Designing a Classifier for Active Fire Detection From Multispectral Satellite Imagery Using Neural Architecture Search

Amber Cassimon, Phil Reiter, Siegfried Mercelis, and Kevin Mets

***Abstract*—Wildfires are becoming increasingly devastating, and detecting them early is essential to containing them. Deep learning-based wildfire detection systems have increased in complexity dramatically in recent years, and in order to manage this added complexity, techniques have been proposed to automate the design of neural network architectures. Such techniques are usually referred to as neural architecture search (NAS). This article showcases the use of a reinforcement learning-based neural architecture search (NAS) agent to design a small neural network to perform active fire detection on multispectral satellite imagery. Specifically, we aim to automatically design a neural network that can determine if a single multispectral pixel is a part of a fire, and do so within the constraints of a low earth orbit nanosatellite with a limited power budget, to facilitate on-board processing of sensor data. A regression model that predicts the F1 score obtained by a particular architecture following quantization is used as a reward function. This model is trained on the classification performance statistics of a sample of neural network architectures. Besides the F1 score, we also include the total number of parameters in our reward function to limit the size of the designed model. Finally, we deployed the best neural network to the Google Coral Micro Dev Board and evaluated its inference latency and power consumption. This neural network consists of 1716 parameters, takes on average 984 μs to inference, and consumes around 800 mW to perform inference. These results show that our approach can be applied to new problems.**

***Index Terms*—Active fire detection, AutoML, deep learning, multispectral imaging, neural architecture search (NAS).**

I. Introduction

MULTISPECTRAL satellite data have many uses ranging from estimating heat storage in urban areas [1], bathymetry [2], or monitoring the evolution of rivers [3]. However, analyzing large volumes of data by hand is cumbersome. Because of this, deep learning techniques have been successfully introduced to automate the analysis of multispectral satellite imagery [4]. While accurate analyses can be made using deep learning, these neural networks often come with high computational costs. This makes deep learning approaches infeasible in environments and tasks where computational resources and power are at a premium, such as when performing processing of multispectral satellite imagery on-board smallsats [5], [6]. Daghouri et al. [5] sized the electrical power system for a nano satellite at several watts, ranging from around 6 W per solar panel at peak times, down to just 1.7 W during low times. Dahbi et al. [6] arrived at a similar number, with their nanosatellite generating between 0 and 3.5 W depending on the precise position of the satellite in an orbit. Deep learning systems can be designed to operate in such low-power environments, but this process is often complex and laborious, requiring experienced engineers to iteratively design neural networks that maximally take advantage of the available resources [7]. To remedy this, various aspects of the design of neural networks, such as hyperparameter selection [8] and architecture selection [9], have been automated. The automated selection of an appropriate neural network architecture is usually referred to as neural architecture search (NAS), and various techniques have been used for this, including continuous relaxation [10], reinforcement learning [11], evolutionary algorithms [12], and Bayesian optimization [13]. While reinforcement learning has fallen out of favor in recent years [14], novel reinforcement learning-based techniques show promising results on established NAS benchmarks [15], [16]. This article aims to validate the performance of such reinforcement learning agents on a practical use-case. The use-case we selected is the detection of active fires from multispectral satellite imagery within a power envelope that would allow operation on-board a smallsat. Neural networks are selected to maximize both classification performance (as measured by the F1 score) and minimize the number of trainable parameters to reduce power consumption. The neural network that is finally selected is deployed onto a Google Coral Micro development board [17], where the power consumption of the whole system is measured to verify that it could indeed be operated within the power envelope of a smallsat system.

In summary, the contributions this article makes are as follows.

1) The performance of the reinforcement learning agent introduced by Cassimon et al. [15] is validated by assessing

Received 11 October 2024; revised 21 January 2025 and 26 February 2025; accepted 28 March 2025. Date of publication 1 April 2025; date of current version 21 April 2025. This work was supported in part by Research Foundation Flanders under Grant 1SC8821N, in part by Flemish Government (AI Research Program) MOVIQ (Mastering Onboard Vision Intelligence and Quality) funded by Flanders Innovation & Entrepreneurship (VLAIO) and Flanders Space (VRI), and in part by European Union NextGenerationEU. *(Corresponding author: Amber Cassimon.)*

The authors are with the IDLab—Faculty of Applied Engineering, University of Antwerp—imec, 2000 Antwerpen, Belgium (e-mail: amber.cassimon@uantwerpen.be).

Digital Object Identifier 10.1109/JSTARS.2025.3556550

it on a problem that is not an NAS benchmark for the first time.
2) To the best of the authors' knowledge, for the first time, a performance prediction model is trained to predict neural network performance after trainable parameters are quantized.
3) A novel, resource-efficient neural network architecture is found for detecting active wildfires from multispectral satellite imagery.
4) NAS is applied to the problem of active fire detection from multispectral satellite imagery for the first time.

The rest of this article is organized as follows. In Section II, we consider state-of-the-art research in various fields related to our use-case. Section III details how the NAS agent was designed, and how we achieved the prerequisites. Next, Section IV discusses the experiments we performed, listing experimental setup, and used parameters in detail. We discuss the results obtained from these experiments in Section V. Finally, Section VI concludes this article.

II. Related Work

This section will discuss existing approaches in the field of reinforcement learning-based NAS (see Section II-A), multispectral image processing (see Section II-B), and active fire detection (see Section II-C).

A. Neural Architecture Search

NAS has been used to design neural networks that outperform human-designed neural networks in a wide variety of domains, including computer vision [18], natural language generation [19], and wind forecasting [20]. The variety of techniques that have been used in NAS is almost as wide as the set of problem domains that have been tried, including Bayesian approaches [13], evolutionary algorithms [12], [18], continuous relaxation [10], graph diffusion [21], and reinforcement learning [11], [22]. Despite having fallen out of use in recent years, recent innovations in the space of reinforcement learning-based NAS approaches encouraged us to use reinforcement learning in this work. Since this article uses a reinforcement learning-based approach, we will focus on reinforcement learning for the remainder of this section.

Some of the first work in the field of NAS was done using reinforcement learning [9], [11]. The authors in [9] and [11] used a long short-term memory-based reinforcement learning agent to sequentially sample architectural decisions and build the computational graph of the neural network. They evaluated their approach in both the language generation and computer vision domains and found their approach surpassed human-designed neural networks, achieving strong performance in both domains. Pham et al. [11] considered both a macro search space and a micro search space. In the macro search space, their reinforcement learning agent designed the entire neural network, while in the micro search space, it designed a small cell that was repeated multiple times to form a complete neural network.

Cassimon et al. [15] introduced a novel reinforcement learning agent that iteratively improves on a given neural network architecture by making small alterations to the architecture. Their transformer-based reinforcement learning agent is evaluated on two NAS benchmarks focused on computer vision applications: NAS-Bench-101 [23] and NAS-Bench-301 [24]. They find that their agent is capable of finding strong architectures on both benchmarks and scales well with the size of the search space. Contrary to this work, Cassimon et al. [15] made use of a look-up table and a pretrained gradient boosted tree model as their reward function for the NAS-Bench-101 and NAS-Bench-301 benchmarks, respectively.

NAS methods require a method to find the performance of a specific neural network architecture on a particular task. The most naive way of achieving this is by simply training neural networks to convergence, as was done in early NAS works [9]. In recent literature, this is usually achieved through performance prediction methods: methods designed to predict the performance of a specific neural network on a specific task, assuming a particular fixed training procedure and set of hyperparameters. Some methods make use of a regression model that takes architectural features as input and outputs a prediction for the neural network's target performance [25]. Others use what are called zero-cost proxies: methods that do not require any training (of the designed network or the performance predictor), but usually do require inference [26], [27]. There also exist one-shot methods that rely on concepts such as weight sharing [11].

Lu et al. [25] proposed a transformer-like NAS performance predictor, which uses permutation–invariance modules to improve predictor performance in the face of graph isomorphism between different representations of the same architecture. They test their method on several existing benchmarks, including NAS-Bench-101 [23] and NAS-Bench-201 [28]. Their PINAT method is also evaluated on the DARTS [10] and Proxyless-NAS [29] search spaces. The architectures designed by PINAT achieve performance rivaling that of other state-of-the-art methods.

B. Multispectral Image Processing

Multispectral satellite imagery can be used for a broad variety of use-cases. This section assesses several relevant works in the field of multispectral image processing that concern applications other than active fire detection.

Zheng et al.[30] presented a framework for patch-free global learning of hyperspectral imagery. They achieve this using an encoder–decoder architecture enhanced using lateral connections. The encoder maps the entire image into a latent space, while the decoder decodes the latent image representation into a per-pixel classification map. Their method is supported by a global stochastic stratified sampling strategy to ensure a diversity of gradients and prevent convergence issues due to a low amount of training samples. Because hyperspectral datasets, such as Pavia University [31], only contain a single image, training must also occur in batches with a batch size of 1. This can lead to issues with batch normalization operations included in many models; thus, the authors opt to replace batch normalization with group normalization [32]. They also introduce a spectral attention module that was included in the search space in this

work. The modules are designed to reweight the feature maps of hyperspectral images based on their importance. Through these techniques, they obtain strong results on the Pavia University [31], Salinas [33], and CASI University of Houston datasets [34]. This article also demonstrates the computational efficiency of their approach by a comparison in terms of the number of floating point operations (FLOPs) and the number of trainable parameters.

Wu et al. [35] presented a U-Net-based approach to detecting the severity of burned areas following a wildfire. Contrary to our work, Wu et al. [35] focused on damage assessment following a fire, rather than detecting ongoing fires. Their U-Net architecture includes several enhancements, such as attention gates in the decoder and the use of residual blocks. The satellite data used are Sentinel-2 L2A bottom of atmosphere (BOA) data, including the use of bitemporal imagery augmented with the normalized burn ratio (NBR) [36] index and some of its derivatives. Three spectral bands (B8A, B11, and B12) are selected for inclusion in the input data to the model. They evaluate the different improvements to the standard U-Net architecture separately and consider the efficacy of different loss functions. To enhance the generalization ability of their model, histogram matching is used, and the effect of this enhancement is investigated quantitatively. The training and test datasets are also swapped to evaluate the ability of the model to be trained with limited data. The model is also evaluated as a binary classification model, disregarding the distinction between the different severity levels of burned areas. Finally, the annotation method is also evaluated on a highly accurate, expert-annotated dataset to assess the quality of the annotated data used to train the neural network. At the end of their related work section, Wu et al. [35] noted the need for more systematic studies assessing the complex architectures of deep learning models, further strengthening the motivation for this publication.

