduced by three dense layers (sizes: 256, 256 and 1) then averaged over all dimensions to obtain a single scalar $V$.

### *C. The Action Selector DNN ($Q$)*

Unlike a deterministic policy (e.g. obtained via Q-learning), the stochastic policy we learn can help mitigate the risk for local minima, because it allows to sample it multiple times (possibly in parallel) for a single inference, producing a (limited size) subset of candidate actions $\mathcal{A}_t$, among which the Selector DNN chooses the best one:

$$A_{best} = \text{argmax}_{a \in \mathcal{A}_t} Q(S_t, a) \tag{13}$$

The Selector employs a DNN architecture similar to the one of the Actor and Critic DNNs, whose aim is to learn $Q_\theta$ by leveraging the following input features:

$$f^{Q\ (recv)}_{i,j,k,h} = m_{i,j}\ell_j[h]\ 2^{-b^a_j}\psi_k(c^a_j, b^a_j) \tag{14}$$

$$f^{Q\ (neigh)}_{i,j,k,h} = \sum_{x \in \mathcal{N}_{AP} \setminus \{i,j\}} f^{Q\ (recv)}_{j,x,k,h}\psi_k(c^a_j, b^a_j) \tag{15}$$

Then $f^{Q\ (recv)}$ and $f^{Q\ (neigh)}$ are aggregated and complemented with further information (e.g., per-AP current and historical load $\ell[h]$ ($h \in \mathcal{N}_H$), and an indication of per-AP configuration change $\mathbb{I}\{c_i \neq c_i^a\} \cdot \mathbb{I}\{b_i \neq b_i^a\}$), to produce the intermediate output $g_Q$ of size $N_{AP}N_C(2N_g + N_g^{''})$ that is fed to a stack of Dense layers and finally averaged over all dimensions to obtain a scalar $Q$ output.

### D. The Reinforcement Learning loop

For estimating the best policy $\pi_\theta$ and the corresponding state value function $V_\theta$ we rely on the following classic double recursion:

$$\begin{aligned} \delta_{t+1}^v &= R_{t+1} + \gamma V_\theta\left(S_{t+1}\right) - V_\theta\left(S_t\right) \\ \theta_{t+1}^v &= \theta_t^v + \alpha_t^v \delta_{t+1}^v \nabla_{\theta^v} V_\theta\left(S_t\right) \\ \theta_{t+1} &= \theta_t + \alpha_t \delta_{t+1}^v \nabla_\theta \log \pi_\theta\left(X_t, A_t | S_t\right) \end{aligned} \tag{16}$$

where $\delta_{t+1}^v$ represents the state value function Temporal Difference (TD)-error, $\alpha_t^v$ and $\alpha_t^\theta$ are learning rates, and $\gamma$ is the discount factor. In addition to that, the Selector DNN is trained in parallel (but used only at inference), with the following recursions:

$$\begin{aligned} \delta_{t+1}^q &= R_{t+1} + \gamma V_\theta(S_t + 1) - Q_\theta\left(S_t, A_t\right) \\ \theta_{t+1}^q &= \theta_t^q + \alpha_t^q \delta_{t+1}^q \nabla_{\theta^q} Q_\theta\left(S_t, A_t\right) \end{aligned} \tag{17}$$

where $\delta_{t+1}^q$ is the action-state value function TD-error and $\alpha_t^q$ is the learning rate.

## V. Local Search (*dynLS*) and other Heuristics

As commonly done in the literature [9], [10], [13]–[16], we consider local search heuristics: namely, the current state of the art (*TurboCA*) [16], a practical Dynamic Local Search (*dynLS*) proposal that improves upon the state of the art, and an idealized version that is useful as a baseline (*Oracle*).

