Multi-Agent Reinforcement Learning for Hyperparameter Optimization of Convolutional Neural Networks

Arman Iranfar, Marina Zapater, *Member, IEEE,* and David Atienza, *Fellow, IEEE*

***Abstract*—Nowadays, Deep Convolutional Neural Networks (DCNNs) play a significant role in many application domains, such as, computer vision, medical imaging, and image processing. Nonetheless, designing a DCNN, able to defeat the state of the art, is a manual, challenging, and time-consuming task, due to the extremely large design space, as a consequence of a large number of layers and their corresponding hyperparameters. In this work, we address the challenge of performing hyperparameter optimization of DCNNs through a novel Multi-Agent Reinforcement Learning (MARL)-based approach, eliminating the human effort. In particular, we adapt Q-learning and define learning agents per layer to split the design space into independent smaller design sub-spaces such that each agent fine-tunes the hyperparameters of the assigned layer concerning a global reward. Moreover, we provide a novel formation of Q-tables along with a new update rule that facilitates agents' communication. Our MARL-based approach is data-driven and able to consider an arbitrary set of design objectives and constraints. We apply our MARL-based solution to different well-known DCNNs, including GoogLeNet, VGG, and U-Net, and various datasets for image classification and semantic segmentation. Our results have shown that, compared to the original CNNs, the MARL-based approach can reduce the model size, training time, and inference time by up to, respectively, 83x, 52%, and 54% without any degradation in accuracy. Moreover, our approach is very competitive to state-of-the-art neural architecture search methods in terms of the designed CNN accuracy and its number of parameters while significantly reducing the optimization cost.**

***Index Terms*—Convolutional Neural Network, hyperparameter optimization, neural architecture search, reinforcement Learning**

I. Introduction

Since the success of AlexNet in 2012 in defeating all existing state-of-the-art machine learning and computer vision approaches with respect to accuracy [1], Deep Learning has become one of the main research topics for industry and academia. Moreover, already today it is considered a very successful alternative to traditional artificial intelligence in various domains. In particular, Convolutional Neural Networks (CNNs), first introduced and then used in the late 1980s and 1990s [2], [3], are now an important member of the Deep Learning family. Indeed, they have been successfully applied to several application domains, such as, image processing, medical imaging, and computer vision.

Despite its success, popularity and vast investment by both academia and industry, designing a Deep CNN (DCNN) able to compete with the state-of-the-art is even more challenging today. Started from AlexNet with 6 layers and 60M network parameters to train, recent CNNs may contain over 100M parameters [4] and more than 1000 layers [5]. With this continuously increasing number of parameters and layers, training time, inference time, and model size may increase dramatically resulting in prolonged and costly design, delayed response in real-time systems, and memory challenges in constrained resources.

On one hand, DCNNs architectures can be different with respect to the number of layers, types of layers, and how these layers connect to each other. On the other hand, each layer contains several parameters to be set at design time before training. These parameters, different from trainable network parameters (such as weights), are often regarded as hyperparameters. For instance, in a convolution layer, as the main building block of CNNs, the number of kernels, kernel width, kernel height, and stride length are considered as the hyperparameters. Their values are shown to considerably affect the CNN accuracy, training time, inference time, energy consumption, etc. As a result, DCNN designers have to spend a considerable amount of time to optimally set these parameters. This task, known as Neural Architecture Search (NAS) and hyperparameter optimization, has become the main focus of many academic researchers [6]–[8] and leading companies, such as, Google or Facebook [9]–[11].

While random search [6] is considered as the baseline approach, researchers have recently focused on design automation of CNNs through Reinforcement Learning (RL), Genetic Algorithms, and Bayesian Optimization to either build a CNN from scratch or to further tune hyperparameters of an already-existing CNN [7]–[10], [12]. Nonetheless, these works are limited by the maximum number of layers to design [9], [12], the design objectives and constraints considered [7], [10], [12], and the type of layers handled [12]. One common major drawback of the existing neural architecture search methods in literature is that they design a CNN from scratch, thus, requiring a considerable amount of time. For instance, NAS [9] needs 2000 GPU days to design a CNN for relatively simple datasets, such as CIFAR10. While there are numerous well-known CNN architectures in the literature for different tasks, a data-driven scheme that can optimize an already-existing CNN for a particular dataset and design objectives and constraints can result in very large time-saving.

Hence, in this work, we address the problem of automatic hyperparameter optimization of DCNNs through Multi-Agent Reinforcement Learning (MARL) [13] without any limits in number and types of layers, as well as design objectives and constraints. Figure 1 shows an overview of our proposed MARL-based hyperparameter optimization of an n-layer CNN with arbitrary skip connections. In particular, we adapt Q-Learning (QL), a model-free algorithm of RL, and define learning agents per layer to split the design space into smaller independent sub-spaces; thus, each agent can fine-tune the hyperparameters of the assigned layer with respect

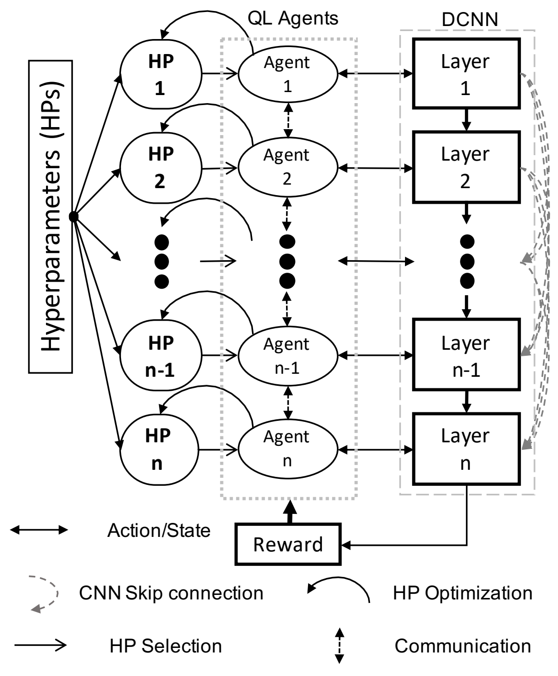


Fig. 1. Overview of proposed MARL-based hyperparameter optimization

to a global reward. In the proposed multi-agent QL-based approach, agents are able to communicate through our novel definition of state-action pairs, Q-tables, and a Q-table update rule.

The main contributions of this paper are as follows:

- We propose a novel multi-agent RL-based approach that eliminates the human effort in hyperparameter optimization of DCNNs.
- Our novel definition of QL elements enables us to split the design space into smaller sub-spaces, providing faster, yet accurate search.
- Our proposed approach is data-driven, i.e., given a CNN architecture, it tunes the hyperparameters according to the input data to the CNN.
- Our proposed approach can consider multiple arbitrary constraints and objectives imposed by the application or the processing platform.
- We show that our solution is capable of fine-tuning any arbitrary DCNN, by applying it to different architectures and well-known datasets. Our experiments reveal that the proposed MARL-based approach can decrease the model size, training time, and inference time by up to 83x, 52%, and 54%, respectively, with no degradation in accuracy in the worst case.

II. Motivation and Problem Definition

CNNs encompass a large range of different architectures and layer depths. While the true history of DCNNs started from AlexNet [14] with only 6 layers, nowadays there are many deeper CNNs available, such as VGG [4] with up to 19 layers, and ResNet [5] with up to 1022 layers [1]. On one hand, these CNNs are composed of several types of layer, such as *convolution*, *pooling*, *fully connected*, *softmax*, etc. During the training process, there may be millions of internal

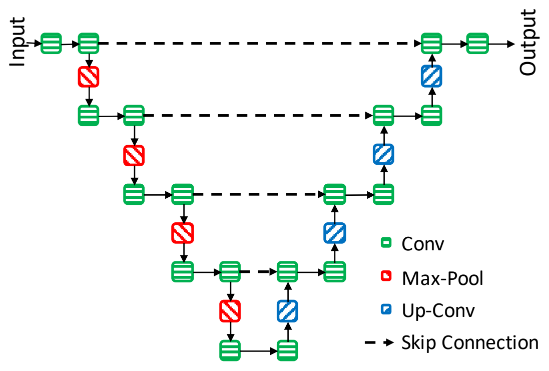


Fig. 2. A simplified U-Net architecture

parameters (e.g., weights) within the CNN that are optimized with respect to a particular loss or accuracy metric via an optimizer, such as Adam [15] and Stochastic Gradient Descent (SGD) [16]. On the other hand, each layer has a couple of so-called hyperparameters, that, unlike these internal parameters, are not or cannot be optimized during the conventional training procedure. Researchers and CNN designers traditionally trust rule-of-thumb approaches followed by grid search or random search. While even with a small number of layers grid search seems impractical, with deeper CNNs random search also becomes too time-consuming and inefficient.

Figure 2 shows a simplified U-Net architecture profoundly used in semantic segmentation of biomedical images [17]. For clarity, let us consider only the number of kernels and kernel size of convolution layers (including *Up-Conv* layers, a.k.a. *Deconvolution*) as the hyperparameters. Moreover, we limit the design space of the hyperparameter as follows. The number of kernels can be any value from $\{16, 32, 64, 128, 256, 512, 1024\}$ and kernels are square-like where width and height are equal and can be any value from $\{3, 5, 7\}$. Therefore, according to the fundamental counting principle, there are 7×3 different choices for each convolution layer. Finally, since there are 22 convolution layers in the U-Net shown in Figure 2, where each is designed independently, 21^{22} different choices are available when designing such a CNN. Throughout this work, we refer to any combination of different hyperparameters of different layers as a hyperparameter set.