C. Active Fire Detection

In this section, we will consider the state of the art in active fire detection from multispectral satellite imagery. While research exists into active fire detection methods for terrestrial purposes [37], this article focuses on the detection of fires from satellite imagery.

Barmpoutis et al. [38] provided an overview of methods for early fire detection based on optical remote sensing. They categorize methods in terms of where the sensors are placed: terrestrial, airborne, or space borne. They consider optical remote sensing systems operating in the visible and infrared (IR) spectrum, as well as multispectral systems based on traditional machine learning and deep learning. In their conclusion, Barmpoutis et al. [38] highlighted the smallsat-based fire detection system as a promising avenue for improving the detection of active wildfires.

Florath and Keller [39] built a system to detect active fires and burnt areas simultaneously based on supervised machine learning. They tackle several challenges, including the generation of good reference data and the detection of active fires and burned areas at a high spatial resolution, while at the same time trying to keep their methodology as generic as possible. To generate useful data for their machine learning models, they leverage a combination of Open Street Maps data and vector data from governmental agencies and Sentinel-2 L2A BOA products. Florath and Keller [39] evaluated seven different supervised machine learning models, including gradient boosting, extremely randomized trees, multilayer perceptrons, and convolutional neural networks (CNNs). They conclude that the performance of all models is satisfactory when it comes to detecting fire, but find most models struggle more with the classification of burned areas. They hypothesize that the models struggle with the separation of burned and unburned areas because of their spectral similarity.

To gather sufficient labeled data to train a deep learning system for active fire detection, we use the algorithm that Massimetti et al. [40] used to detect and monitor volcanic activity. Meoni et al. [41] found this algorithm useful for detecting fire events on Sentinel-2 data. The algorithm proposed by Massimetti et al. [40] is relatively basic. It consists of the computation of reflectance ratios between different bands of Sentinel-2 data. These reflectance ratios are then compared against reference thresholds, and the result of these comparisons is combined through a series of distinct logical tests. Four different logical tests are used, and if at least one of them returns true, a pixel is flagged as a hotspot.

Xu and Wooster [42] discussed the challenges faced in developing active fire detection products from Sentinel-3 data. Sentinel-3 carries various instruments and can capture information in IR channels with a spatial resolution of 1 km [43]. They form a daytime active fire detection product by combining different spectral bands and adapting existing active fire detection methods designed for nighttime detection of active fires based on a single IR band [44]. The original active fire detection algorithm relies on a series of masking operations using varying thresholds, atmospheric corrections, and cluster detections. The new active fire detection product is compared against existing products and found to have similar performance, with nuanced differences in which fires are detected by both products.

Maillard et al. [45] made use of an object detection neural network to detect active wildfires from a variety of data sources, including satellite imagery, thermal imagery, aerial imagery, and images captured from ground-based cameras. The neural network they use is the YOLO-NAS network, an object detection network created by Deci AI [46] using a proprietary NAS algorithm called AutoNAC. The authors note that they achieved mixed results, with strong mean average precision and recall, but low F1 and precision. In their publication, a heterogeneous dataset was used, while this improved applicability in real-world use-cases, careful evaluations are necessary due to large discrepancies between different image types.

Rashkovetsky et al. [47] evaluated the use of four different sources of satellite imagery to perform active fire detection. They train a U-Net [48] for each data source and compare the performance of different U-Nets under cloudy and clear conditions. The paper also investigates the fusion of pairs of data sources to enhance detection performance, including an investigation under clear and cloudy conditions. The data used were taken from the Sentinel -1, -2, and -3 satellites, as well

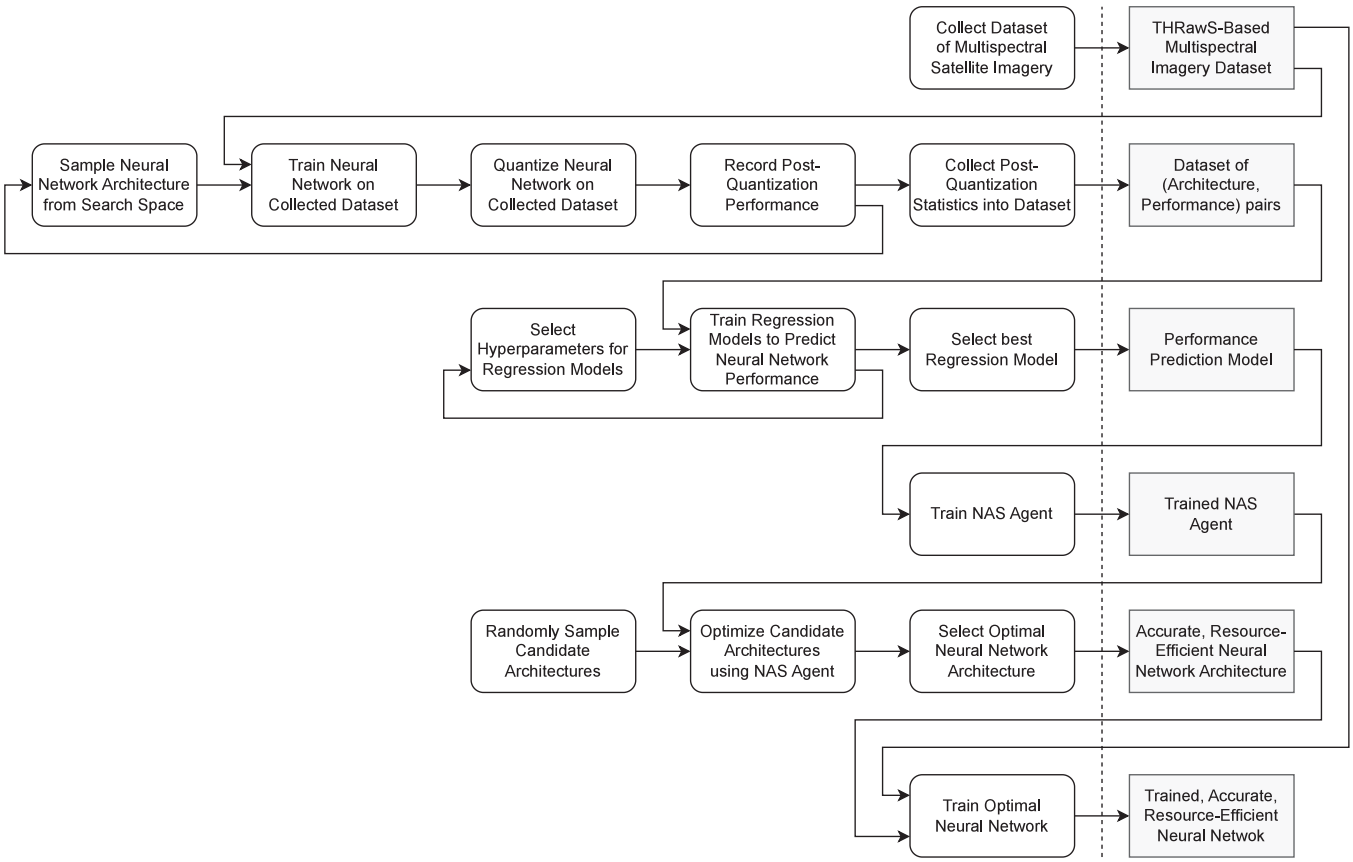


Fig. 1. Schematic overview of the complete methodology. Steps taken are shown on the left, while the result of each step is shown on the right.

as from the MODIS instrument onboard the Terra and Aqua satellites. They conclude that in a single data source scenario with clear weather, Sentinel-2 is the most suitable data source, achieving high detection accuracy. In a scenario with multiple data sources, Sentinel-2 data seem to benefit most from other data sources, such as Sentinel-1 and Sentinel-3.

III. Methods

This section will describe the various aspects of our research in more detail, including the target task considered (see Section III-A), the way we gathered data on the training performance of our neural networks (see Section III-B), the training procedure for our performance predictors (see Section III-C), and the training procedure for our NAS agent (see Section III-D).

First, Fig. 1 provides an overview of the complete methodology. The left-hand side of the figure describes the various steps taken, while the right-hand side shows the results from the different steps, and how they are used by subsequent steps.

The first step is the collection of a dataset of multispectral satellite imagery from the THRawS dataset [41]. The details of this collection process are given in Section III-A.

Following this, we randomly sample a subset of neural network architectures from the search space outlined in Section III-B. This set of neural networks is trained on the dataset of multispectral imagery gathered in the previous step. Following training, the neural networks are quantized, and their classification performance is recorded along with their architecture. This dataset of neural network architectures combined with their classification performance will be used in the next step.

Next, a regression model will be trained to predict the postquantization classification performance of neural network architectures that were not trained before. Multiple regression models will be trained on the dataset gathered in the previous step. The process for training these regression models is elaborated in Section III-C.

Using this regression model as a reward function, a reinforcement learning agent is trained to incrementally design neural networks optimal for the task of detecting fire from multispectral satellite imagery. The reward function is augmented to also consider the computational requirements of the designed neural networks. Once trained, another subset of architectures will be sampled at random from the search space. The reinforcement learning agent will improve each architecture incrementally until an optimum is reached. The architecture with the highest predicted classification performance following optimization by the reinforcement learning agent will be selected for deployment. Section III-D covers the procedure for training the reinforcement learning agent.

As a final step, the selected neural network will be trained with several random initializations. The trained neural network with the highest classification performance will then be deployed onto a Google Coral Dev Board Micro.