### A. TurboCA

*TurboCA* [16], used in Meraki products, is the current state of the art. *TurboCA* performs node-by-node updates to the channel allocations, so as to adapt to load changes at different time scales: on a relatively short timescale (15 min) the AC tries to find a better configuration for individual APs, and on a longer timescale (3hr and daily) larger parts of the network are reconfigured by going over neighborhoods of nodes iteratively, adapting to more significant load changes while avoiding bad local optima. For lack of space, we refer the reader to [16] for further details. Our *TurboCA* implementation performs a 2-neighborhood clearance in the first iteration, and a 1-neighborhood clearance every 12 iterations afterwards. To perform a fair comparison, we modify the original objective function with the same regret of *dynLS* and *DRL* so that all solutions are compared on the same ground.

### B. Dynamic Local Search (dynLS)

At high level, *dynLS* iteratively improves the current configuration in a randomized edge-by-edge fashion, performing a number of runs and outputting the result of the best run. In detail, *dynLS* takes as input the network topology $m$, the current load $\ell_0$ and the current channel and bonding configuration $(c, b)$. It then iterates in a randomized fashion over the set $\mathcal{E}$ containing all AP pairs with at least one interference relation. Each AP pair is optimized while keeping all other configurations constant. One run of *dynLS* ends when no further improvement is made, or some configurable step limit is reached. In our simulations, we allow for 4 random runs. For details see Alg. 1.

**Algorithm 1** dynLS

1: $(c^a, b^a) = (c, b)\,, i = 0, done = \text{false}$ ▷ initialize
2: **while** not $done$ and $k <$ thresh **do** ▷ local opt or interrupt
3: Sort $\mathcal{E}$ randomly, $k = k + 1$
4: $done = \text{true}$ ▷ only remains true if all $\mathcal{E}$ checked
5: **for each** $\{i, j\} \in \mathcal{E}$ **do**
6: Find $c_i, b_i, c_j, b_j$ with min regret (complete enum)
7: **if** $c_i, b_i, c_j, b_j$ improve over $(c^a, b^a)$ w.r.t $R^s + R^t$ **then**
8: $c_i^a = c_i, b_i^a = b_i, c_j^a = c_j, b_j^a = b_j$ ▷ update
9: $done = \text{false}$
10: **break** ▷ reshuffle

Note that, within one run, the current solution keeps improving with respect to regret (4), and thus terminates in a local minimum unless interrupted early. The randomization in step 3 expands the search space and helps avoiding *bad* local minima. Two APs $i$ and $j$ are optimized in step 6 by complete enumeration of all their possible configurations. Since each configuration needs to be evaluated w.r.t the regret function, there are $O(|\mathcal{C}|^2\,|\mathcal{B}|^2)$ such evaluations per edge and $O(|\mathcal{E}|\,|\mathcal{C}|^2\,|\mathcal{B}|^2)$ per for-loop (steps 5 to 10).

*a) Advantages of* dynLS*:* While *dynLS* is inspired by prior LS approaches, it differs in that it improves edge-by-edge instead of node-by-node. We explain its advantage with two relevant examples from the literature. In [9], the authors noticed the propensity to being stuck in bad local optima, and thus introduced a swapping step to avoid it: when a local optimum is reached, they attempt to improve by exchanging channels on neighboring nodes. Such step is unnecessary when operating in edge-by-edge fashion as in *dynLS*, since channel configurations are already established for *neighbor pairs* (as opposed to individual nodes). Moreover, particularly in the case in which channel bonding is allowed, a simple configuration swap is not sufficient to avoid bad local optima: in *dynLS*, directly optimizing neighboring nodes jointly also leads to a steeper descent towards a good local optimum. In *TurboCA* [16], we remark that a similar problem related to node-wise operation appears: in each larger optimization step, nodes are reconfigured in a greedy fashion. The problem with this approach is that the last nodes of each larger optimization step can produce significant interference that will not be eliminated until the next step – which *dynLS* avoids by design.

*b) Limits of* dynLS*:* The evaluation in step 8 needs to compute with respect to the initial configuration $(c, b)$, the reconfiguration and state regret for each new configuration. Since the load in the following time period (19) is not available to *dynLS*, the state-regret is calculated using the current load $\ell_0$ as predictor of future load – a clear oversimplification.