In addition, Figure 3a shows the model size, training time per batch, and accuracy (with respect to Intersection over Union metric) for 1000 different hyperparameter sets used to train the U-Net described in Figure 2 for a limited number of epochs on an NVIDIA V100 GPU. As shown in Figure 3a, there are many sets of hyperparameters for which the accuracy, as the most important metric in designing DCNN, remains close to 0. On the other hand, although for several hyperparameter sets the U-Net converges to higher accuracy, the training time and model size, as the other two important metrics, change considerably depending on the hyperparameters. Obviously, smaller models with shorter training/inference time and higher accuracy are the most desirable of all. However, finding such a hyperparameter set that satisfies all constraints and objectives is extremely challenging with such a huge design space of 21^{22} different hyperparameter sets.

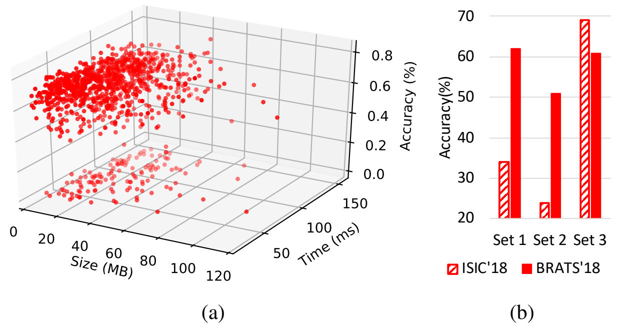


Fig. 3. a) Model size, validation accuracy, and inference time for 1000 different sets of hyperparameters for BRATS'18 dataset, b) Accuracy with three different hyperparameter sets for BRATS'18 and ISIC'18 datasets.

Finally, Figure 3b compares the accuracy of a U-Net trained for limited number of epochs with two different datasets, BRATS'18 [18]–[20] and ISIC'18 [21], [22]. As shown in the figure, with the same hyperparameter sets, the output accuracy is different due to different input data. This observation demonstrates that a successful hyperparameter optimization should be data-driven. In this context, a successful hyperparameter optimization approach would be the one that considers input data characteristics and tunes the hyperparameters accordingly, since we require a particular DCNN to perform well on the specific dataset for which it is trained.

Among all traditional approaches, RL is very efficient in dealing with very large design spaces and is known to provide such a data-driven solution, as it does not require any prior knowledge about the input data.

III. Related Work

Random Search [6], with almost no complexity overhead, is known as the baseline for hyperparameter optimization. Although under certain conditions and in specific problems it has been claimed that Random Search could be competitive to other neural architecture search and hyperparameter optimization approaches [6], in more complex scenarios with larger design space, as in the case of DCNNs, Bayesian Optimization, Genetic algorithms, and RL are shown to be superior solutions [8], [10].

Bayesian optimization is among the most popular methods for neural network architecture search and hyperparameter optimization [7], [8]. However, since Bayesian optimization is based on Gaussian processes, its application is limited to optimization problems with low dimensionality [23].

Evolutionary search and Genetic algorithms (GAs) are the most traditional approaches for neural network optimization [24] used recently in neural architecture search and hyperparameter optimization [25], [26]. These algorithms, however, are strongly dependent on heuristics. Unlike GAs, RL algorithms are based on Markov Decision Process (MDP), a mathematically grounded framework. Therefore, RL-based approaches for hyperparameter optimization have recently attracted a lot of attention.

These approaches can be divided into two different categories. In the first category, RL is used to optimize an existing network, either by tuning its hyperparameters or by adding and removing layers. In this context, [27] uses policy gradient to prune filters in CNN while having the CNN perform at a desirable accuracy, whereas EAS [28] uses RL to enable extending pre-existing plain convolutional neural networks to more sophisticated structures.

In the second category, RL is used to build a complete network from scratch. Therefore, in comparison to the first category, this one tackles a larger design space, thus, it is more time-consuming. In this context, Zoph et al [9] use Recurrent Neural Networks (RNNs) along with RL to build DNNs. MetaQNN [12] uses Q-Learning to build CNNs from scratch. DARTS [29] proposes a differentiable architecture search method that speeding up the design space search procedure compared to [9] and [12]. A few works, rather than building the whole CNN, design blocks similar to famous Residual and Inception modules and build the network by concatenating them. Examples of these works are BlockQNN [30], PNAS [31], and ENAS [32].

One drawback of the state-of-the-art in this area is that the types of layers are limited [12], [27]. Moreover, the maximum number of layers of the CNN should be known a priori in these works and the application of these approaches in designing deeper CNNs remains in question [9], [12]. None of these works [9], [12], [27], [30]–[32] address multi-objective/constraint design of DCNNs. In contrast, MONAS [33] is a multi-objective neural architecture search approach inspired by [9] which finds hyperparameters with respect to model accuracy and energy consumption. MONAS, however, considers a more limited subset of different action values (hyperparameters per layer) and requires a redesign of the RNN-based controller for designing different CNNs. On the contrary to MONAS, our proposed multi-agent QL-based approach is consistent in designing different CNNs without any need for change in its structure. Finally, few-shot learning [34] is a technique that can be used within other state-of-the-art NAS approaches to facilitate neural architecture search by considering multiple supernetworks and searching on smaller spaces. As reported by [34], few-shot learning settings can improve the classification accuracy of DARTS [29] on CIFAR10 dataset by more than 1% due to more precise design space search, at the cost of the increased GPU days required for the CNN design.

To recap, in contrast to these existing RL-based approaches, our proposed MARL-based method does not limit the number of layers in the CNN, can include skip connections in the CNN architecture, and scales well with the depth of CNN architectures. Table I compares the state-of-the-art in terms of types of layer optimized, objectives and constraints considered, support for unconventional modules, and types of CNNs (tasks) evaluated. In this table, we differentiate between support for among arbitrary layers and support for residual modules, as the latter is a subset of the former.

Moreover, in this work, we address automation of hyperparameter optimization of an existing architecture rather than performing Neural Architecture Search [9] (NAS) from scratch. Our idea lies in the fact that many existing architectures, in terms of connections between different layers, have been already shown to achieve satisfactory results for a wide-range of applications, such as, image classification, semantic segmentation, and object tracking, under particular open-sourced datasets. These models, however, have been mostly designed for particular competitions, such as, Large Scale Visual Recognition Challenge (ILSVRC). Thus, they may not suit other datasets, even for the same tasks. Also, they are not usually optimized with respect to other design objectives or constraints, such as, inference time, energy consumption, and model size. Moreover, designing the CNNs from scratch is excessively time-consuming and state-of-the-art NAS approaches have not shown any significant improvement with respect to the classification accuracy compared to the existing well-known CNN architectures. Therefore, we focus on hyperparameter optimization of the existing CNN architectures for arbitrary datasets and design objectives and constraints.

IV. Reinforcement Learning: Preliminaries and Background Concepts

RL is a machine learning approach that deals with environment-dependent problems through dynamic optimization programming [37]. RL traditionally refers to single-agent reinforcement learning (SARL) where there is one, and only one agent dealing with the environment. In contrast, Multi-Agent Reinforcement Learning (MARL) employs more than one agent in interaction with the environment to cope with more complicated problems.

A. Single-Agent Reinforcement Learning

Conventional RL, i.e., SARL, employs an agent capable of taking actions from a finite action set, $\boldsymbol{A}$, observing states from a state space, $\boldsymbol{S}$, and using a reward as a result of selecting an action in a particular state to further modify its future behavior. In this work, we leverage Q-Learning (QL), a model-free algorithm of RL. In QL, the agent learns the best policy, π, to apply an action in each specific state by storing a Q-value for each state-action pair as $Q^{\pi}(s, a)$. Q-values are updated after a new state and reward are observed, as follows:

$$\begin{aligned} Q_{t+1}(s_t, a_t) = (1-\alpha) \times Q_t(s_t, a_t) + \\ \alpha \times (R_{t+1} + \gamma \max_{a} Q_t(s_{t+1}, a)) \end{aligned} \tag{1}$$

where $Q_t(s_t, a_t)$ and $Q_{t+1}(s_t, a_t)$ are, respectively, the current and updated Q-values corresponding to the current taken action a_t and at the current state s_t, R_{t+1} is the immediate reward after next state s_{t+1} is observed, α determines the learning rate, and γ is the discount factor.

B. Multi-Agent Reinforcement Learning

MARL is composed of multiple agents competitively or cooperatively coping with a particular problem. While in competitive MARL agents compete with each other to maximize their own reward obtained from the environment, in cooperative MARL, agents help each other to more efficiently solve a problem and, thus, obtain a higher shared reward. In particular, the latter is beneficial if the task given to an agent is very large or complex to handle. In this context, agents can be homogeneous, i.e., they have the same features and responsibilities. In this variation of MARL, agents explore the whole environment autonomously, but share their experience, e.g., the learned policies, to each other to come up with a more likely optimal policy. In contrast to homogeneous agents, cooperative MARL agents can have different features and, hence, different tasks can be assigned to them. These so-called heterogeneous agents can split the large and complicated tasks to provide faster, yet accurate exploration in the environment.