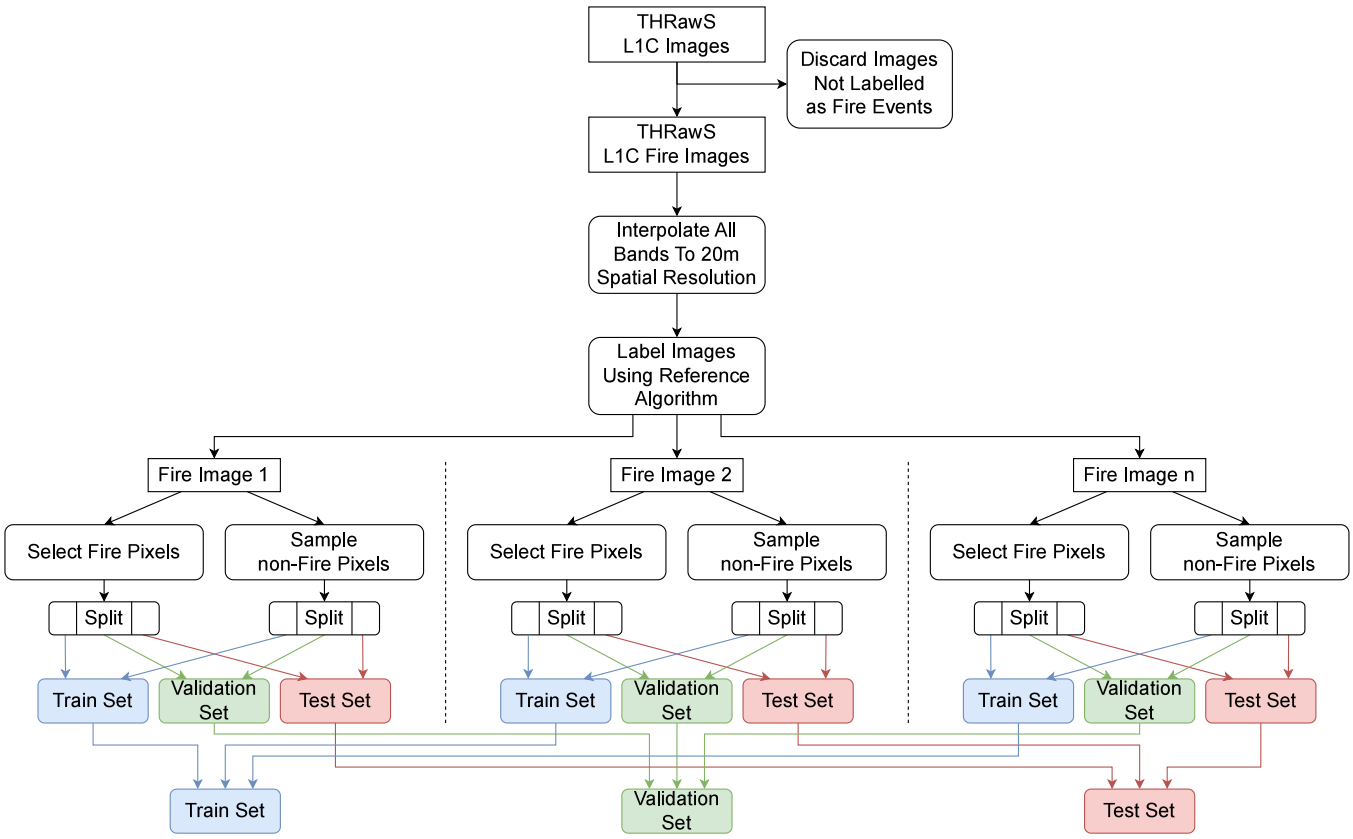


Fig. 2. Schematic overview of the procedure used to generate the dataset of multispectral satellite images.

A. *Target Task*

The task considered in this article is the detection of active wildfires from Sentinel-2 multispectral satellite imagery. The dataset used was a subset of the THRawS dataset [41]. Specifically, we selected all tiles that contain "fire" events at the time of writing. We use L1C top of atmosphere products, following Rashkovetsky et al. [47], since they do not require the computationally expensive atmospheric correction process necessary for L2A BOA products [41]. Wildfires will be detected on a per-pixel basis, i.e., we will consider the spectral data for a single pixel at a time (13 bands) and output a binary classification: a pixel is labeled as either "fire" or "no fire." Following Meoni et al. [41], we use the algorithm from [40] to obtain a binary classification mask for the entire image; specifically, we use the implementation provided by PyRawS [49]. Given that the different spectral bands in a Sentinel-2 image have different spatial resolutions, we resample all bands to the 20 m resolution. This follows the 20 m spatial resolution used by Massimetti et al. [40] in their hotspot detection algorithm. In the THRawS satellite images, negative label ("no fire") pixels significantly outnumber positive label pixels. To compensate for this, we first select all fire pixels, and then randomly sample an equal number of "no fire" pixels at random from each satellite image. This is done on a per-image basis to ensure that there is no distribution shift in regard to geographic regions, vegetation, etc., between both the positive and negative pixels. Following this, in each image individually, positive and negative pixels are split in a 70%/15%/15% fashion to create a training, validation, and test subset. The individual training, validation, and test subsets of all images are then merged into datasets that span all images. This procedure is illustrated in Fig. 2.

To aid our classification algorithm, we include three indices as a form of feature engineering. Specifically, we compute the NBR [36], normalized difference vegetation index (NDVI) [50], and Active Fire Detection Index for Sentinel-2 (AFD) [51], and append these as features to our pixel data to improve classification performance, giving each pixel a total of 16 features (13 spectral bands and three indices). Since our goal is to eventually deploy the designed neural networks to an embedded device, we include the computation of these indices as part of the model. This ensures that the device can accept data without requiring that preprocessing is done externally.

B. *Performance Data Gathering*

NAS algorithms require a way to evaluate the performance of a neural network, preferably without training it to convergence. While many approaches have been proposed over the years, including one-shot methods [11] and zero-cost proxies [26], in the context of this work, we opted for a performance prediction model [52]. To train such a model, we require performance data from trained neural networks. Often, gathering large amounts of such training data is computationally expensive due to the

computational requirements of a single neural network. For our work, however, the computational resources needed for training a single neural network are very limited, given that the network must be able to operate in a power envelope of several watts. This allows us to more easily gather performance data by training many neural networks. Alternative approaches, such as one-shot methods exist, but one-shot methods are complex to train correctly, and if done incorrectly, yield low ranking correlation with the ground-truth data [53]. Zero-cost proxies are another alternative, but they often require instantiating the designed neural network and performing inference on it [27]. These limitations make performance prediction models an attractive alternative to one-shot methods and zero-cost proxies in our case.

Following training, neural networks were quantized to INT8 precision using post-training quantization, with the aim of eventually deploying them on the Google Coral Micro Dev Board.

Our search space is a macro search space, which requires selecting the topology and node labels of a computational graph, similar to the cells considered by Ying et al. [23]. We consider architectures with up to eight nodes, including one input and output node, leaving six nodes requiring operation labels to be assigned by the NAS agent. Node labels are selected from a set of ten possible labels: "linear-prelu," "linear-relu," "linear-relu6," "linear-tanh," "linear," "conv-3," "conv-5," "max-pool-3," "max-pool-5," and "spectral-attn." Preliminary experiments showed that linear classifiers can perform fairly well with the features we use; thus, we opted to include the "linear" operation, which is a simple linear transformation without nonlinearity, to offer the NAS agent the freedom to design both linear and nonlinear classifiers. We operate on a single pixel at a time, rather than on a patch of multiple pixels as is often the case. Given that we have a single flat feature vector, we opted to include a linear layer with a variety of nonlinearities in the form of the "linear-prelu," "linear-relu," "linear-relu6," and "linear-tanh," operations. Since spectral features in our dataset are roughly ordered by frequency, the first 13 features do include a notion of locality. To exploit this locality and allow the exploitation of relations between neighboring frequency bands, we included both 1-D convolution and pooling operations in the search space through the "conv-3" and "conv-5," "max-pool-3," and "max-pool-5" operations. Finally, we also include the spectral attention operation first proposed by Zheng et al. [30]. From this information, we can compute an upper bound on the size of the search space, Ω. Our neural networks have between 2 and 8 vertices ($v \in [2, 8]$). For each vertex count, we need to label $v-2$ nodes with one of ten ($|o| = 10$) operations. v vertices in a directed acyclic graph (DAG) can be connected with between $v-1$ and $\frac{v\cdot(v-1)}{2}$ edges ($e \in [v-1, \frac{v\cdot(v-1)}{2}]$). The number of ways we can sample d edges from a set of e possible edges can be expressed mathematically as a combination. Combined, this leads to the expression in (1) as an upper bound on the size of the search space. In reality, the number of unique architectures in the search space will be lower, since not every set of edges is a valid set of edges, and isomorphism has not been accounted for. Still, this number can give us an idea of the size of the search space we are operating in

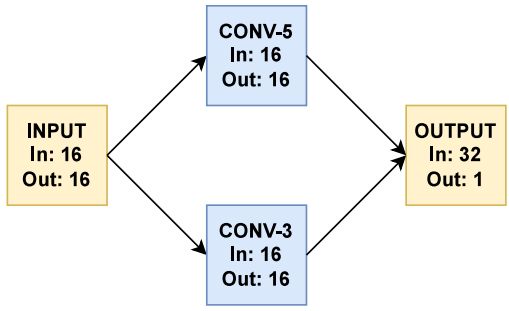


Fig. 3. Example network involving multiple inputs to the output node.

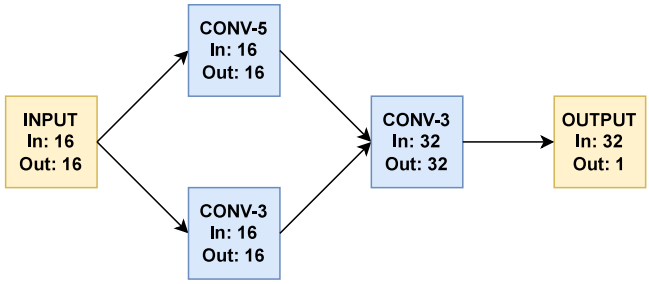


Fig. 4. Example network involving multiple inputs to an intermediate node.

$$|\Omega| \leq \sum_{v=2}^{8} \sum_{e=v-1}^{\frac{v\cdot(v-1)}{2}} \binom{\frac{v\cdot(v-1)}{2}}{e} \cdot (v-2)^{|o|}. \tag{1}$$

Evaluating the expression in (1) for our case yields 268 143 512 722 241 or 2.68×10^{14}, roughly four orders of magnitude smaller than the DARTS search space [10] for combined normal and reduction cells, but nine orders of magnitude larger than the NAS-Bench-101 search space [23] for a single cell.

In order to go from a neural network architecture expressed as a computational DAG to an actual neural network, a ruleset is needed to compute things, such as the number of filter channels in a convolutional neural network or the number of hidden units in recurrent neural network. We used the following rules to determine the number of features a node outputs.

1) Input nodes are a no-op, and thus output the 16 features they are given.
2) Output nodes concatenate all inputs and sum them together to obtain the final probability of a pixel containing an active fire.
3) Intermediate nodes concatenate all inputs before running them through their respective operations.

Figs. 3 and 4 demonstrate how the number of features that each operation outputs is determined.