### C. Oracle

The *Oracle* baseline extends *dynLS* in two ways:

- First, we directly provide the *Oracle* with knowledge of future load, hence avoiding the need for learning, which we argue to be an unfair advantage over both the naïve forecast of *dynLS*, as well as over the learning of the DRL-based method;
- Second, we relax the time constraint to extend the *Oracle* search space, using a threshold of 100 runs (15 starting from the state configuration, and 85 from additional random states).

Albeit impractical due to the use of future knowledge, the *Oracle* baseline is interesting for us as it allows to get closer to the optimal by (i) allowing *dynLS* to significantly extend the search space, and as well as (ii) relieving it from the need of learning the future demand.

## VI. Performance Evaluation

We evaluate performance via simulation and experiments. In particular, while the regret and evaluation metrics (Sec.VI-A) are common to both evaluation methods, we are only able to fully control (Sec.VI-B) the scenarios of the simulation simulation-based methodology (Sec.VI-C), but do not have full control on the real deployment (Sec.VI-D).

### A. Regret and evaluation metric

As introduced earlier, *net2seq*, *TurboCA* and *dynLS* algorithms are not bound to a specific regret function. Without loss of generality, we fix the choice for the regret (4) used consistently to train, guide and evaluate the different algorithms. First, we define the state regret:

$$R^s_{t+1} = \sum_i \hat{\rho}_{2^{B_{t+1,i}}}\left(U^+_{t+1,i}\right) L_{t+1,i} \quad (18)$$

where $U^+_{t+1}$ corresponds to the maximum channel utilization at time $t+1$ as defined in (2) and $\hat{\rho}_\beta(x)$ is a concave function computing the individual regret of each AP shown in Fig. 5. For $x < 0.9$, $\hat{\rho}_\beta(x) = \rho_\beta(x)$: the throughput of an M/M/1 queue scaled by the quota of used channels out of the maximum 8 allowed by the standard ($\beta/8$), wrapped into a logarithm to impose fairness among APs. To avoid the vertical asymptote of $\rho$ at $x \to 1$, we substitute it with an exponential function for $x \geq 0.9$. Finally, we sum over the individual AP regrets, weighting by the AP load to give more importance to busy APs. Next, the reconfiguration regret simply identifies the configuration change for each AP and sums them weighting by load of the AP:

$$R^r_{t+1} = \sum_i \left(1 - \mathbb{I}_{\{C_{t+1,i}=C_{t,i}\}}\mathbb{I}_{\{B_{t+1,i}=B_{t,i}\}}\right) L_{t,i} \quad (19)$$

We also experimented with the relative importance $w$ of reconfiguration vs state regret, i.e. $R = R^s + w \cdot R^r$: as results are qualitatively similar, we fix $w = 1$ in what follows.

### B. Evaluation Scenarios

*1) Topologies:* We compare the algorithms on a reference *real* network topology composed of 49 APs (unless otherwise specified, but up to ∼150 APs for the most challenging scenarios), where each AP has on average 15 neighbors

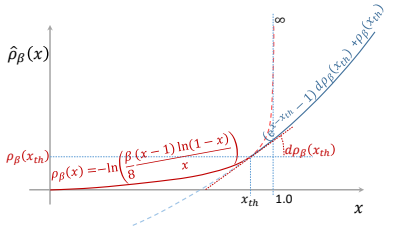


Fig. 5: Illustration of $\hat{\rho}$ used in the state regret definition

(unless otherwise specified, but up to 47 neighbors for the most challenging scenarios). As in the real deployment, every 10 minutes (15 min in [1]), we reconfigure APs using 9 channels in the 5 GHz band (excluding problematic DFS channels [1]). For bonding, we only consider a maximum of two channels. To study the transfer learning of DRL, we generate synthetic typologies varying the number of APs and the number of neighbours: APs are randomly placed based on a Space Poisson Point Process in the $[0,1]^2$ square. We use a log-distance pathloss model (with an exponent of 3.0), 4dB asymmetric log-normal shadowing and 3dB log-normal transmit power variation over APs (a constant term is added to scale the inter-AP distance and control the neighborhood).