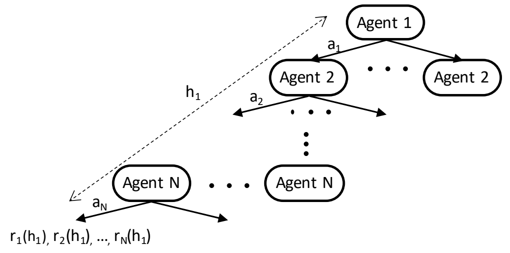


Fig. 4. Extensive-form Markovian game [38]

From another perspective, MARL frameworks can be divided into two seemingly different yet closely related theoretical frameworks, namely, normal-form and extensive-form Markov games [38]. While in the former, all agents take actions simultaneously, in the latter, agents make decisions on choosing actions successively and receive each reward at the end of the game round as depicted by Fig. 4. In this figure, h_i refers to a history passed after all agents have taken their action in the i^{th} round of the game. Although in extensive-form games agents have only partial observations, i.e., imperfect information of the environment, it is still possible to model imperfect information for multi-agent decision-making [39]. These two forms of MARL settings are still closely connected to each other, such that through certain assumptions the extensive-form can be reduced to the normal-form [39]–[41].

V. Proposed Reinforcement Learning-Based Approach

In this work, we propose to use multiple QL agents to design a DCNN, layer by layer, through cooperative teamwork. During the learning process, each agent is assigned to design a single layer. An agent's action, however, affects the following agents' behavior. Therefore, agents target to find the best action at a given state to lead the team to gain a higher reward. Since the current state observed by each agent is a direct consequence of the previous agent's action, each agent has to optimize its behavior such that the next agent is situated in a desirable state. Such a behavior, then, should propagate throughout the whole CNN and all layers, where, finally, the last agent's action results in a reward signal.

Figure 5 shows an abstract view of an arbitrary CNN composed of 5 layers to be designed. The design of each layer L_i, $i \in \{1, 2, 3, 4, 5\}$, is managed by an agent, AG_i, with a particular available action set, $\boldsymbol{A_i} = \{a_{i,j} | j = 1, 2, 3, ...\}$, after splitting the design space into smaller sub-spaces, as the shown in the figure. After observing its current state, each agent can apply an action from this action sub-set. A

TABLE I
STATE-OF-THE-ARTS ON HYPERPARAMETER OPTIMIZATION AND NEURAL ARCHITECTURE SEARCH OF CNNS

SoA	**Optimized Layers**			**Stride**	**Objectives and Constraints**				**Unconventional modules**			**CNN task**	
	Conv	Pool	Dense		Tr. time	Inf. time	Acc.	Size	Skip Conn.	Residual	Inception	Class.	Seg.
MetaQNN [12]	✓	✓	✓	✓	x	x	✓	x	x	x	x	✓	x
NAS [9]	✓	✓	✓	✓	x	x	✓	x	✓	✓	x	✓	x
EAS [28]	✓	✓	✓	✓	x	x	✓	x	x	✓	x	✓	x
BlockQNN [30]	✓	✓	x	x	x	x	✓	x	✓	✓	✓	✓	x
PNAS [31]	✓	✓	x	✓	x	x	✓	x	✓	✓	✓	✓	x
ENAS [32]	✓	✓	x	x	x	x	✓	x	✓	✓	✓	✓	x
MONAS [33]	✓	✓	x	x	x	✓	✓	x	✓	✓	x	✓	x
MANAS [35]	✓	✓	x	✓	x	x	✓	x	✓	✓	✓	✓	x
DARTS [29]	✓	✓	x	✓	x	x	✓	x	✓	✓	✓	✓	x
SNAS [36]	✓	✓	x	x	x	x	✓	x	✓	✓	✓	✓	x
Proposed	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

unique reward signal is also received by each agent, based on which further updates on the Q-tables are possible. In particular, every two successive agents share one Q-table, as shown in Figure 5. In conventional Q-learning, a Q-table is simply a table where rows and columns represent, respectively, states and actions of a single agent. In our defined multi-agent environment, however, each agent has its own action set, whereas its current state is determined by the actions taken by the previous agent in the sequence as will be explained in Section V-C. Thus, the state set, $\boldsymbol{S_i}$ observable by the i^{th} agent, is the same as the action set available to the $(i-1)^{th}$ agent, $\boldsymbol{A_{i-1}}$. To model this scenario in the proposed multi-agent QL-based solution, we propose to use Q-tables shared between each two consecutive agents, as shown in Figure 5.

Our proposed approach has a similar setup to the one shown by Figure 4, where the game starts from the first agent and the next agent acts based on the information coming from the previous agent. Although the extensive-form has been mostly used in competitive settings, we have adapted it for our cooperative setting by letting all agents share a common reward signal. Therefore, in our MARL problem $r_1 = r_2 = ... = r_N = R$. Moreover, the proposed MARL setting for our particular problem can be considered as several 2-player extensive-form games, but in a cooperative manner. In particular, each two successive agents refer to a Q-table that keeps the history ($h_{i,j}$) of the agents' joint action space which is a required component of extensive-form MARL [38]. In our specific definition of history, i represents the i^{th} iteration and j denotes the j^{th} pair of two consecutive agents. Therefore, Figure 6 illustrates the extensive-form game adapted in our work.

In our work, the action space of each agent is different (as shown in Figure 5) and, thus, they are heterogeneous. Nonetheless, theoretically, it is still possible to consider only one agent to sequentially design the layers similar to Figure 4. However, in this formulation, the agent requires to keep track of a very long history. This history eventually is extremely hard to follow as it grows super-linearly with the number of layers to be designed. Nevertheless, one of the main goals of our work is to provide a solution for designing very deep CNNs without increasing the exploration overhead. Therefore, we propose to use multiple agents each taking charge of designing one layer sequentially. In this case, each agent faces

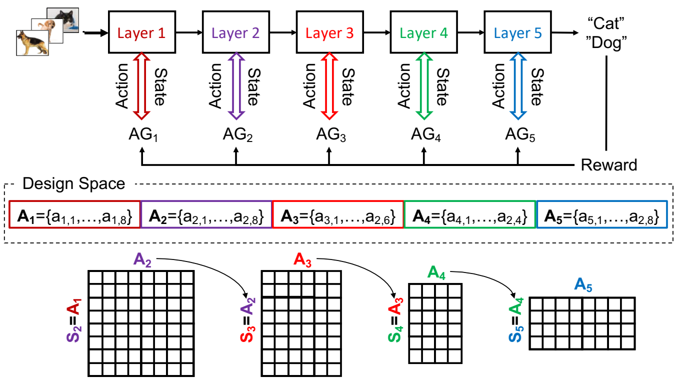


Fig. 5. Schema of proposed MARL-based approach on a 5-layer CNN

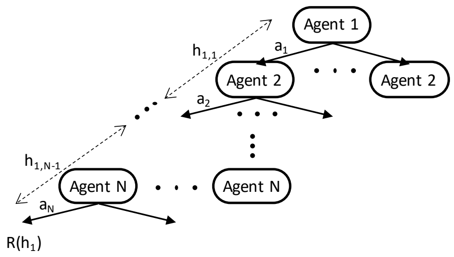


Fig. 6. Extensive-form Markovian game adapted in our work

a significantly smaller design space. Moreover, considering one agent per layer makes each agent keep track of only a part of the history, as shown in Figure 6. By formulating the MARL problem as described in our work, we are able to significantly reduce the number of iterations required for both exploration and exploration-exploitation phases. Although this formulation of MARL may lead to near/sub-optimal results due to imperfect information, it makes it possible to design extremely deep CNNs, such as with over 1000 layers.

In what follows, we detail the different elements of our multi-agent environment, namely, agents, states, actions, Q-tables, and reward function for hyperparameter optimization of DCNNs.

A. Agents

Given a CNN architecture, there are as many agents as the number of layers whose parameters need to be decided. In a particular layer, there could be more than one parameter value to be selected, however, we let a single agent be in charge of tuning these parameters. Moreover, agents do their part one after each other, i.e., sequentially, in the same order as of the CNN architecture, from the input to the output.

B. Actions

Each agent is responsible for hyperparameter tuning of a particular layer. Therefore, the action space of an agent is defined by the number of different parameters and their corresponding values in the layer. Each layer may contain more than one hyperparameter. In addition, each of these hyperparameters can take different values. Hence, an agent's action set is composed of tuples of all possible values of all different available hyperparameters. For instance, in a convolution layer, parameters, such as, number of kernels ($\{n_{k_1}, \cdots, n_{k_N}\}$), kernel size ($\{s_{k_1}, \cdots, s_{k_M}\}$) , and stride length ($\{l_{s_1}, \cdots, l_{s_L}\}$) can be considered as hyperparameters and their corresponding values. Thus, for the case of this convolution layer, the agent's action set is defined as:

$$\begin{aligned}\boldsymbol{A_{conv}} = \{&(n_{k_1}, s_{k_1}, l_{s_1}), (n_{k_2}, s_{k_1}, l_{s_1}), \cdots, (n_{k_N}, s_{k_1}, l_{s_1}),\\ &(n_{k_1}, s_{k_2}, l_{s_1}), (n_{k_1}, s_{k_3}, l_{s_1}), \cdots, (n_{k_1}, s_{k_M}, l_{s_1}),\\ &(n_{k_1}, s_{k_1}, l_{s_2}), (n_{k_1}, s_{k_1}, l_{s_3}), \cdots, (n_{k_1}, s_{k_1}, l_{s_L})\}.\end{aligned}$$

Consequently, when the convolution agent takes an action, it is choosing a tuple from $\boldsymbol{A_{conv}}$.

C. States

By splitting the design space into multiple independent action sub-spaces, states observed by each particular agent are different from others during the whole learning process. In particular, we define the state sub-space, to be experienced by an agent, as the actions taken by the previous agent in the sequence of agents explained in Section V-A. Therefore, whenever an agent takes an action, it modifies the state of the next agent in the sequence. The first agent in the sequence, however, is always considered to remain in an initial state.