C. Performance Predictor Training

As mentioned at the start of Section III-B, NAS methods require a method to ascertain the performance of a given neural network on a given task, without training the neural network

to convergence. We opted to train a supervised learning model to predict this performance from architectural features. Reinforcement learning requires sampling many (on the order of 10^4–10^7, depending on the algorithm) transitions, and thus also a high number of evaluations of the reward function. From this, it follows that a reward function must be quick to evaluate, on the order of milliseconds per execution. Currently, the best way to meet this performance requirement is through the use of a small machine learning model to predict the performance of an architecture once trained. While alternatives exist, such as one-shot methods [10], [54] and zero-cost proxies [26], [27], many are still prohibitively expensive to calculate. Since we will be deploying the designed neural networks to an embedded device that requires INT8 quantization, we should also evaluate the postquantization performance of the model. More specifically, we predict the model's postquantization F1 score based on architectural features. We use the upper triangle of the adjacency matrix, combined with a one-hot encoded set of operations and the number of vertices and edges (as integers) as features for our predictor.

D. NAS Agent Training

Once a sufficiently accurate performance prediction model has been trained, we can start building a NAS agent. In this setting, the performance prediction model is used as the reward function for the reinforcement learning agent.

Since we are considering the case of on-board processing, we must also consider the computational resources required to evaluate the designed neural networks. A large and complex neural network might provide accurate predictions, but if we are unable to execute the network on-board the satellite because of its computational requirements, it is of limited use. There are a number of possible metrics that can be used for this. Some parameters can easily be computed from the neural network architecture itself and are quick to evaluate. This includes parameters, such as the number of FLOPs required to evaluate a neural network, the total number of trainable parameters, the total working set size [55], etc. Such parameters have the advantage that they are independent of the device the neural network gets deployed on, thus allowing the same NAS agent to be used to design neural networks for different hardware devices without invalidating the results.

It is also possible to instantiate the neural network, execute it on a device, and measure certain parameters, such as the inference latency or the energy consumption. Compared to using a basic parameter, such as FLOP count, measuring these metrics produces a much more accurate view of the actual impact of certain design decisions. The disadvantage of using a measured metric is that it must be measured, which can be a slow and cumbersome process. Updating the neural network used on a microcontroller usually requires rewriting some form of permanent storage, such as on-board flash memory or an SD card, which can be slow. Similar to the performance prediction model used earlier, our method of evaluating the resource requirements of a neural network must be executable in milliseconds to be able to gather the required number of samples to train an effective reinforcement learning agent. This makes the use of measured metrics infeasible in our setting.

Recently, there has also been research into predicting measured metrics, such as latency and energy consumption [26], [56]. Such prediction models provide a viable alternative to measuring these metrics in the context of the constraints imposed by our reinforcement learning setting.

However, in a reinforcement learning setting, the combined computational effort of predicting both computational requirements and classification performance is still unacceptably high. The computational burden of two separate regression models can be alleviated by designing a single regression model capable of predicting both classification performance and computational requirements. Such a model would still require the collection of a dataset of latency measurements, however, a slow and complex process for reasons outlined earlier. Thus, in this article, we use the total number of trainable parameters as a proxy for the resource consumption of the designed neural network on the target device.

So far, we have discussed the individual optimization objectives considered in this article, but we have yet to touch on the subject of how multiobjective reinforcement learning is applied to solve this optimization problem. Regardless of the exact setting, optimizing for multiple conflicting objectives simultaneously always involves the selection of tradeoffs between the advantages and disadvantages of different solutions to a problem. In the multiobjective reinforcement learning literature, one approach to resolving such tradeoffs is the use of a utility-based approach first proposed by Roijers et al. [57]. The utility-based approach requires the definition of a utility function that captures how a user derives utility from a reinforcement learning policy in a multiobjective setting. How this utility function is defined has a significant impact on how a multiobjective reinforcement learning problem should be approached.

Unfortunately, how utility is derived from a satellite-based active fire detection system is a complicated matter touching on many areas of expertise. Factors that can be considered include the following.

1) The cost of false positives compared to the cost of false negatives. This requires assessing the impact of dispatching firefighting resources to a location with no fire versus not dispatching fire fighting resources to a location that has fire.
2) The cost of a certain area being burned. Wildfires are costly because of the damage they cause, but creating numerical models for the damage caused by a wildfire is complex and requires expertise in many areas.
3) How quickly wildfires can be detected. Finding wildfires quickly is essential to controlling them, since it is much easier to control and extinguish a small wildfire compared to a larger fire.

Building a utility function to compute the utility derived from the use of a particular neural network for the detection of fires requires expertise across a number of areas and is beyond the area of expertise of the authors. With this in mind, we opt to use a linear combination that values task-performance, and the required computational resources equally. The use of a linear

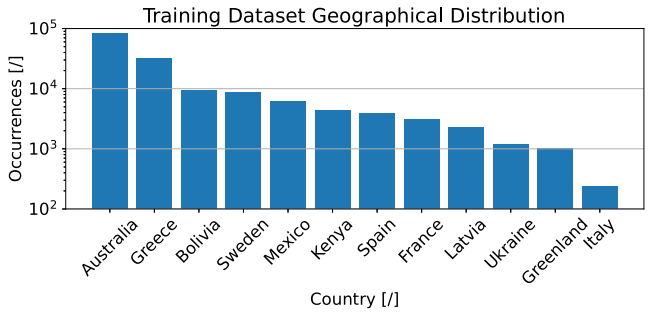


Fig. 5. Histogram of the geographic distribution of our dataset. Note the logarithmic *Y*-axis.

combination as a utility function also significantly simplifies the multiobjective reinforcement learning problem considered. For example, under the assumption of a linear utility function, the two different optimization criteria outlined by Rădulescu et al. [58] (scalarized expected return and expected scalarized return) become equivalent.

IV. Experiments

In this section, we outline the data collected and the different experiments performed in the context of this article. First, we analyze the distribution of samples in the dataset of multispectral satellite images in Section IV-A. Following this, we give a brief analysis of the performance of different neural networks on the target task in Section IV-B. Next, we analyze the process of training a supervised performance prediction model on the collected performance data in Section IV-C. We follow this with an analysis of the performance of our reinforcement learning agent on the NAS problem in Section IV-D. Finally, in Section IV-E, we select the best model found by our reinforcement learning agent and deploy it on a Google Coral Micro Dev Board. The model is also compared to several other machine learning models. Finally, we conduct some measurements regarding inference performance and power consumption to assess the viability of using our neural network-based approach in a nanosatellite context.

A. Multispectral Imagery Dataset

Using the process described in Fig. 2, we were able to create a dataset consisting of 20 images. In total, we find 110 448 positive label pixels and sample an equal amount of negative label pixels, leading to a total of 220 896 samples in the complete dataset. After splitting the dataset into a training, validation, and test set, the training set contains 154 636 samples, while the validation and test sets each contain 33 130 samples.

Fig. 5 shows the geographic distribution of our data points. This distribution is identical between the training, validation, and test sets. Our data are clearly not distributed uniformly in terms of geography, with more than half of all samples in the training set (82 938 or 53.63%) originating from Australia, while only 236 out of 154 636 samples (0.15%) originate in Italy.

B. Neural Network Performance Dataset

To gather performance data, we used a fixed computational budget of 14 days (two weeks) using 12 central processing units and a single NVIDIA A100 80 GB PCIe graphics processing unit (GPU), and trained as many neural networks as possible within this time period. When gathering neural network performance data, up to six neural network architectures are sampled from the search space and then trained concurrently using the same GPU.

Training was done using version 2.15 of the TensorFlow framework [59], given that the use of TensorFlow Lite would be required to deploy the models later on. Sometimes, quantization failed, with TensorFlow Lite reporting a violation of the same scale constraint. All inputs to a concatenation operation in TensorFlow Lite are required to use the same scale and zero point quantization parameters, and TensorFlow Lite's quantization systems failed to satisfy this constraint in a number of cases. Usually, this was the consequence of the computation of the different indices (NBR, NDVI, and AFD) being folded into the model. TensorFlow Lite failed to quantize the division operations used in these indices, which resulted in quantization and dequantization operations being inserted around them. The additional quantization and dequantization operations resulted in different quantization parameters being used when the different indices were concatenated with the original features they were calculated from. When this happened, it was impossible for us to ascertain the postquantization F1 score of the trained neural network; thus, this instance of the architecture was ignored. Other training runs with different random initializations for the same architecture were included in the dataset.

Neural networks were randomly sampled from the search space using the same algorithm that Cassimon et al. [15] used to sample initial states for their reinforcement learning environment. They were trained using the stochastic gradient descent (SGD) optimizer [60] with a learning rate of 1×10^{-2} and Nesterov momentum with a weight of 0.9. Gradients norms were clipped to 20, and we applied kurtosis regularization [61] to aid in quantization with a weight of 1×10^{-2} and a target of kurtosis of 1.8. The neural networks also had L2 regularization applied with a weight of 1×10^{-2}. The networks were trained with a batch size of 16 384 for 500 epochs. All convolution and pooling operations use "same" padding and operate with a stride of 1.

First, we analyze the performance we attained on the task of detecting active fires from single multispectral images. As mentioned in Section III-B, we were able to train 34 295 neural networks within our computational budget. In total, this dataset comprises of 11 547 unique neural network architectures, each trained using three different random initializations, with seeds ranging from 0 to 2 (both inclusive). Version of TensorFlow Lite 2.15 does not always succeed in quantizing a neural network following training. Fig. 6 shows a distribution of the post-quantization F1 score achieved on the validation set for all networks that we were able to successfully quantize. A solid black line indicates the median F1 score of 64.36%. We note that a large percentage of architectures ended up in the first and last buckets (buckets have a width of 1%), indicating many architectures either have an F1 score in the $[0\%, 1\%[$ or the $[99\%, 100\%]$ range.

In their paper, Cassimon et al. [15] noted the necessity of reward shaping when rewards are concentrated in a small region

TABLE I
OVERVIEW OF THE HYPERPARAMETERS USED TO TRAIN THE DIFFERENT PERFORMANCE PREDICTORS

Ridge Regression		**Least Absolute Shrinkage and Selection Operator (LASSO)**			
α	1	α	1×10^{-3}		
Multi-Layer Perceptron		**Radius Nearest Neighbour Regression**		**K Nearest Neighbour Regression**	
hidden_layer_sizes	$[48, 48]$	*radius*	16	*n_neighbours*	100
α	1×10^{-3}	*weights*	*distance*	*weights*	*uniform*
Gradient Boosted Trees		**SGD**		**Gaussian Processes**	
n_estimators	75	*max_iter*	5000	*kernel*	$32 \times \exp\left(-\frac{d(x_i,x_j)^2}{2 \cdot 16^2}\right)$
max_depth	4	η_0	1×10^{-5}	α	$1 + \sigma^2$
max_leaves	8	*learning_rate*	*constant*	*normalize_y*	*True*
Graph Neural Networks (GNN)		**Random Forest**		**Support Vector Machine (SVM)**	
hidden_layer_sizes	$[64, 64]$	*n_estimators*	100	*kernel*	*rbf*
activation	*LeakyReLU*	*max_depth*	15	C	5
η_0	1×10^{-2}	*min_samples_split*	50	γ	*auto*
η_γ	2.5×10^{-1}	*min_samples_leaf*	25	ϵ	1×10^{-2}

Ordinary Least Squares (OLS) was ommitted because we used the default hyperparameters.