*2) Traffic:* We express the traffic demand volume as a time-resource requirement. Following prior work [11], we use two types of traffic profiles exhibiting time-correlation.

*a) Volatile Demands:* In this profile, the load of all network APs oscillates between 0 and 100%, which represent a challenging scenario for any dynamic reconfiguration algorithm since all APs have highly volatile traffic profile. In particular, each AP starts with a random load $x \in [0,1]$, at each step the load is incremented by a random uniform amount in $[0, 0.2]$ until the load reaches 100%, after which the load decreases with the same random process, and so on.

*b) Flashcrowd:* This profile simulates transient traffic surge affecting regions of the network. All APs have a base load of 0.1 and an additional stochastic load component fluctuating in $[0, 0.2]$. A random subset of 3 APs is selected together with their 4 closest neighbors to form hotspot (HS) regions. HS APs get an additional load of 0.7, for a random duration of [3,9] time slots. Then a new flascrowd occurs: the HSs are re-selected and the entire process repeats.

*3) Training and testing:* We train each version of *DRL* on 14400 iterations, using a batch size of 32. To give an idea, training one *DRL* version in this setting takes around 1 hour on a Tesla V100 PCIe 16 GB GPU. Evaluation is done in two steps: in a first step, the best out of 10 models is selected as the one minimizing the average regret over a *validation-set* comprising 16 instances; results are then reported by running the selected model on a *test-set* comprising 16 instances. Each instance consists of 144 time slots, corresponding to a single day (144 reconfiguration cycles every 10 minutes). To be fair, we give any algorithm the same time budget of 1sec (2sec in case of bonding) to compute a full reconfiguration. Algorithms are evaluated by comparing the two components

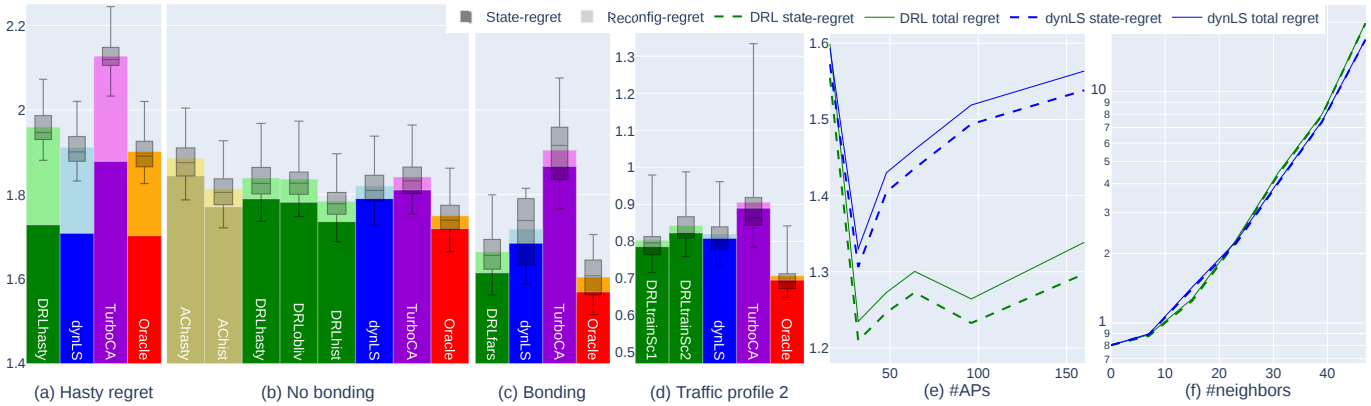


Fig. 6: Evaluation: (a)-(c) Comparison of regret under heuristic (*TurboCA*, *dynLS*, *Oracle*) vs DRL-based (*net2seq*) algorithms and (d)-(f) Transfer learning capabilities of *net2seq*

of the regret as given in (18)-(19), reporting both the spatio-temporal average (i.e. over all APs and iterations after a warmup of 25 iterations), as well as box plots with quartiles, minimum and maximum across the 16 test scenarios.