D. Multi-agent Q-tables

Each Q-table, Qt_i, is an array of size $N_{A_i} \times N_{A_{i+1}}$, where N_{A_i} and $N_{A_{i+1}}$ are the number of actions available to the i^{th} and $(i+1)^{th}$ agent. Each two consecutive agents in the agent sequence share one Q-table, hence, there are $N_{AG} - 1$ Q-tables in total, where N_{AG} denotes the number of agents (layers). Finally, we initialize each Q-table with zeros.

E. Reward Function

The reward function in our framework can contain a large variety of signals readable, observable, or measurable at the end of each episode, such as, training loss/accuracy, validation loss/accuracy, training/validation time, GPU/CPU/memory utilization, model size, etc. In this work, we use validation accuracy and model size as the main parameters of the reward function. Moreover, we use training loss and training time as monitoring parameters in the early termination of the current episode (training with currently selected hyperparameters). In particular, we monitor training time for each batch (t_{batch}),

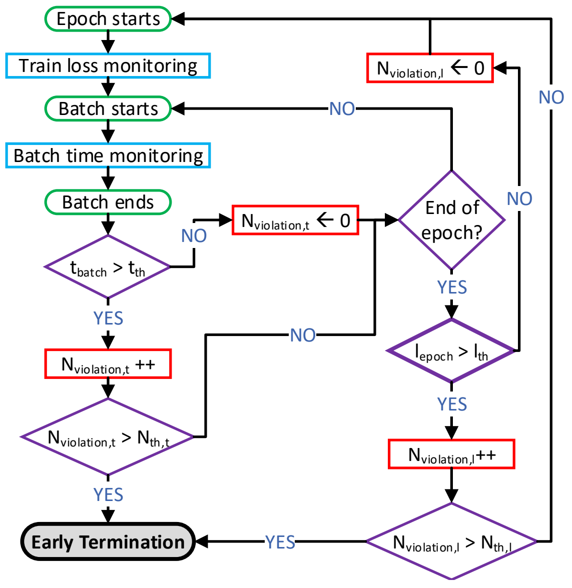


Fig. 7. Early termination mechanism

and activate early termination of the current episode if the current training time violates a predefined threshold time (t_{th}) for $N_{th,t}$ consecutive batches. Moreover, we monitor the training loss at the end of each epoch. If the training loss (l_{epoch}) violates a predefined loss threshold (l_{th}) for $N_{th,l}$ consecutive epochs, training with the current hyperparameter set is terminated. Figure 7 shows how these two mechanisms work to provide early termination, where $N_{violation,t}$ and $N_{violation,l}$ are two counters used for counting the number of consecutive violation of training batch time and epoch loss, respectively.

Finally, whenever early termination is activated through batch training time, an immediate penalty is given to the agents, whereas, if the early termination is due to the epoch loss, we propose to postpone this penalty. The reason lies in the fact that a CNN may perform poorly in the beginning due to the initialization of the weights. By restarting the training of the CNN with the same hyperparameters, we target to decrease the impact of this initialization. However, we tolerate such behavior only once, i.e., if the early termination is activated for the second time, agents are given a penalty (negative reward). The complete definition of the reward function is as follows:

$$\mathcal{R} = \begin{cases} -1 & Termination = True \\ c_A A_{val} - c_S S_{model} & otherwise \end{cases} \tag{2}$$

where A_{val} is the validation accuracy and S_{model} is the model size in MB, and c_A and c_S are coefficients such that $c_A + c_S = 1$. These two coefficients can be used to prioritize one objective over the other to the agents.

VI. Learning Process

The proposed learning process consists of two main phases, namely *exploration* and *exploration-exploitation*. In each phase, the strategy to take an action is different. Moreover, since our multi-agent QL-based solution is different from the conventional QL algorithm, we propose a new rule to update the Q-tables.

Algorithm 1: Random action in exploration phase

```
Input  : $\boldsymbol{Qt_i}, \boldsymbol{A_i} = \{a_{ij}\, j \in \{1, ..., N_{A_i}\}\}$
Output: $a_i^*$ ;                    // action of the $i^{th}$ agent
1 forall $i$ do
2     if $i = 1$ then
3         find $(a_i^*, a_{i+1}^*)$ $st.$ $\boldsymbol{Qt_i}[a_i^*, a_{i+1}^*] = 0$
4     else
5         find $a_{i+1}^*$ $st.$ $\boldsymbol{Qt_i}[a_i^*, a_{i+1}^*] = 0$
```

A. Exploration Phase

In the exploration phase, agents are only allowed to take random actions. This is similar to an $\epsilon - greedy$ [37] policy with ϵ always kept at 1. However, to help each agent explore more completely different states, we propose Algorithm 1. As shown in our algorithm, each agent takes diverse actions without repetition until its Q-table does not contain any cells initialized to zero. Note that the first two agents take their actions at a single step without any condition on the next agent (i.e., AG_3). On the contrary, the next agents take actions whose values are still equal to the initialized value (zero), but with respect to the action already selected by the previous agent (Line 5). Moreover, if there is no action found such that $\boldsymbol{Qt_i} = \boldsymbol{0}$ (Lines 3 and 5), AG_i does not follow Algorithm 1 and takes a completely random action.

Then, based on Algorithm 1, the number of episodes to eliminate all zeros (initial values) from all the Q-tables is determined by the largest Q-table. In other words, we require at least $\max(N_{A_i} \times N_{A_{i+1}}), i \in \{1, ..., N_{AG}\}$ episodes during the exploration phase.

Each CNN created by the agents is trained for a very limited number of epochs. Although the validation and/or train loss and accuracy at the end of these epochs would be far from the one to which the designed CNN can ultimately converge, it gives a useful insight into the behavior of the CNN with respect to training and validation time, and model size, as well as whether the CNN hyperparameters sound or could be discarded later on. This number of epochs for each CNN and input data, however, may be different. In order to automatically set this number, we first set the maximum number of epochs in each episode equal to 10 similar to [12]. At the end of the episode, we monitor the loss and the epoch number where the predefined loss threshold discussed in Section V-E is already satisfied. If this satisfaction occurs at an epoch smaller than 10, we set the number of epochs in each episode to this smaller value.

For the first episodes, the agents may benefit from more epochs to reach the loss threshold. In order to provide a fair comparison among different episodes, we use an accuracy-like score value instead of the absolute accuracy in the reward function of Eq. 2, where A_{val} becomes $\frac{A_{val}}{N_{epoch}}$, with N_{epoch} indicating the minimum number of epochs through which the loss threshold constrained is satisfied during one episode. Our results (Section VIII) indicate that the number of epochs required to train the CNN in an episode finally converges to a minimum value. Hence, our experimental validations indicate that this minimum value should not be lower than three, and we do not look for smaller values once the minimum number of epochs is found to be three.

Algorithm 2: Q-table update rule

```
Input  : $\boldsymbol{Qt_i}, a_i^*$;                 // $i \in \{1, ..., N_{AG} - 1\}$
Output: $\boldsymbol{Qt_i}$ ;                      // Updated Q-table
1 forall $i$ do
2     if $i = N_{AG} - 1$ then
3         $\boldsymbol{Qt_i}[a_i^*, a_{i+1}^*] \leftarrow \boldsymbol{Qt_i}[a_i^*, a_{i+1}^*] + \alpha\mathcal{R}$
4     else
5         if $\exists a_{i+2}$ $st.$ $\boldsymbol{Qt_{i+1}}[a_{i+1}^*, a_{i+2}] = 0$ then
6             $\boldsymbol{Qt_i}[a_i^*, a_{i+1}^*] \leftarrow \boldsymbol{Qt_i}[a_i^*, a_{i+1}^*] + \alpha\mathcal{R}$
7         else
8             $\boldsymbol{Qt_i}[a_i^*, a_{i+1}^*] \leftarrow (1-\alpha)\boldsymbol{Qt_i}[a_i^*, a_{i+1}^*] + \alpha(\mathcal{R} + \gamma \max_{a_{i+2}}(\boldsymbol{Qt_{i+1}}[a_{i+1}^*, a_{i+2}]))$
```

B. Q-table updates

After all agents apply an action to their own layers, a reward is available. This reward is used to update the agents' Q-tables. We propose to follow Algorithm 2 as the Q-table update rules in our specific problem. The main idea of this new Q-table update rule is that if there is any cells that remained with the initial zero value within the next Q-table ($\boldsymbol{Qt_{i+1}}$), then, the current Q-table ($\boldsymbol{Qt_i}$) is updated only according to its own current Q-values and the obtained reward (Lines 5-6). Otherwise, $\boldsymbol{Qt_i}$ is updated by the maximum expected Q-value of $\boldsymbol{Qt_{i+1}}$ and the obtained reward (Line 8). Finally, as shown in Algorithm 2, the last Q-table is updated slightly differently from the others, since there is no Q-table afterward (Lines 2-3).

One of the key elements of the proposed Q-table update rule is the learning rate value, α. We treat this parameter in two different ways in exploration and exploration-exploitation phases. In this context, we initialize α to 0.95 and do not change it during the exploration phase. However, once the exploration-exploitation phase starts, we reduce the learning rate at every episode, as follows:

$$\alpha_{new} = \alpha_{old} \times 0.999^{n_{episode}} \tag{3}$$

where $n_{episode}$ is the number of episodes already passed in the exploration-exploitation phase.