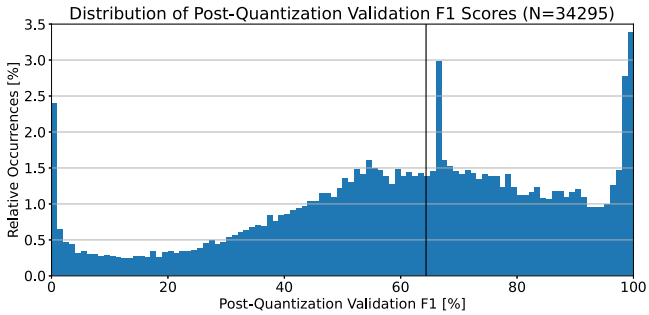


Fig. 6. Histogram showing the postquantization F1 score of all architectures in the performance prediction dataset, obtained on the validation set. The median is indicated with a solid black vertical line.

of their possible range. They used an exponential reward shaping function to correct for the fact that the majority of the architectures in the NAS-Bench-101 dataset have accuracies around 90%. In our use-case, reward shaping will not be necessary based on the data in Fig. 6.

Fig. 6 also shows that F1 scores in our use-case are not uniformly distributed. Given that bins in this histogram have a width of 1%, we would also expect the bars to have a height of 1% under the assumption of a uniform distribution. This is not the case, with architectures with an F1 score below roughly 40% being underrepresented, while architectures with an accuracy above 40% being overrepresented.

C. Performance Predictor

To find a good prediction model, we consider a wide variety of linear and nonlinear regression models. We use four different training strategies for our linear regression model: ordinary least squares (OLS), ridge, least absolute shrinkage and selection operator (LASSO), and SGD. We also consider eight nonlinear regression models with varying degrees of complexity: Gaussian processes, random forests, support vector machines (SVMs), K-nearest neighbor regression, radius neighbor regression, gradient boosted trees, multilayer perceptrons, and graph neural networks (GNNs). Finally, we also include two random sampling methods to contextualize our results: one method that samples uniformly, and one method that samples according to a normal distribution with the mean and standard deviation matching that of the training set.

All models except the GNN and both random samplers are taken from the scikit-learn Python library [62]. Models were trained using fivefold cross-validation with a held-out test set to be used later for evaluation. We consider four different evaluation metrics: the Pearson correlation coefficient, Kendall's τ, the coefficient of determination (R^2), and the root-mean-squared error (RMSE), obtained on the training and validation sets.

Table I lists the hyperparameters for the different performance predictors. Both random samplers have no hyperparameters. Parameters that are not specified use the default values in `scikit-learn` 1.4.0 [62]. All models that accept a seed were given the same seed. The seed was a randomly selected integer in $[0, 2^{32} - 1]$ selected from a `NumPy` [63] random number generator initialized with seed 0. Models that accept an "n_jobs" argument were given -1 for "n_jobs." All linear models were configured to fit scale and intercept.

Our GNN is based on the graph convolutions first introduced by Kipf and Welling [64]. The DAG of each architecture is given as input and fed through two graph convolutional network (GCN) layers with leaky ReLU [65] activations after each. After the final GCN layer, the features of each node are accumulated into one vector for the entire graph. The feature vector for the graph is finally fed through a linear layer resulting in a single scalar output. The GNN was trained using the Adam optimizer [66] minimizing the RMSE over 100 epochs. The learning rate started out at $\eta_0 = 0.01$, and was multiplied by $\eta_\gamma = 0.25$ after the fifth, tenth and 50th epoches.

Figs. 7–10 show the performance of these different prediction models. In each of these figures, the predicted F1 score is plotted against the true F1 score for each sample in the validation set for one of the folds used in k-fold cross-validation. A perfect predictor would have all data points on a diagonal line going from the bottom left to the top right, also shown on the figures

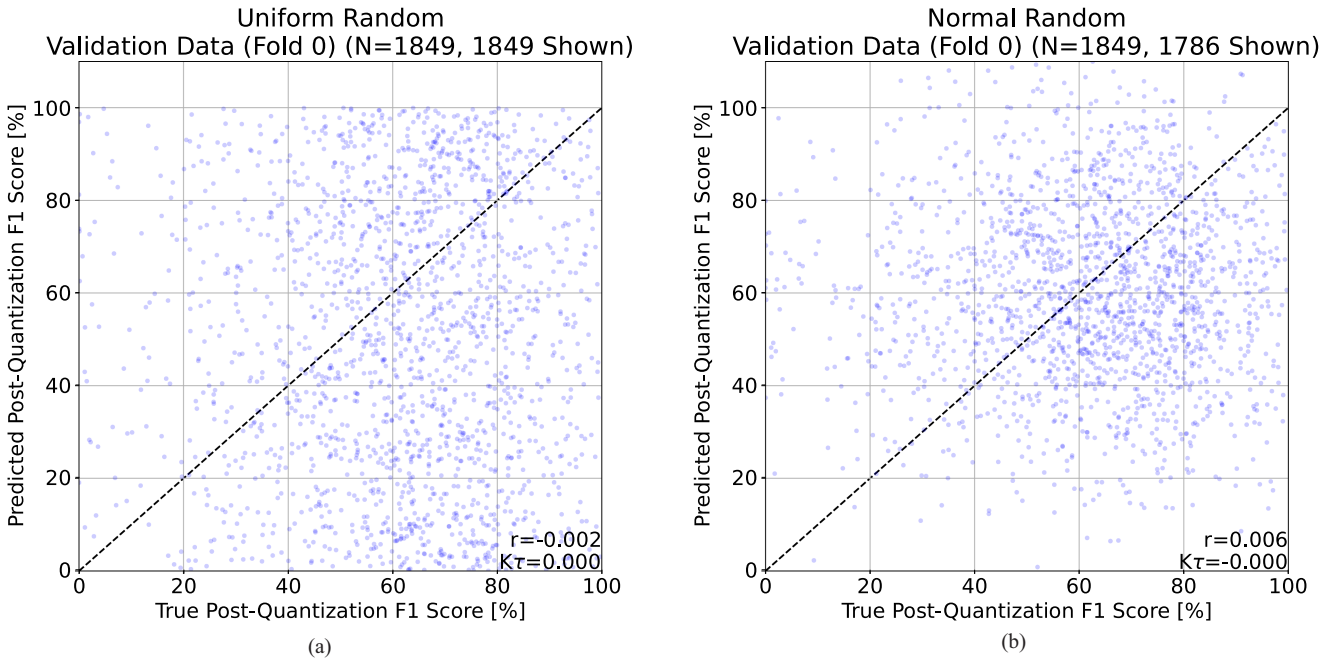


Fig. 7. Scatter plot for the predictions from the different random predictors. (a) Uniform model. (b) Normal model.

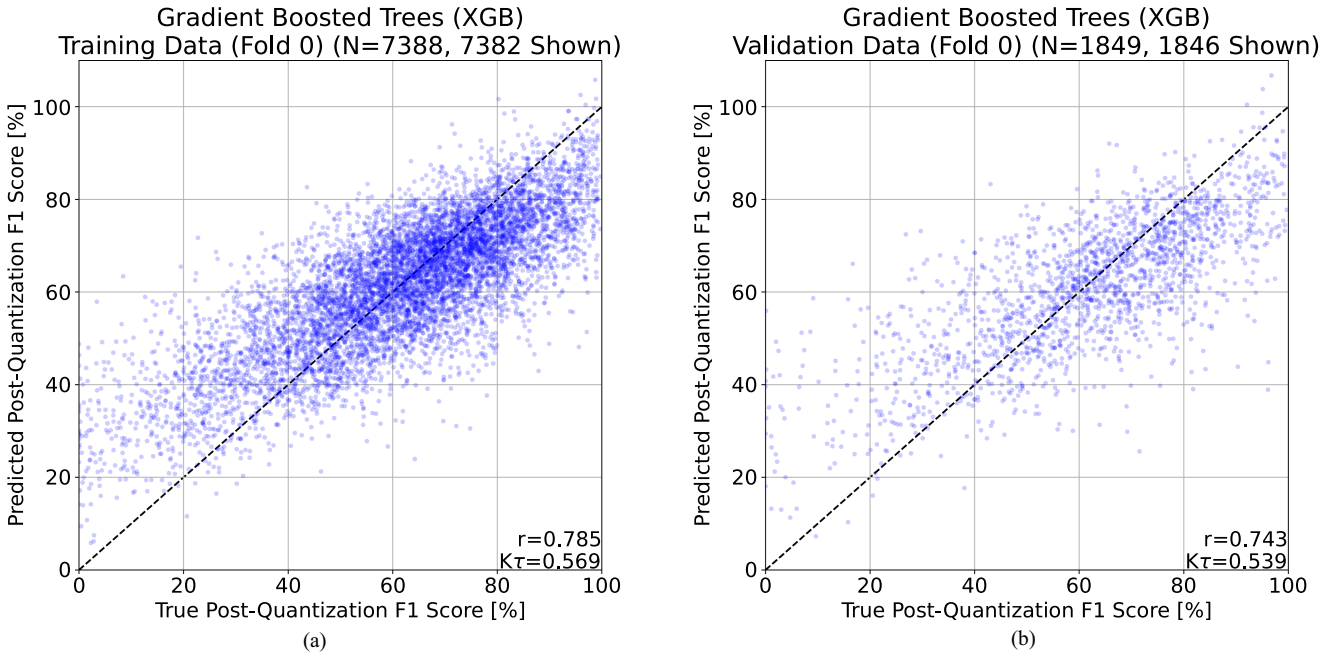


Fig. 8. Scatter plot for predictions from the gradient boosted trees model. (a) Training set. (b) Validation set.

as a dashed line. The more samples deviate from this line, and the greater the deviation, the less accurate a regression model is.

Table II gives the numerical results for performance predictors, sorted by RMSE on the validation set.