### C. Simulation results

We now compare the performance of heuristic vs DRL designs in Fig. 6 (a)-(c) and systematically assess DRL transfer learning capability in Fig. 6 (d)-(f).

*a) Hasty regret:* We first assess the algorithms based on their sole ability to find the best configuration for a given load, regardless of whether they can anticipate future load evolution or not. We do so by considering a *hasty* regret: altering $R^s_{t+1}$ in (18) by replacing $U^+_{t+1,i}$ and $L_{t+1,i}$ with $U^+_{t,i}$ and $L_{t,i}$, we optimize the configuration only for the current (known) load distribution. Accordingly, we train *DRL* on hasty regret, with no load history ($N_H = 0$) and no farsightedness ($\gamma = 0$). To further eliminate anticipation abilities, a random channel configuration is reset after each step. This penalizes TurboCA which cannot benefit from its varying optimization steps, which we partly compensate by applying 2-neighborhood clearance in each iteration. We remark in Fig. 6-(a) that *dynLS* is within 1% from the *Oracle* solution (diminishing return of a broader search space), closely followed by *DRL*.

*b) No bonding:* From now on, we focus on the initially defined regret (18) and let the algorithms run continuously, considering first only channel allocation, with no bonding. Fig. 6-(b) shows the performance of several *DRL* versions: the previously trained *hasty* version, an *oblivious* version trained with the normal regret but only instantaneous load ($N_H = 0$), and a version including *historical* load ($N_H = 2$). Note that $\gamma = 0$ for all *DRL* variations here.

First, we observe that *dynLS* and *DRL* performance are both (slightly) better than *TurboCA*, and also closer. Second, unlike *TurboCA* and *dynLS*, *DRL* is able to predict future demand from historical information, with a significant advantage over other *DRL* versions and approaching *Oracle* results. Third, we run *net2seq* in actor-critic mode only, disabling the selector network, and denote results as *AC*: results show that the use of the selector network improves the result for both *hasty* evaluation on current load as well for the case of *historical* load – confirming the soundness of our DNN design.

*c) Bonding:* When bonding is allowed, difference between *TurboCA* and the other algorithms widens, as shown in Fig. 6-(c). In particular, we see that the gap between *dynLS*, *DRL* (from here on *farsighted* $\gamma = 0.5$, with $N_H = 2$) and the *Oracle* remains consistent, with *DRL* still close to *Oracle*.

*d) Variable traffic profile:* While in practice, *DRL* should be trained on a wide variety of scenarios, Fig. 6-(d) assesses the transfer learning ability across traffic profiles: we test on a flashcrowd scenario two versions of *DRL*, trained on volatile vs flashcrowd respectively. We observe that the model trained on the more volatile profile achieves better results, which may be due to the fact in this case the load evolution is smoother allowing for a more stable learning.

*e) Variable network size:* Fig. 6-(e) employs a *DRL* model trained on the real 49-nodes topology and test it on synthetic networks with variable size and fixed density (15 neighbors/AP). Interestingly, we find that unlike *dynLS* whose performance decreases with increasing network size, the *DRL* performance is relatively steady. This result hints to a fundamental difference between heuristics, whose randomized search solution is impacted by the instance size, vs the *DRL* approach which seems to have learned a successful strategy, that is less impacted by the problem size. Further investigation is needed to explain the root cause of this notable difference.

*f) Variable number of neighbors:* Finally, Fig. 6-(f) shows that regrets of both *dynLS* and *DRL* grow for denser networks (where the number of APs is fixed to 49), that are both dominated by increasing interference, so that there is no longer any noticeable advantage on *DRL* over *dynLS*.