C. Exploration-Exploitation Phase

In this phase, we use a decay function to reduce ϵ and provide a tradeoff between exploration and exploitation similar to Eq. 3. We clarify that, in our work, exploitation does not mean to apply an already taken set of hyperparameters, but to have each agent look into its Q-table shared with the next agent to pick an action that maximizes the expected reward. In this phase, if agents are to exploit their previous experience, Algorithm 3 is used, otherwise, a random action is taken. In the action strategy shown in Algorithm 3, the first two agents take their actions simultaneously to maximize their shared Q-table (Line 3). The next agents, however, always select an action that maximizes a particular row of its Q-table, determined by the previous agent in the sequence. As explained in Section VI-B, each Q-table cell is updated with respect to the next Q-table. Therefore, when in the exploitation phase an agent looks into its own Q-table and selects the best action accordingly, it is, indeed, selecting the one that is expected to benefit the next agent the most. All in all, this

Algorithm 3: Action selection in exploitation phase

```
Input : $Qt_i, A_i = \{a_{i,j}, j \in \{1, ..., N_{A_i}\}\}$
Output: $a_i^*$ ;                    // action of the $i^{th}$ agent
1 forall $i$ do
2     if $i = 1$ then
3         $(a_i^*, a_{i+1}^*) \leftarrow \underset{a_i, a_{i+1}}{\text{argmax}}\, Qt_i[a_i, a_{i+1}]$
4     else
5         $a_{i+1}^* \leftarrow \underset{a_{i+1}}{\text{argmax}}\, Qt_i[a_i^*, a_{i+1}]$
```

procedure most probably provides a completely new set of hyperparameters in the beginning. Due to these new findings of the agents, we keep updating the Q-tables by Algorithm 2. This way, agents can further revise their behavior. Nonetheless, if a hyperparameter set exactly matches a previously experienced one, Q-tables are not updated. In the end, an optimal set of hyperparameters may be selected for several episodes by the agents. This point is where we achieve the convergence of the MARL-based approach.

D. Support for Skip Connections, Residual, Inception, and other Unconventional Modules

Algorithms 1, 2, and 3, as explained in this section, work for all classical CNNs, such as, AlexNet and VGG. However, they require modifications when dealing with more modern CNNs where skip connections and other modules, such as, Inception [42] and Residual [5] are added to the network. Figure 8 shows two examples of different unconventional connections between layers in modern CNNs. In such modules, the layer that feeds multiple layers, or the one that is fed by multiple layers, shares one separate Q-table with the layer to which it is connected.

In Figure 8a, layer m needs to take action and update its shared Q-table with layer $m-1$, with respect to layers $m+1$ to n. First, in Algorithm 1 only Line 3 is affected if $m = 1$. This line changes to the following:

$$\textbf{find} \ \ (a_j^*, a_{j+1}^*) \ \ st. \ \ \boldsymbol{Qt_j}[a_j^*, a_{j+1}^*] = 0, j \in \{1, ..., n\}$$

Second, Lines 5-8 of Algorithm 2 change to the following such that the Q-table shared between layer m and $m-1$, i.e., $\boldsymbol{Qt_{m-1}}$ is updated:

$$\begin{aligned}
&\textbf{if} \quad \exists a_{m+j} \ \ st. \ \ \boldsymbol{Qt_{m-1+j}}[a_m^*, a_{m+j}] = 0, j \in \{1, ..., n\} \\
&\qquad \boldsymbol{Qt_{m-1}}[a_{m-1}^*, a_m^*] \leftarrow \boldsymbol{Qt_{m-1}}[a_{m-1}^*, a_m^*] + \alpha\mathcal{R} \\
&\textbf{else} \\
&\qquad \boldsymbol{Qt_{m-1}}[a_{m-1}^*, a_m^*] \leftarrow (1-\alpha)\boldsymbol{Qt_{m-1}}[a_{m-1}^*, a_m^*] + \\
&\qquad\qquad \alpha(\mathcal{R} + \gamma \max_{a_{m+j}}(\boldsymbol{Qt_{m-1+j}}[a_m^*, a_{m+j}]))
\end{aligned}$$

Finally, only Line 3 in Algorithm 3, where $m = 1$, changes to the following:

$$(a_m^*) \leftarrow \underset{a_m}{\text{argmax}}\, \boldsymbol{Qt_{m+j}}[a_m, a_{m+j}], j \in \{1, .., n\}$$

In Figure 8b, layers m to $n-1$ need to take random actions in the exploration phase, such that their corresponding Q-tables shared with layer n do not have any zeros (Algorithm 1). Then, each $\boldsymbol{Qt_i}, i \in \{1, ..., n-1\}$ is updated according to the maximum Q-value of the Q-table shared between that layer and layer n (Algorithm 2). Finally, if we follow Algorithm 3 to apply action in the exploitation phase, each layer m to $n-1$ may point to a different action in layer n. However, we modify this algorithm such that agent n selects an action that maximizes a greater number of Q-tables shared between layer n and layers m to $n-1$. If such an action does not exist, agent n selects an action that obtains the highest average Q-value among all Q-tables shared.

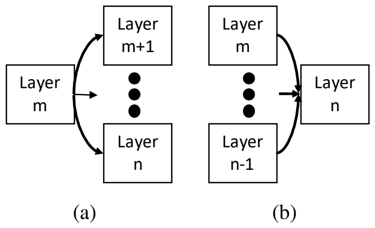


Fig. 8. Two types of unconventional connections in modern CNNs: a) one layer feeds multiple layers, b) one layer is fed by multiple layers

TABLE II
MODEL SETTINGS AND DATASETS

	VGG-16	**GoogLeNet**	**U-Net**
Batch size	128	32	20
Loss	Categorical crossentropy	Categorical crossentropy	Dice
Optimizer	SGD	SGD	Adam
Dataset	CIFAR100	CIFAR100	BRATS'18
	ImageNet	ImageNet	ISIC'18

VII. EXPERIMENTAL SETUP, TEST-CASE DCNNS, AND DATASETS

In this work, we use Keras with Tensorflow backend to implement all test-case CNNs and we perform all experiments on an NVIDIA V100 GPU. In order to show our proposed MARL-based solution is capable of optimizing hyperparameters of CNNs with different architectures, we apply it to U-Net [43], VGG-16 [4], and GoogLeNet [42]. Table II shows the datasets and the settings used to train each CNN. Note that the VGG-16 and GoogLeNet architectures were originally used for ImageNet datasets. To make them compatible to CIFAR100 datasets, we change the size of output *softmax* layer to 100. In addition, in the case of VGG-16, we use only one Dense layer before the output, while we keep the same number of dense layers as in the original architecture for the case of GoogLeNet. Finally, to have a fair comparison with the latest methods [9], [12], [29]–[31], [35], [36], we use data augmentation techniques, such as, rotation, width and height shift, and horizontal flip, on all networks.

By using different datasets on each CNN we show how the proposed MARL-based approach is able to provide a data-driven solution. In particular, the inputs for U-Net are BRATS'18 [18]–[20] and ISIC'18 [21], [22] for semantic segmentation tasks, whereas we consider ImageNet [44] and CIFAR100 [45] for VGG-16 and GoogLeNet as image classification tasks. Although both BRATS'18 and ISIC'18 are biomedical databases, the former is for brain tumor detection and segmentation, while the latter is concerned with skin lesion boundary segmentation. As shown in the table, since U-Net is used for semantic segmentation we consider Dice Loss as the loss metric, whereas categorical cross-entropy is considered for both VGG-16 and GoogLeNet. Note that for training the original networks and those designed by Random Search and

TABLE III
LAYERS AND HYPERPARAMETERS

Layer Type	**Convolution**			**Pooling**	**Dense**
Hyperparameter	**Number of Kernels**	**Kernel Size**	**Stride**	**Size**	**Output Dimension**
VGG-16	{16, 32, 64, 128, 256}	{3,5}	{1,2}	{2,3}	{128,256,512,1024, 2048,4096}
GoogLeNet	{32, 48, 64, 80, 96, 112, 128, 160, 208, 384}	{1,3,5}	{1,2}	{2,3,5,7}	{128,256,512,1024, 2048,4096}
U-Net	{16, 32, 64, 128, 256, 512, 1024}	{3,5,7}	{1,2}	{2,3}	-

our approach, we do not apply any forms of optimizations. In other words, for the sake of fair comparison, we only use the original plain CNN architectures and adapt their hyperparameters through these two approaches. Consequently, by original CNNs, we mean the plain architectures and original hyperparameters without any further optimization. Then, all the networks are trained until the validation accuracy does not improve more than 0.01% for 10 successive epochs.

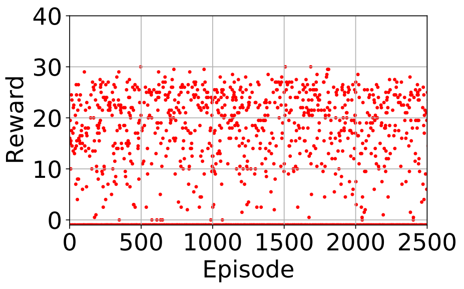
Besides, Table III shows different types of layers and their corresponding hyperparameters considered for each DCNN. We limit the hyperparameter values to those commonly used in the literature for optimizing each DCNN. As shown in the table, unlike VGG-16 and GoogLeNet, U-Net does not contain any dense (fully connected) layer.

VIII. Experimental Results and Discussion

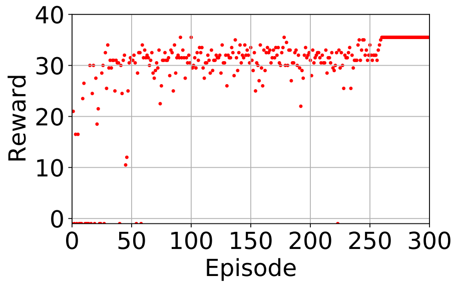
In the following, first, we show the convergence of the proposed MARL-based approach with respect to the reward values. Second, we evaluate the DCNNs designed through our proposed approach compared to the original DCNNs and those designed by Random Search approach. Third, we compare our approach against state-of-the-art neural architecture search and hyperparameter optimization approaches. Next, we discuss the optimization cost and overhead of the proposed approach. Then, we show how the number of episodes in the exploration phase and the number of epochs in each episode may affect the final accuracy and model size of the designed CNN. Finally, we evaluate the impact of the epsilon decay rate in the exploration phase.