First, we consider predictions on the validation set by both random predictors, to provide a frame of reference when analyzing other predictors. Fig. 7(a) shows the scatter plot for a uniform random sampler. There is little remarkable about this figure, but one interesting thing to note is that the density of the scatter plot nicely reflects the distribution seen in Fig. 6. Fig. 7(b) shows the same scatter plot, but from our model based on a normal distribution. We note that in this case, the model generated several predictions outside the range shown on the graph. This is reflected in the title of the graph: Of the 1849 datapoints, only 1786 are within the range shown on the graph.

Next, we consider the results for the best performing model, gradient boosted trees. We show a scatter plot for the first fold of both the training set [see Fig. 8(a)] and the validation

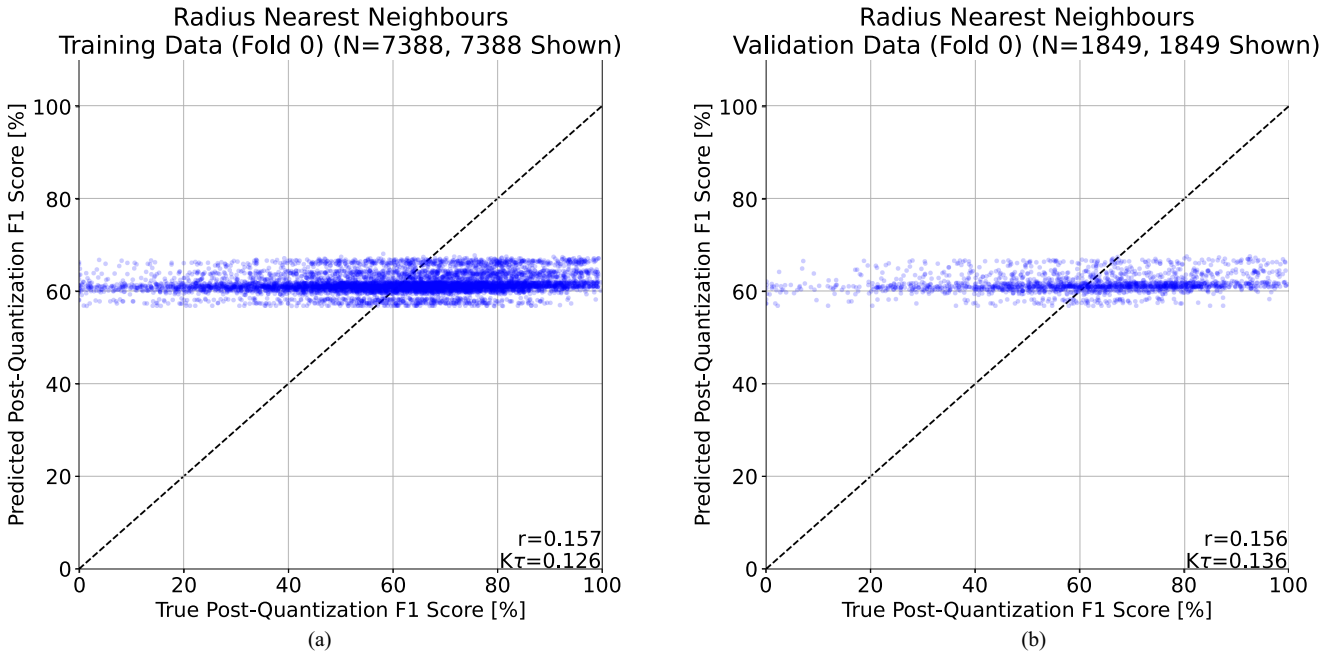
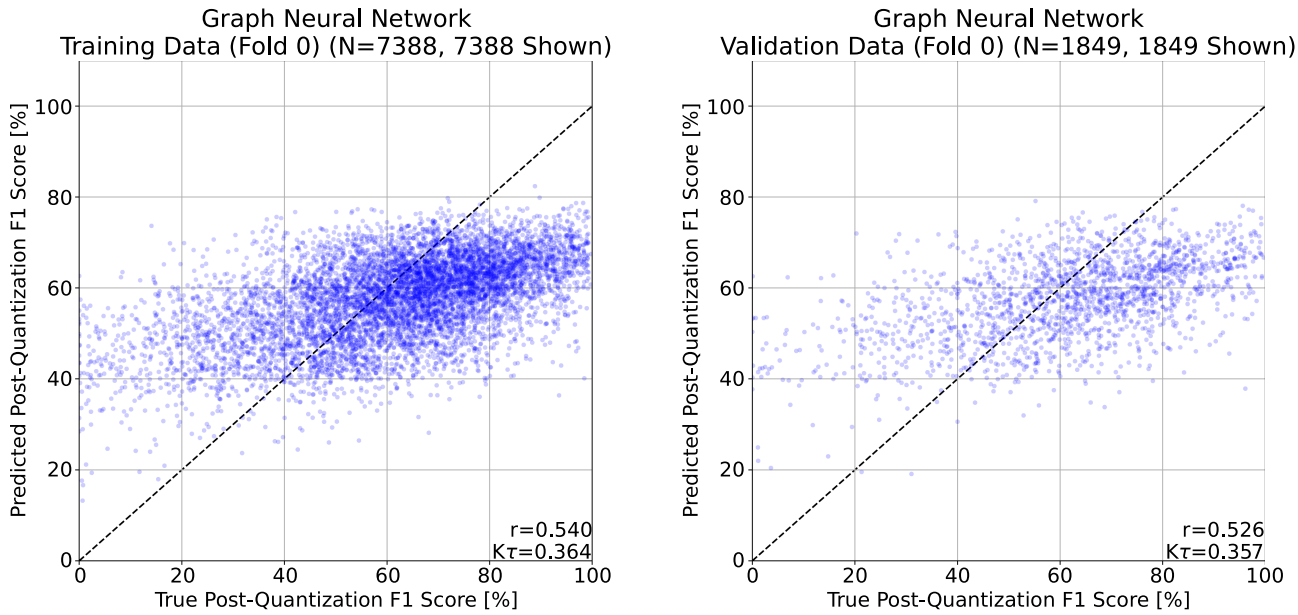


Fig. 9. Scatter plot for the predictions from the radius nearest neighbors model. (a) Training set. (b) Validation set.



(a) Training set. (b) Validation set.

Fig. 10. Scatter plot for the predictions from the GNN. (a) Training set. (b) Validation set.

set [see Fig. 8(b)], and we note that other folds show similar results. The predictions capture the overall trend, but are far from precise. The predictions still display a significant error that is often between -20% and $+20\%$. This is also reflected in Table II, which gives an RMSE on the validation set of 13.7% for gradient boosted trees. Interesting to note on Fig. 8(a) in particular is the difference in prediction accuracy between samples with a ground-truth F1 score below 40% and samples with a ground-truth F1 score above 40%. This is likely a consequence of the observation we made in Section IV-B about Fig. 6 that architectures with an F1 score below 40% are underrepresented in our dataset.

From Table II, we see that most predictors managed to produce a reasonably good model. The best result in each column is highlighted in bold. Excluding the random models, the radius nearest neighbor predictor is the only model that has a Pearson's R below 50%. We considered several distance metrics and radii and tested both uniform and distance weights, but were unable

TABLE II
COMPARISON OF THE DIFFERENT PERFORMANCE PREDICTION MODELS

Algorithm	Pearson's R	Kendall's τ	R^2	RMSE	RMSE (Training Set)
Gradient Boosted Trees	$\mathbf{74.2\% \pm 0.58\%}$	$\mathbf{53.1\% \pm 0.80\%}$	$\mathbf{54.9\% \pm 0.74\%}$	$\mathbf{13.7\% \pm 0.17\%}$	$12.7\% \pm 0.02\%$
Support Vector Machine	$73.9\% \pm 0.41\%$	$53.1\% \pm 0.54\%$	$54.4\% \pm 0.68\%$	$13.8\% \pm 0.18\%$	$\mathbf{11.3\% \pm 0.07\%}$
Gaussian Process	$73.2\% \pm 0.28\%$	$52.3\% \pm 0.69\%$	$53.5\% \pm 0.40\%$	$13.9\% \pm 0.13\%$	$13.3\% \pm 0.03\%$
Ridge Regression	$70.4\% \pm 0.28\%$	$49.9\% \pm 0.70\%$	$49.5\% \pm 0.44\%$	$14.5\% \pm 0.15\%$	$14.4\% \pm 0.04\%$
Ordinary Least Squares	$56.8\% \pm 27.25\%$	$49.9\% \pm 0.70\%$	$49.5\% \pm 0.47\%^*$	$14.6\% \pm 0.08\%^*$	$14.4\% \pm 0.04\%$
SGD Regression	$70.0\% \pm 0.23\%$	$49.5\% \pm 0.61\%$	$48.5\% \pm 0.26\%$	$14.7\% \pm 0.16\%$	$14.5\% \pm 0.03\%$
LASSO Regression	$69.7\% \pm 0.37\%$	$49.2\% \pm 0.76\%$	$47.9\% \pm 0.44\%$	$14.7\% \pm 0.12\%$	$14.6\% \pm 0.04\%$
Multi-Layer Perceptron	$70.7\% \pm 1.22\%$	$50.0\% \pm 1.20\%$	$47.6\% \pm 2.06\%$	$14.8\% \pm 0.27\%$	$11.4\% \pm 0.37\%$
Random Forest	$67.5\% \pm 0.80\%$	$47.5\% \pm 0.76\%$	$45.4\% \pm 1.03\%$	$15.1\% \pm 0.11\%$	$14.4\% \pm 0.02\%$
K Nearest Neighbours	$61.3\% \pm 1.15\%$	$42.0\% \pm 0.77\%$	$28.3\% \pm 0.72\%$	$17.3\% \pm 0.20\%$	$17.0\% \pm 0.04\%$
Graph Neural Network	$51.9\% \pm 2.34\%$	$35.6\% \pm 1.60\%$	$23.7\% \pm 1.39\%$	$17.8\% \pm 0.30\%$	$17.8\% \pm 0.18\%$
Radius Nearest Neighbours	$15.4\% \pm 1.81\%$	$12.6\% \pm 1.66\%$	$2.0\% \pm 0.32\%$	$20.2\% \pm 0.22\%$	$20.2\% \pm 0.05\%$
Normal Random	$0.0\% \pm 1.94\%$	$-0.5\% \pm 1.43\%$	$-102.6\% \pm 5.17\%$	$29.1\% \pm 0.30\%$	$29.0\% \pm 0.15\%$
Uniform Random	$0.7\% \pm 2.37\%$	$0.4\% \pm 1.38\%$	$-237.1\% \pm 6.35\%$	$37.5\% \pm 0.46\%$	$37.0\% \pm 0.12\%$

Each column is given as a mean and standard deviation over 5 folds. The algorithm with the best mean performance in each column has been marked in bold. Numbers marked with an asterisk excluded some outliers caused by poor convergence of the algorithm. All statistics were computed using the validation set, unless stated otherwise.

to find a configuration that yielded a good fit. When using distance weights, the model overfit very easily, while when using uniform weights, the model usually underfit. Fig. 9(a) and (b) shows predictions made by the radius nearest neighbor model on the training and validation datasets. Using this configuration, it is clear that the model underfit the training dataset. Instead, the predictions seem to vary (narrowly) around the mean post-quantization F1 score of 64% we show in Fig. 6.