### D. Deployment results

As a proof of concept, we obtained access to a real corporate network composed of 34 APs, that supports about a thousands users and visitors every day. During a month, we deploy *dynLS*, *DRL* (*net2seq*) and *TurboCA* for five working days each, and contrast the results to those obtained with a static

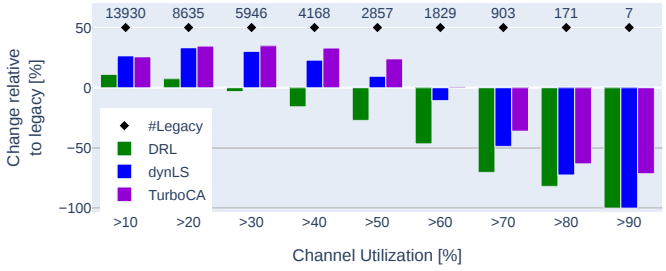


Fig. 7: Measurement results from real network deployment

daily re-optimization (denoted as *legacy*). We train *net2seq* on the simulated traffic profiles with $\gamma = 0.5$ and $N_H = 2$. Clearly, unlike in simulations where we can rigorously compare algorithms using the exact same input, in live deployment the comparison is more complex: while conditions (e.g., the network usage, traffic patterns, interference, etc.) are *similar* over different days, they are however not *identical*. As such, conclusive results can be gathered only from a longer experimental campaign than the one we present here: results in this section should be seen more as an empirical validation of our regret function and assumptions, rather than a fully fledged experimental comparison.

With the above caveat in mind, we start by observing from Fig.7 that the three algorithms improve over the legacy baseline, confirming the soundness of our regret design, which leads to a real enhancement in actual network conditions. The figure shows the evolution of channel utilization, as reported by existing telemetry. The top outlines absolute counts of observations with a value $> x\%$ as indicated on the $x$-axis, while the bars show the relative changes when deploying *DRL* *dynLS* and *TurboCA* , respectively. For example, all dynamic algorithms reduce by at least 50% the events where APs experience a utilization $> 80\%$. Overall, *dynLS* outperforms *TurboCA* and *net2seq* outperforms both. We plan to conduct a more thorough evaluation protocol as part of our future work.

## VII. Conclusions

This paper provides a novel Deep Reinforcement Learning (*net2seq*) design to tackle the problem of real-time channel and bandwidth allocation in WLANs. We have shown that *net2seq* can compete, performance and time-wise, with state-of-the art heuristics on this combinatorial optimization problem. Moreover, when given access to historical data, our *net2seq* design natively develops predictive abilities, allowing it to outperform our best heuristic by optimizing for future load. Finally, whereas the design of the individual DNNs presented in this paper is specific to WLAN channel allocation, our *net2seq* design can be easily extended to other combinatorial problems (e.g., plain graph coloring, travelling salesman), that are part of our future research agenda.

## References


[1] https://documentation.meraki.com/MR/Monitoring_and_Reporting/Location_Analytics/Meraki_Auto_RF%3A__Wi-Fi_Channel_and_Power_Management.

[2] https://www.arubanetworks.com/assets/tg/TB_AirMatch.pdf.

[3] https://www.mist.com/leveraging-reinforcement-learning-optimize-wi-fi/.

[4] M. Alicherry *et al.*, "Joint channel assignment and routing for throughput optimization in multi-radio wireless mesh networks," in *MobiCom*, 2005.

[5] A. Raniwala *et al.*, "Centralized channel assignment and routing algorithms for multi-channel wireless mesh networks," *ACM SIGMOBILE*, 2004.

[6] K. N. Ramachandran *et al.*, "Interference-aware channel assignment in multi-radio wireless mesh networks." in *Infocom*, 2006.

[7] M. K. Marina *et al.*, "A topology control approach for utilizing multiple channels in multi-radio wireless mesh networks," *Computer networks*, 2010.

[8] B.-J. Ko *et al.*, "Distributed channel assignment in multi-radio 802.11 mesh networks," in *IEEE WCNC*, 2007.