A. MARL convergence

Figure 9 shows the reward value during the exploration and exploration-exploitation phases. As shown in Figure 9a, where actions are taken randomly, the reward values range from -1 to 31, where those with -1 belong to hyperparameter sets unable to satisfy the predefined constraints on training time and loss. The exploration-exploitation phase, shown in Figure 9b starts from random actions based on the $\epsilon - greedy$ policy. In the first 50 episodes, where actions are mostly random, small and unsatisfactory rewards are provided. However, for the next episode agents enter the exploitation phase more frequently, as a result of the decayed ϵ value. As shown in this figure, when the agents exploit their Q-tables they are able to statistically obtain higher rewards. With the continuation of the exploitation of the Q-tables, agents are finally able to find the set of actions (hyperparameters) for which the reward is maximized (36). If for a couple of number of consecutive episodes agents pick the same hyperparameters set, then Q-tables have converged. In Figure 9b, the optimal actions are probably found at around episode 260. From this point, agents will take the same actions in the exploitation phase unless, due to the $\epsilon - greedy$ policy, another random action is taken.



(a) Exploration



(b) Exploration-exploitation

Fig. 9. Convergence of proposed MARL-based approach w.r.t reward values

B. Comparison to Random Search and Original Networks

As already highlighted in Table I, the existing works in the literature, most of which focus on the neural architecture search from scratch rather than hyperparameter optimization of the available architectures, do not consider all the aspects we include in this work. In particular, [29]–[33], [35], [36] do not optimize dense layers, only [33] addresses the multi-objective design, [9], [12], [28], [33] fail to address optimization of all modern modules, and none of them consider CNNs for semantic segmentation. As a consequence, we compare the proposed method against Random Search as it is the state-of-the-art baseline solution that can be adapted to consider all the different aspects we address in this work.

Table V compares the accuracy, training/inference time, and model size obtained for each CNN designed through our proposed MARL-based approach, Random Search [6] and the original one (with original hyperparameters in the literature). For Random Search, we find the best hyperparameter set w.r.t the same reward function defined in our MARL-based approach (Section V-E). To measure the accuracy of GoogLeNet and VGG-16, we consider the Top-1 accuracy, whereas, for the case of U-Net, Dice Coefficient is considered.

As shown in the table, for all three DCNNs studied, model size and training/inference time are reduced considerably by our MARL-based approach while the accuracy, in the worst case, is the same as the original DCNNs. Such observation indicates that the proposed reward function (Section V-E) is more biased to optimizing the model size, rather than the accuracy. This bias, however, can be changed by adjusting the coefficients introduced in the reward function.

An important observation, inferred from Table V, is that the proposed approach can improve the accuracy of the original CNNs with respect to those datasets for which they have not been originally designed. Thus, as shown in the table, the accuracy has improved in the case of CIFAR100 for both VGG-16 and GoogLeNet. On the contrary, the accuracy improvement on the ImageNet dataset is insignificant, as these CNNs have been originally optimized for maximizing the accuracy for this particular dataset. Yet, our proposed approach can considerably improve the model size and inference time.

Conversely, Random Search cannot improve the accuracy in any case, compared to the original CNNs. Figure 10 compares the improvements provided by our solution compared to the CNNs designed through Random Search method.

C. *Comparison to State-of-the-Art Neural Architecture Search Methods*

In order to demonstrate the capability of the proposed solution against state-of-the-art neural architecture search and hyperparameter optimization approaches, we consider CNNs designed by these works with respect to the ImageNet dataset. The comparison only considers the ImageNet dataset for classification because, as discussed in Section III, state-of-the-art mainly addresses the automated design of CNNs for image classification. In particular, some of them are limited by a specific layer (e.g., softmax layer) as the termination state of one episode of the exploration phase [12]. Thus, to provide a fair comparison, we consider the ImageNet dataset for classification and avoid any kind of modifications to the state-of-the-arts.

Table IV shows the Top-1 classification accuracy and number of CNN model parameters obtained through the proposed approach and state-of-the-art. The values shown in the table are those either directly reported by the original work (if ImageNet is considered as the test dataset) or extracted from other state-of-the-arts where they made the effort to re-implement those works for ImageNet dataset. As shown in Table IV, not only is the accuracy achieved by our approach close to or even slightly better than the state-of-the-art, but also the number of parameters is considerably reduced.

Nevertheless, the main purpose of our proposed approach is not to design a CNN from scratch, but rather to automate the optimization of an already existing CNN architecture while taking into account a multi-objective design where a compromise between the model size, inference time, and accuracy is vital. Hence, the benefits of our work are better understood if the outcome CNNs are compared with the state-of-the-art CNN architectures. As shown in Table IV, for ImageNet dataset our approach is able to compete against the original design of VGG-16 and GoogLeNet with respect to accuracy with a reduced model size which results in a smaller number of model parameters. In the following section, we detail this comparison.

TABLE IV
COMPARISON WITH STATE-OF-THE-ART NEURAL ARCHITECTURE SEARCH AND HYPERPARAMETER OPTIMIZATION

Approach	Accuracy	#parameters
NAS	72.8	5.3M
PNAS	74.2	5.1M
DARTS	73.1	4.9M
SNAS	72.7	4.3M
MANAS	73.9	2.6M
Block-QNN	75.7	NA[1]
Proposed (VGG)	74.36	3.9M
Proposed (GoogLeNet)	70.31	2.4M
VGG (Original)	74.48	138M
GoogLeNet (Original)	70.02	6.4M

[1] Not reported for ImageNet, but 39M for CIFAR10 dataset

D. *Optimization Cost and Overhead*

We consider 2000, 500, and 4500 random episodes in Random Search and during the exploration phase of our MARL-based approach for U-Net, VGG-16 and GoogLeNet, respectively. These numbers are slightly higher than the minimum number of episodes required for each network and available hyperparameters described in Section VI-A (1764, 400, 4356, respectively). Moreover, the number of epochs for which the CNN is trained in each episode is automatically found to be 3, 6, and 6, respectively, for the case of U-Net, VGG-16, and GoogLeNet for both datasets shown in Table V. Due to the constraints set for training time and validation accuracy, an episode for some of the models is halted and these models are skipped. Table VI shows the amount of time spent for optimizing different CNNs and datasets. When training VGG-16 and GoogLeNet for ImageNet dataset, we do not use the whole 1M training images for the exploration phase, but rather use a subset of 500000 training images, i.e., 500 images per class. This partial observation of the dataset is sufficient for the early rewards given to each model by the agents.

Moreover, state-of-the-art neural architecture search and hyperparameter optimization methods usually evaluate their capacity under simple datasets, such as CIFAR10. Thus, the optimization cost is not available for larger datasets, such as ImageNet, or other CNN tasks including semantic segmentation. Therefore, to have a fair comparison, we consider the optimization cost on CIFAR10 by our work and state-of-the-arts. Table VII shows the GPU days required by different methods, extracted from the original works or from those works which made the effort to re-implement and re-run them on their own experimental setup. Although the GPU types are different, for a small dataset, such as CIFAR10, recent GPU types have negligible impact on the optimization cost. For instance, the optimization cost of our approach is exactly the same on both Nvidia V100 and Tesla T4 for CIFAR10. In addition, we re-ran SNAS [36] on our own experimental setup, confirming that it requires 1.5 GPU days on V100. As shown in this table, the proposed MARL-based approach is able to reduce the optimization cost.

Furthermore, the overhead of our solution compared to Random Search is only $2ms$ for each episode. Therefore, considering that Random Search can also benefit from the same number of episodes we take in both exploration and

TABLE V
EXPERIMENTAL RESULTS: TOP-1 ACCURACY FOR IMAGE CLASSIFICATION AND DICE COEFFICIENT FOR SEMANTIC SEGMENTATION.

CNN	Dataset	Method	Accuracy (%)	Training Time/Batch (ms)	Model Size (MB)	Inference Time/Batch (ms)
VGG-16	CIFAR100	Proposed	73.35	23	7	7
		Original	69.32	37	60	14
		Random Search	63.03	26	13	8
	ImageNet	Proposed	74.36	44	53	16
		Original	74.48	74	141	27
		Random Search	68.70	59	63	22
GoogLeNet	CIFAR100	Proposed	73.48	54	22	13
		Original	70.19	98	48	35
		Random Search	64.68	118	49	40
	ImageNet	Proposed	70.31	35	25	15
		Original	70.02	66	54	23
		Random Search	68.53	49	49	17
U-Net	BRATS'18	Proposed	83.24	98	2	33
		Original	83.25	204	138	68
		Random Search	80.16	157	16	52
	ISIC'18	Proposed	82.35	70	3	18
		Original	81.57	123	138	39
		Random Search	77.84	92	14	31

TABLE VI
OPTIMIZATION COST OF THE PROPOSED APPROACH WITH NVIDIA V100

CNN	Dataset	GPU Hours
VGG-16	CIFAR100	15
	ImageNet	97
GoogLeNet	CIFAR100	257
	ImageNet	1670
U-Net	BRATS'18	61
	ISIC'18	42

TABLE VII
CPU TIME REQUIRED BY DIFFERENT NEURAL ARCHITECTURE SEARCH METHODS

Method	GPU Days	GPU
NAS	2000	P100
ENAS	4	GTX 1080Ti
PNAS	225	P100
MANAS	<2.8 >0.5	GTX 1080
Meta-QNN	100	NA
Block-QNN	96	TitanX
DARTS	4	GTX 1080T
DARTS (w/ few-shot)	1.1	P100
SNAS	1.5	TITAN XP
Proposed (VGG)	0.6	V100

exploration-exploitation phases, the MARL-based approach can come up with the solution with only 4, 17, and 29 extra seconds for VGG-16, GoogLeNet, and U-Net, respectively. These values, nonetheless, are very pessimistic, since during the exploration-exploitation phase, hyperparameters are not taken randomly, and the training time for each episode improves in comparison to that of Random Search. As a result, the MARL-based approach can even spend less wall-clock time for designing DCNNs compared to Random Search.