Finally, we also note the behavior of the GNN. Despite being intrinsically suited for dealing with graph data, the GNN showed relatively weak performance. The GNN seems to exhibit a much more extreme version of the bias displayed by the gradient boosted trees in Fig. 8(a) and (b). Fig. 10(a) and (b) shows the predictions for the GNN model on the training [see Fig. 10(a)] and validation [see Fig. 10(b)] sets. The model rarely predicts F1 scores below 40%, reflecting the bias in the underlying distribution shown in Fig. 6.

D. NAS Agent

In this section, we cover the details of the reinforcement learning agent that was used to design the final classification network for active fire detection.

Following the experiments in Section IV-C, we selected the gradient boosted tree regressor to be used as part of the reward function for our reinforcement learning, given its strong performance in predicting the postquantization F1 score of the trained neural networks. As mentioned in Section III-D, the reward function we used is a combination of the postquantization F1 score predicted by the gradient boosted tree model and the total number of trainable parameters in the designed neural network. The total number of trainable parameters is first normalized between 0 and 1 before the linear combination is applied, to ensure both rewards have a similar magnitude. While it is relatively straightforward to define a lower bound for the total number of trainable parameters, finding an upper bound is harder. Since both bounds are necessary to compute the normalized total parameter count, we make an assumption on the worst case. We assume that an architecture with the maximum number of edges and vertices, where every node is assigned the "linear-prelu" label is the worst case in terms of the total parameter count. This assumption is justified by the fact that for the "linear-prelu" operation not only the linear layer but also the activation function has trainable parameters. Finally, both components of the reward function are assigned an equal weight of 0.5, and combined using a linear combination.

We use the reinforcement learning agent introduced in [15] to design neural networks in an iterative fashion. The agent is given an architecture, along with a set of architectures obtained by modifying the first architecture as input. The agent then outputs the index of the preferred architecture. If the agent selects the current architecture, the episode terminates. If the agent selects a modified architecture, this architecture becomes the new current architecture and the process repeats.

The architecture of the reinforcement learning agent is shown in Fig. 11. Architectures are first prepared by taking the lower triangular half of the adjacency matrix, flattening it, and concatenating it with a one-hot encoding of the operation on each node of the architecture. Following preparation, architectures are projected into a latent space using several linear layers with ReLU activations, before having their latent representations fed to a transformer encoder as a sequence. The first output of the transformer encoder is then used to compute a state-value and an advantage value for each action, as is standard practice when using dueling heads [67]. During training, episodes have a length of up to 16 time steps, with episodes taking up to 32 time steps during evaluation. Each agent is trained five times, with numbers from 0 to 4 (both inclusive) being used as seeds for random number generation. Training is terminated once the agent has been trained on 10×10^6 time steps of experience. We use $\gamma = 0.9$ following the ablation study in [15]. Agents are trained using the Ape-X algorithm [68], a variant of deep Q learning. Agents are shown up to 50 neighbors each time step. We use the Adam optimizer [66] with a learning rate of 5×10^{-5}. We use double Q-learning and dueling heads with target networks that are updated every time the online network is trained for 8192 samples. Exploration is handled using a per-worker epsilon-greedy strategy. Every worker uses an epsilon-greedy exploration strategy with a different value of epsilon. For the precise calculation of epsilon values, we refer to the original Ape-X paper [68]. Following Cassimon et al. [15],

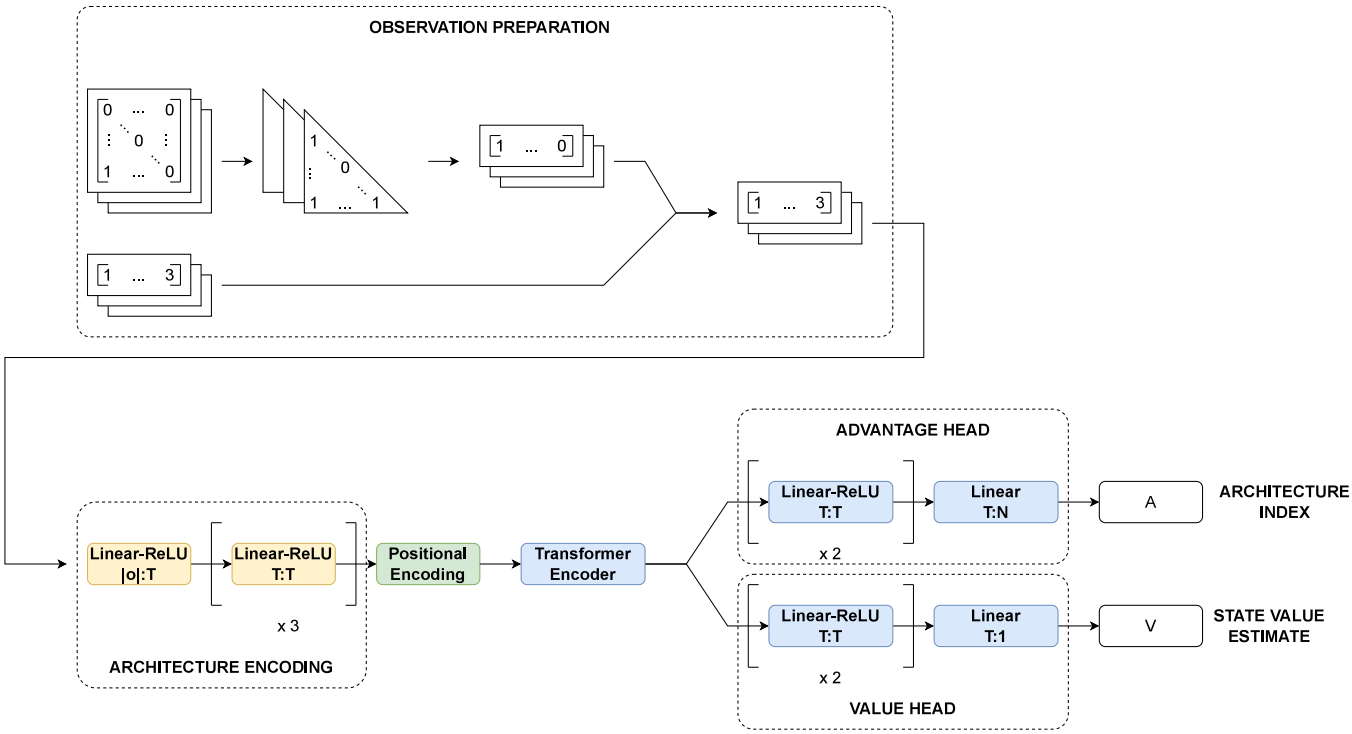


Fig. 11. Architecture of the reinforcement learning agent. Figure taken from [15] with permission.

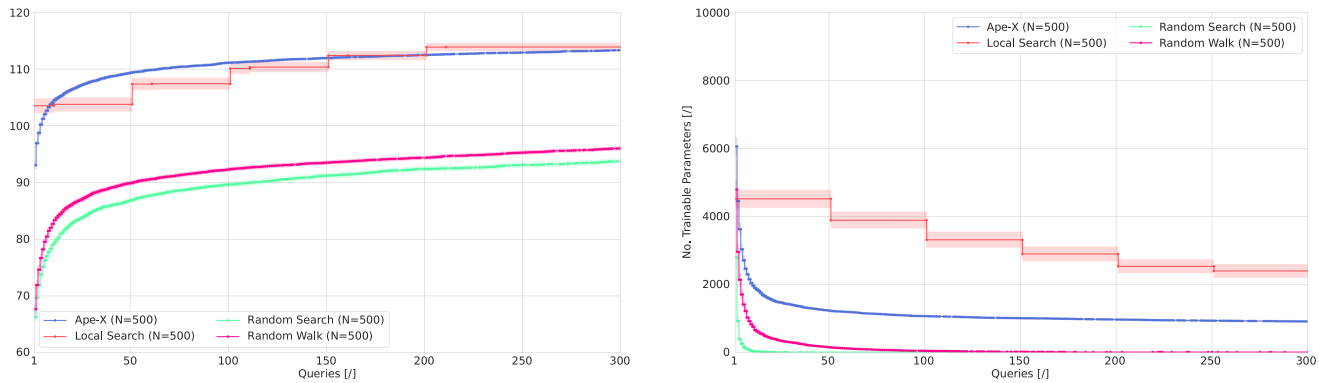


Fig. 12. Performance of the best architecture found so far for each objective as a function of the number of queries made. (a) Highest predicted F1 score as a function of the number of queries made. (b) Lowest number of trainable parameters as a function of the number of queries made.

we use a replay buffer with 2.5×10^4 entries, using prioritized replay with $\alpha = 0.6$ and $\beta = 0.4$. Following the results from Section IV-B and contrary to [15], we do not use any reward shaping. When evaluating, we randomly sample five sets of 1×10^4 architectures from the search space and evaluate the agents on each of these sets. We also include random search, random walks, and local search as additional baselines.

Fig. 12(a) and (b) shows the performance of all agents on both objectives as a function of the number of queries. One thing that immediately stands out in Fig. 12(b) is how easy it is for the random search agent to design neural networks with very low parameter counts. While this may seem surprising at first, it follows from the sampling algorithm used for random search. We use the same sampling strategy detailed in [15]. As shown in [15, Fig. 5], the random search algorithm samples uniformly in function of vertex counts. Since we consider architectures with 2–8 vertices (both inclusive), random search has a roughly 1/6 chance ($16.66\ldots\%$) to sample an architecture with only two vertices. An architecture with only two vertices has exactly one edge and no trainable parameters, apart from the linear projection used in the output. Thus, such an architecture has 17 trainable parameters and occurs with a $16.66\ldots\%$ probability at the end of an episode. This also explains why random search struggles much more to find architectures with high postquantization F1 scores, given that these are much rarer.