[9] T. Park and C. Y. Lee, "Application of the graph coloring algorithm to the frequency assignment problem," *Journal of the Operations Research society of Japan*, 1996.

[10] A. Mishra *et al.*, "A client-driven approach for channel management in wireless lans," in *IEEE INFOCOM*, 2006.

[11] E. Rozner *et al.*, "Traffic-aware channel assignment in enterprise wireless lans," in *IEEE ICNP*, 2007.

[12] Y. Liu *et al.*, "Measurement-based channel management in wlans," in *IEEE WCNC*, 2010.

[13] M. Bernaschi *et al.*, "A capwap-based solution for frequency planning in large scale networks of wifi hot-spots," *Comput. Commun.*, 2011.

[14] H. Zhang *et al.*, "Channel assignment with fairness for multi-ap wlan based on distributed coordination function," in *IEEE WCNC*, 2011.

[15] B. H. S. Abeysekera *et al.*, "Network-controlled channel allocation scheme for ieee 802.11 wireless lans: Experimental and simulation study," in *VTC Spring*. IEEE, 2014.

[16] A. Bhartia *et al.*, "Measurement-based, practical techniques to improve 802.11ac performance," in *IMC*, 2017.

[17] M. Seyedebrahimi *et al.*, "Sdn-based channel assignment algorithm for interference management in dense wi-fi networks," in *EuCNC*, 2016.

[18] S. Chieochan *et al.*, "Channel assignment schemes for infrastructure-based 802.11 wlans: A survey," *IEEE Communications Surveys Tutorials*, 2010.

[19] O. Vinyals *et al.*, "Pointer networks," in *NeurIPS*, 2015.

[20] I. Bello *et al.*, "Neural combinatorial optimization with reinforcement learning," *arXiv:1611.09940*, 2016.

[21] H. Dai *et al.*, "Learning combinatorial optimization algorithms over graphs," in *NeurIPS*, 2017.

[22] M. Nazari *et al.*, "Reinforcement learning for solving the vehicle routing problem," in *NeurIPS*, 2018.

[23] W. Kool *et al.*, "Attention, learn to solve routing problems!" *arXiv:1803.08475*, 2018.

[24] Q. Ma *et al.*, "Combinatorial optimization by graph pointer networks and hierarchical reinforcement learning," *arXiv:1911.04936*, 2019.

[25] A. Nowak *et al.*, "Revised note on learning algorithms for quadratic assignment with graph neural networks," *arXiv:1706.07450*, 2017.

[26] B. Ghahfarokhi, "Distributed qoe-aware channel assignment algorithms for ieee 802.11 wlans," *Wireless Networks*, 2014.

[27] H. Lemos *et al.*, "Graph colouring meets deep learning: Effective graph neural network models for combinatorial problems," in *IEEE ICTAI*, 2019.

[28] K. Nakashima *et al.*, "Deep reinforcement learning-based channel allocation for wireless lans with graph convolutional networks," *IEEE Access*, 2020.

[29] O. Vinyals *et al.*, "Order matters: Sequence to sequence for sets," in *ICLR*, 2016. [Online]. Available: http://arxiv.org/abs/1511.06391

[30] A. Vaswani *et al.*, "Attention is all you need," in *NeurIPS*, 2017.

[31] M. Gori *et al.*, "A new model for learning in graph domains," in *IEEE International Joint Conference on Neural Networks*, 2005.

[32] F. Scarselli *et al.*, "The graph neural network model," *IEEE Transactions on Neural Networks*, 2009.

[33] N. Naderializadeh *et al.*, "Resource management in wireless networks via multi-agent deep reinforcement learning," *arXiv:2002.06215*, 2020.

[34] H. Dai *et al.*, "Discriminative embeddings of latent variable models for structured data," in *ICML*, 2016.

[35] M. Henaff *et al.*, "Deep convolutional networks on graph-structured data," *arXiv:1506.05163*, 2015.

[36] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. The MIT Press, 2018.