Finally, the memory overhead of the proposed approach comes from the size of the Q-tables used for the optimization of each CNN, which depends on the number of hyperparameters. In particular, the memory overhead of optimizing GoogLeNet is only 380KB, while this overhead is even smaller for U-Net and VGG-16, with only 255KB, and 164KB, respectively.

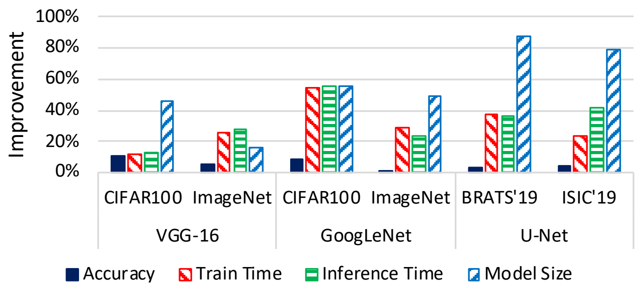
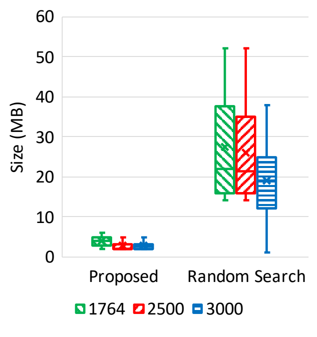


Fig. 10. Improvement in accuracy, model size, and training/inference time provided by MARL-based approach compared to Random Search

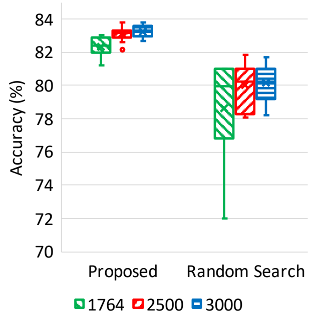
E. Impact of Number of Episodes in Exploration Phase

As explained in Section VI-A, in the exploration phase agents take random actions for quite a large number of episodes. Since each episode contains several epochs, this phase is the most time-consuming part of the proposed MARL-based hyperparameter optimization approach.

Figure 11 shows how the number of episodes in the exploration phase can affect the outcome of the exploration-exploitation phase, and ultimately, the model size and accuracy of the designed DCNN. In the box plot of Figure 11, we consider three different numbers of episodes in designing a U-Net for BRATS'18 dataset, each run 10 times. The first one equals the minimum number of episodes to fill all cells of the Q-tables according to Algorithm 1 and Table III, i.e., $(7\times3\times2)\times(7\times3\times2)$ for two consecutive Convolution layers. Even with this minimum value, the proposed MARL-based approach is able to find a quite satisfying hyperparameter set with respect to the model size and accuracy. However, compared to the other two larger number of episodes, there is more variation in the outcome. Thus, we suggest using more episodes in the exploration phase than the minimum required to fill all Q-tables. Nevertheless, as depicted in Figure 11, although increasing the number of episodes to 2500 provides more consistent results, it only improves trivially when using 3000 episodes. This behavior is, in fact, desirable because it means with a very small increase from the minimum number of episodes, the agents are not only able to improve the outcome but also can provide more statistically consistent results. Hence, there is no need to further increase the time



(a) Model size



(b) Accuracy

Fig. 11. Impact of number of episodes in exploration phase

overhead of the exploration phase.

For comparison, we apply the random search for the same three different number of episodes. As shown in the figure, although with increasing the number of episodes model size and accuracy improves, it is not able to defeat the outcome of the MARL-based solution, nor it can provide more statistically consistent results. Our results reveal that similar trends exist for the other DCNNs and datasets considered in this work.

F. Impact of Number of Epochs in Episodes

As discussed in Section VI-A, in each episode, we train the designed CNN for only a limited number of epochs, automatically found by our proposed solution. Figure 12 shows how this number can affect the outcome of the exploration-exploitation phase and, ultimately, the model size and accuracy of the designed U-Net, VGG-16, and GoogLeNet. In this figure, we show the U-Net outcome for BRATS'18 dataset, whereas, CIFAR100 dataset is considered for both VGG-16 and GoogLeNet. Similar results can be shown for other datasets considered in this work. As shown in Figure 12, the optimal number of training epochs in each episode is less than the maximum number initially defined in Section VI. Moreover, by considering the number of epochs less than 3, 6, and 6, for U-Net, GoogLeNet, and VGG-16, respectively, the agents are not able to well assess the hyperparameters selected at each episode due to the insufficient change in validation accuracy and, thus, the reward signal. Moreover, the impact of the number of epochs is more evident in the accuracy of the designed CNN than in the model size, because the model size is known from the first epoch and does not change with increasing the number of epochs.

G. Impact of Epsilon (ϵ) Decay Rate in Exploration-Exploitation Phase

The best practice for the exploration phase is to let each two successive agents fully explore their actions. Although it is possible to let each agent only partially explore this design space, it most likely results in sub-optimal behavior during the exploitation phase because the impact of some actions remains unknown. The exploration-exploitation phase, however, can take much shorter without a considerable impact on the metrics of the designed CNN if ϵ value is decayed faster. With a faster decay of ϵ, agents can more frequently

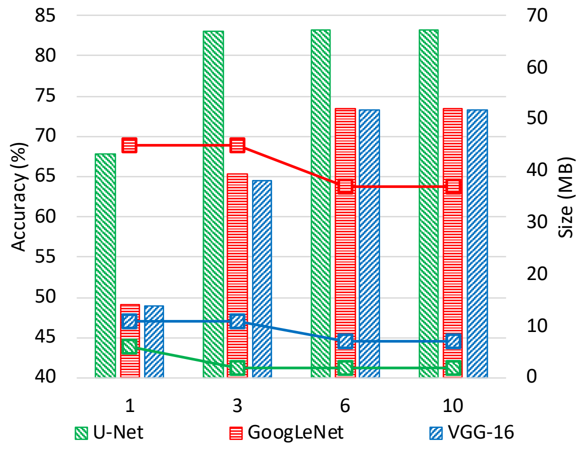


Fig. 12. Impact of number of epochs used for an episode in final model accuracy (bar plots) and size (line plots)

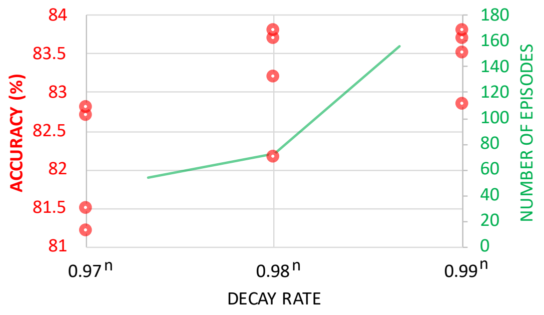


Fig. 13. Impact of decay rate on convergence speed an model accuracy

exploit their past experience than explore new actions in the exploration-exploitation phase. For instance, the number of episodes in exploration-exploitation can be easily cut to half if a decay rate of $0.98^{n_{episode}}$ is used instead of $0.99^{n_{episode}}$. Nonetheless, this faster convergence may also result in more uncertainties in the final accuracy and model size obtained. Figure 13 shows 4 runs of exploration-exploitation phase for 3 different decay rates. As depicted in this figure, with a slower decay rate, the number of episodes for convergence increases while statistically higher accuracy can be achieved. On the other hand, with a faster decay rate, fewer number of episodes are required for convergence of the exploration-exploitation phase with the cost of attaining statistically lower model accuracy.

IX. Conclusion

In this work, we have addressed hyperparameter optimization of Deep CNNs (DCNNs) through a novel Multi-Agent Reinforcement Learning (MARL)-based approach. This approach uses different Q-Learning agents per layer to split the design space into smaller independent action sub-spaces to provide faster, yet accurate design space search. In contrast to the state of the art, our approach is not limited to particular types of layers, can scale well with the depth of CNNs without any search time overhead, and can optimize the CNN hyperparameters with respect to any arbitrary set of constraints and objectives, thus, eliminating the time-consuming and manual human effort. We assessed our MARL-based approach by applying it to three different CNNs, VGG-19, GoogLeNet, and U-Net, each with two different datasets. Our results have shown that, compared to the original CNNs, the MARL-based approach can reduce the model size, training time, and inference time by up to, respectively, 83x, 52%, and 54% without any degradation in accuracy. Moreover, our approach is very competitive to the state-of-the-art neural architecture search methods in terms of model accuracy and number of model parameters, while it can considerably reduce the GPU time required for CNN optimization.

Acknowledgement

This work has been supported by the ERC Consolidator Grant COMPUSAPIEN (GA No. 725657), the H2020 DeepHealth Project (GA No. 825111), and an Unrestricted Research gift by the AI Hardware Infrastructure Unit of Facebook for ESL-EPFL.

References

[1] M. Z. Alom, T. M. Taha, C. Yakopcic, S. Westberg *et al.*, "The history began from alexnet: A comprehensive survey on deep learning approaches," *arXiv preprint arXiv:1803.01164*, 2018.

[2] Y. LeCun, B. Boser, J. S. Denker, D. Henderson *et al.*, "Backpropagation applied to handwritten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[3] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[4] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[5] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[6] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of machine learning research*, vol. 13, no. Feb, pp. 281–305, 2012.

[7] H. Jin, Q. Song, and X. Hu, "Auto-keras: An efficient neural architecture search system," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 1946–1956.

[8] K. Kandasamy, W. Neiswanger, J. Schneider, B. Poczos *et al.*, "Neural architecture search with bayesian optimisation and optimal transport," in *Advances in Neural Information Processing Systems*, 2018, pp. 2016–2025.

[9] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," *arXiv preprint arXiv:1611.01578*, 2016.

[10] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," in *Proceedings of the aaai conference on artificial intelligence*, vol. 33, 2019, pp. 4780–4789.

[11] M. Feurer, B. Letham, and E. Bakshy, "Scalable meta-learning for Bayesian optimization," *arXiv preprint arXiv:1802.02219*, 2018.

[12] B. Baker, O. Gupta, N. Naik, and R. Raskar, "Designing neural network architectures using reinforcement learning," *arXiv preprint arXiv:1611.02167*, 2016.

[13] L. Buşoniu, R. Babuška, and B. De Schutter, "Multi-agent reinforcement learning: An overview," in *Innovations in multi-agent systems and applications-1*. Springer, 2010, pp. 183–221.

[14] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[15] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[16] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*. Springer, 2010, pp. 177–186.

[17] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *International Conference on Medical image computing and computer-assisted intervention*. Springer, 2015, pp. 234–241.

[18] B. H. Menze, A. Jakab, S. Bauer, J. Kalpathy-Cramer *et al.*, "The multimodal brain tumor image segmentation benchmark (brats)," *IEEE transactions on medical imaging*, vol. 34, no. 10, pp. 1993–2024, 2014.

[19] S. Bakas, H. Akbari, A. Sotiras, M. Bilello *et al.*, "Advancing the cancer genome atlas glioma mri collections with expert segmentation labels and radiomic features," *Scientific data*, vol. 4, p. 170117, 2017.

[20] S. Bakas, M. Reyes, A. Jakab, S. Bauer *et al.*, "Identifying the best machine learning algorithms for brain tumor segmentation, progression assessment, and overall survival prediction in the brats challenge," *arXiv preprint arXiv:1811.02629*, 2018.

[21] P. Tschandl, C. Rosendahl, and H. Kittler, "The ham10000 dataset, a large collection of multi-source dermatoscopic images of common pigmented skin lesions," *Scientific data*, vol. 5, p. 180161, 2018.

[22] N. Codella, V. Rotemberg, P. Tschandl, M. E. Celebi *et al.*, "Skin lesion analysis toward melanoma detection 2018: A challenge hosted by the international skin imaging collaboration (isic)," *arXiv preprint arXiv:1902.03368*, 2019.

[23] T. Elsken, J. H. Metzen, and F. Hutter, "Neural architecture search: A survey," *arXiv preprint arXiv:1808.05377*, 2018.

[24] G. F. Miller, P. M. Todd, and S. U. Hegde, "Designing neural networks using genetic algorithms." in *ICGA*, vol. 89, 1989, pp. 379–384.

[25] M. Suganuma, S. Shirakawa, and T. Nagao, "A genetic programming approach to designing convolutional neural network architectures," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2017, pp. 497–504.

[26] R. S. Olson and J. H. Moore, "Tpot: A tree-based pipeline optimization tool for automating machine learning," in *Automated Machine Learning*. Springer, 2019, pp. 151–160.

[27] Q. Huang, K. Zhou, S. You, and U. Neumann, "Learning to prune filters in convolutional neural networks," in *2018 IEEE Winter Conference on Applications of Computer Vision (WACV)*. IEEE, 2018, pp. 709–718.

[28] H. Cai, T. Chen, W. Zhang, Y. Yu *et al.*, "Efficient architecture search by network transformation," in *Thirty-Second AAAI conference on artificial intelligence*, 2018.

[29] H. Liu, K. Simonyan, and Y. Yang, "Darts: Differentiable architecture search," *arXiv preprint arXiv:1806.09055*, 2018.

[30] Z. Zhong, J. Yan, W. Wu, J. Shao *et al.*, "Practical block-wise neural network architecture generation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 2423–2432.

[31] C. Liu, B. Zoph, M. Neumann, J. Shlens *et al.*, "Progressive neural architecture search," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 19–34.

[32] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le *et al.*, "Efficient neural architecture search via parameter sharing," *arXiv preprint arXiv:1802.03268*, 2018.

[33] C.-H. Hsu, S.-H. Chang, J.-H. Liang, H.-P. Chou *et al.*, "MONAS: Multi-objective neural architecture search using reinforcement learning," *arXiv preprint arXiv:1806.10332*, 2018.

[34] Y. Zhao, L. Wang, Y. Tian, R. Fonseca *et al.*, "Few-shot neural architecture search," *arXiv preprint arXiv:2006.06863*, 2020.

[35] F. M. Carlucci, P. Esperanca, R. Tutunov, M. Singh *et al.*, "Manas: multi-agent neural architecture search," *arXiv preprint arXiv:1909.01051*, 2019.

[36] S. Xie, H. Zheng, C. Liu, and L. Lin, "SNAS: stochastic neural architecture search," *arXiv preprint arXiv:1812.09926*, 2018.

[37] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press Cambridge, 1998, vol. 1, no. 1.

[38] K. Zhang, Z. Yang, and T. Başar, "Multi-agent reinforcement learning: A selective overview of theories and algorithms," *arXiv preprint arXiv:1911.10635*, 2019.

[39] Y. Shoham and K. Leyton-Brown, *Multiagent systems: Algorithmic, game-theoretic, and logical foundations*. Cambridge University Press, 2008.

[40] M. Lanctot, E. Lockhart, J.-B. Lespiau, V. Zambaldi *et al.*, "Openspiel: A framework for reinforcement learning in games," *arXiv preprint arXiv:1908.09453*, 2019.

[41] F. A. Oliehoek and C. Amato, *A concise introduction to decentralized POMDPs*. Springer, 2016.

[42] C. Szegedy, W. Liu, Y. Jia, P. Sermanet *et al.*, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[43] P. F. Christ, M. E. A. Elshaer, F. Ettlinger, S. Tatavarty *et al.*, "Automatic liver and lesion segmentation in ct using cascaded fully convolutional neural networks and 3d conditional random fields," in *International Conference on Medical Image Computing and Computer-Assisted Intervention*. Springer, 2016, pp. 415–423.

[44] J. Deng, W. Dong, R. Socher, L.-J. Li *et al.*, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.
[45] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

Arman Iranfar received his PhD in electrical engineering from the Swiss Federal Institute of Technology Lausanne (EPFL). He is currently a postdoctoral researcher at Embedded Systems Laboratory (ESL), EPFL. His research interest includes applied machine learning and reinforcement learning in multi-objective management of MPSoCs. He has published over 14 peer-reviewed papers in top-notch conferences and journals and served as reviewer in several top-ranked conferences and journals in including IEEE TC, IEEE TSUSC, and IEEE TSC.

Marina Zapater (M'16) is Associate Professor in the School of Engineering and Management of Vaud (HEIG-VD), in the University of Applied Sciences Western Switzerland (HES-SO) since 2020, and Research Associate in the Embedded System Laboratory (ESL) at the Swiss Federal Institute of Technology Lausanne (EPFL), Switzerland, since 2016. She received her Ph.D. degree in Electronic Engineering from Universidad Politécnica de Madrid, Spain, in 2015. Her research interests include thermal, power and performance design and optimization of complex heterogeneous architectures, from embedded edge devices to high-performance computing processors; and energy efficiency in servers and data centers. In these fields, she has co-authored more than 50 papers in top-notch conferences and journals, She is an IEEE and CEDA member, and has served as CEDA YP representative (2019-2020).

David Atienza (M'05-SM'13-F'16) is an associate professor of electrical and computer engineering, and head of the Embedded Systems Laboratory (ESL) at the Swiss Federal Institute of Technology Lausanne (EPFL), Switzerland. He received his PhD in computer science and engineering from UCM, Spain, and IMEC, Belgium, in 2005. His research interests include system-level design methodologies for high-performance multi-processor system-on-chip (MPSoC) and low power Internet-of-Things (IoT) systems, including new 2-D/3-D thermal-aware design for MPSoCs and many-core servers, and edge AI architectures for wireless body sensor nodes and smart consumer devices. He is a co-author of more than 350 papers in peer-reviewed international journals and conferences, one book, and 12 patents in these fields. Dr. Atienza has received the ICCAD 2020 10-Year Retrospective Most Influential Paper Award, the DAC Under-40 Innovators Award in 2018, the IEEE TCCPS Mid-Career Award in 2018, an ERC Consolidator Grant in 2016, the IEEE CEDA Early Career Award in 2013, the ACM SIGDA Outstanding New Faculty Award in 2012, and a Faculty Award from Sun Labs at Oracle in 2011. He served as DATE 2015 Program Chair and DATE 2017 General Chair. He is an IEEE Fellow, an ACM Distinguished Member, and served as IEEE CEDA President (period 2018-2019).