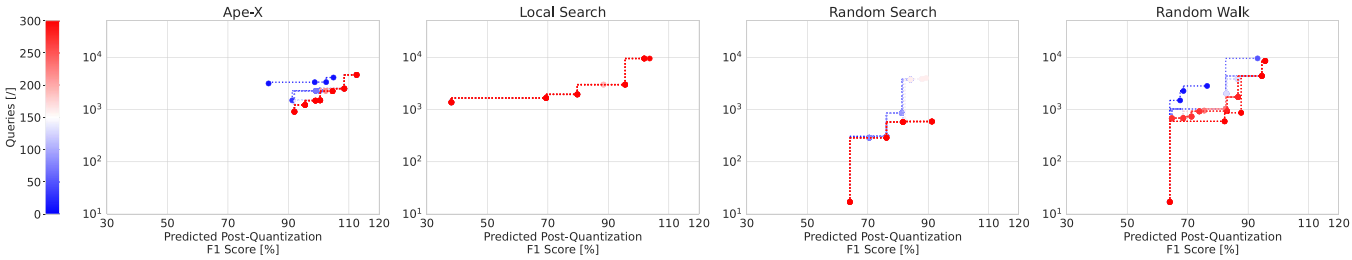


Fig. 13. Pareto fronts generated by each agent given a query budget of 300 queries. Every ten queries, it is redrawn. The first Pareto front is drawn in blue, the last in red, and others interpolate between blue and red based on how many queries were required to find the front. Note the logarithmic *Y*-axis.

Fig. 12(a) also shows an interesting artifact. Both local search and our reinforcement learning agent are able to find architectures with postquantization F1 scores above 100% fairly easily. This indicates that both algorithms have managed to uncover adversarial inputs to the performance prediction model, which lead it to predict an impossible F1 score. We saw a limited number of samples that were predicted to have F1 scores above 100% in Fig. 8(b), but they are much more frequent in the evaluation data. Specifically, such adversarial samples are most common with local search, occurring in 246 755 traces (98.70%), followed by our reinforcement learning agent with 39 925 (15.97%) occurrences. They occur only rarely with either random algorithm, with random search reporting 95 occurrences (0.04%) and random walks reporting 136 (0.05%) occurrences. We hypothesize that these adversarial samples likely adversely impacted the performance of the reinforcement learning agent to a limited degree. A possible mitigation strategy is to simply clamp the predictions from the performance prediction model between 0 and 1 (or 100%). The impact of such a mitigation measure may be limited, however, since it does not address the underlying cause of the adversarial samples (an imperfect regression model), rather it only addresses the symptom (overprediction of post-quantization F1 scores). Building stronger performance prediction models is likely the best mitigation strategy against such issues.

Fig. 13 shows the Pareto fronts found by each agent under a query budget of 300 queries. These Pareto fronts confirm what was already visible in Fig. 12(a) and (b), which random algorithms do not struggle with finding neural networks with low parameter counts, but they do struggle with finding neural networks with high postquantization F1 scores. We count queries in the same way as [15], which severely limits the number of opportunities local search has for making improvements to the architecture. In this search space, most architectures have 50 neighbors, allowing local search six time steps to make improvements before its query budget is exhausted. Comparatively, other algorithms can play up to 300 episodes (up to 4800 time steps), since they only need to query an architecture's accuracy at the end of the episode. This likely also explains why local search did comparatively worse than Ape-X, with Ape-X finding better architectures both in terms of the total number of trainable parameters and the predicted postquantization F1 score. Ape-X's Pareto front is also relatively compact compared to those of the random algorithms, which is likely explained by the fact that Ape-X (and local search) try to honor the preference vector we selected ($[0.5, 0.5]$), whereas random search and random walks simply sample randomly, without regard for the preference vector used.

E. Deployment

Finally, to fully evaluate the performance of the architecture returned by our reinforcement learning agent, we deploy the best performing architecture on a Google Coral Micro Dev Board [17], a low-power deep learning accelerator with support for the widely used TensorFlow Lite deep learning framework [59]. The architecture selection process happens through the use of the linear utility function from Section III-D. For each evaluation episode, we take the final architecture, compute the value of the utility function, and select the architecture with the highest utility. This architecture is then trained 100 times, each with a different random initialization, and the trained network with the highest post-quantization F1 score is selected for deployment. The architecture obtained a median post-quantization F1 score of 99.884% on the validation set. The lowest post-quantization F1 score obtained on the validation set was 98.917%.

The best neural network architecture is showcased in Fig. 15. This architecture has several interesting features. First of all, we note that the architecture only contains linear layers with various activation functions. Despite the addition of other operations, such as convolutions, max pooling, and spectral attention, the reinforcement learning agent selected an architecture that consists solely of linear layers. The next element of note is the two-stream design. The agent has essentially designed a network with two parallel information streams of equal depth that are combined at the output node. Because of this design, no intermediate node has more than one input, and thus, every intermediate node only has 16 input and 16 output features. This reduces the overall parameter count of the neural network, while still retaining depth, allowing for complex decision boundaries with a limited number of trainable parameters. We also note that the agent is aggregating features at multiple levels. There are branches that connect the input to the output with a path containing 1–3 intermediate nodes. This is reminiscent of the structure of U-Nets [48], or auxiliary towers [69] sometimes employed to improve the performance of convolutional neural networks.

The firmware makes use of the `coralmicro` support library to access the tensor processing unit (TPU). Because TensorFlow

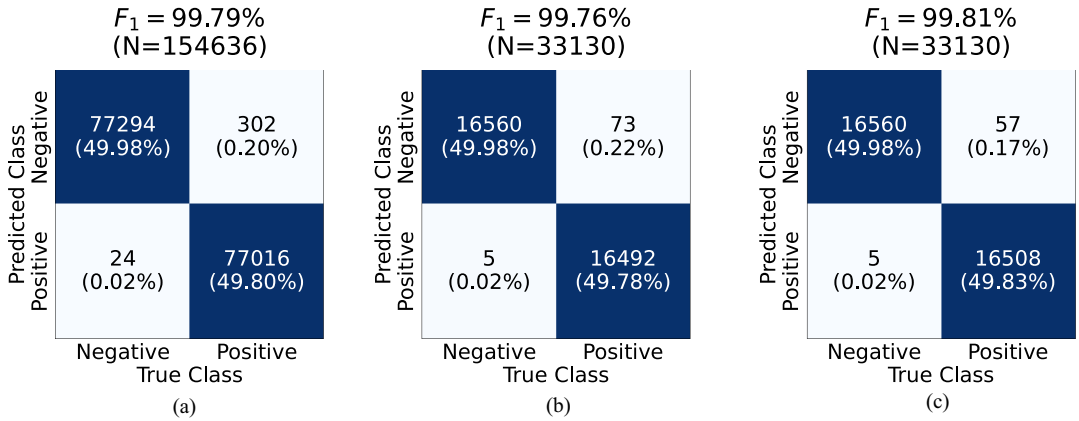


Fig. 14. Confusion matrices for the neural network architecture in Fig. 15. (a) Training set. (b) Validation set. (c) Test set.

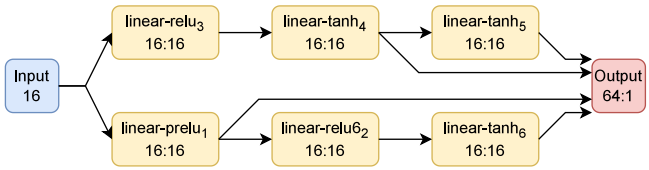


Fig. 15. Best neural network found by our reinforcement learning agent. The architecture has 1716 trainable parameters and obtains a median postquantization F1 score of 99.884%. Each node shows the number of input features and output features, separated by a colon.

Lite does not support dynamic batch sizes, images are inferenced in batches of one. Data are fed to the Coral Micro Dev Board through an Ethernet-over-USB connection. The firmware operates on a request–response paradigm and supports two commands. The first command is a simple "PING" to verify that the network connection between the device and the laptop is operational. The second command is an inference request, with a single data sample attached. Upon reception of the inference request, the device performs inference and returns the classification result as an INT8 value representing the probability of the pixel containing fire.

The version of TensorFlow Lite bundled with the Google Coral Dev Board Micro does not support division operations. Our preprocessing code does require division operations, however, since NDVI, NBR, and AFD are all instances of the generalized normalized difference index [51], which requires a division. Thus, we opted to execute our preprocessing using the floating point unit (FPU) present in the ARM M7 core of the NXP i.MX RT1176 microcontroller. The data are sent to the device in FP32 format, preprocessing is done and executed by the MCU, the data are quantized, and inference of the neural network is executed on the TPU.

Fig. 14(a) shows the classification results for the neural network in Fig. 15 for the training set, with results for the validation and test sets shown in Fig. 14(b) and (c). From these confusion matrices, we can conclude that the neural network is a very strong classifier, achieving F1 scores in excess of 99% on the training, validation, and test sets. We can also see that the network tends to produce false negatives more than it does false positives, with false negatives about $10\times$ more prevalent than false positives.

When considering only the misclassifications, we see that on each dataset, the neural network classified at least two events perfectly (two on the training set, eight on the test set, and ten on the validation set). The absolute highest number of misclassifications was made on the "Australia_1" event for the training set, with 168 samples from this set being misclassified (0.2% of the total samples for "Australia_1" in the training set). In relative terms, the "Greece_4" event in the validation set yielded the worst result, with 2.6% of samples being misclassified (four out of 154 samples). Both complete images are shown in Fig. 16. We assess if there is a significant difference in the ranking of the different events between different datasets by computing the Kendall's τ correlation between the ranking of events for different sets. The Kendall's τ correlation between the training and validation set, as well as between the training and testing set is very low, at -1.05% and -7.37%, respectively. Between the validation and testing sets, we do find a moderate correlation, at 43.16%. This implies that when our network did badly on an event in the validation set, it also did badly on the same event in the test set, which is to be expected under the assumption that there is no overfitting.

Fig. 16 shows some example images from the THRawS dataset, along with the predictions generated with our neural network. False color images were created by using Bands 4, 3, and 2 for the red, green, and blue channels of the image, respectively. "Australia_1" and "Greece_4" were included since these are the images that our network performed the worst on when considering the subset of pixels used for training. From the images and the confusion matrices, we can see that the network tends to predict false positives more often than false negatives, contrary to what we see on the dataset used for training in Fig. 14. This is to be expected, since the training set was sampled to be balanced, while the complete images are extremely unbalanced, with $>99\%$ of all samples belonging to the negative class. We also see that despite heavy cloud cover, performance in the "Sweden_0" image is not severely degraded, showing that our network is robust in the face of cloudy weather.

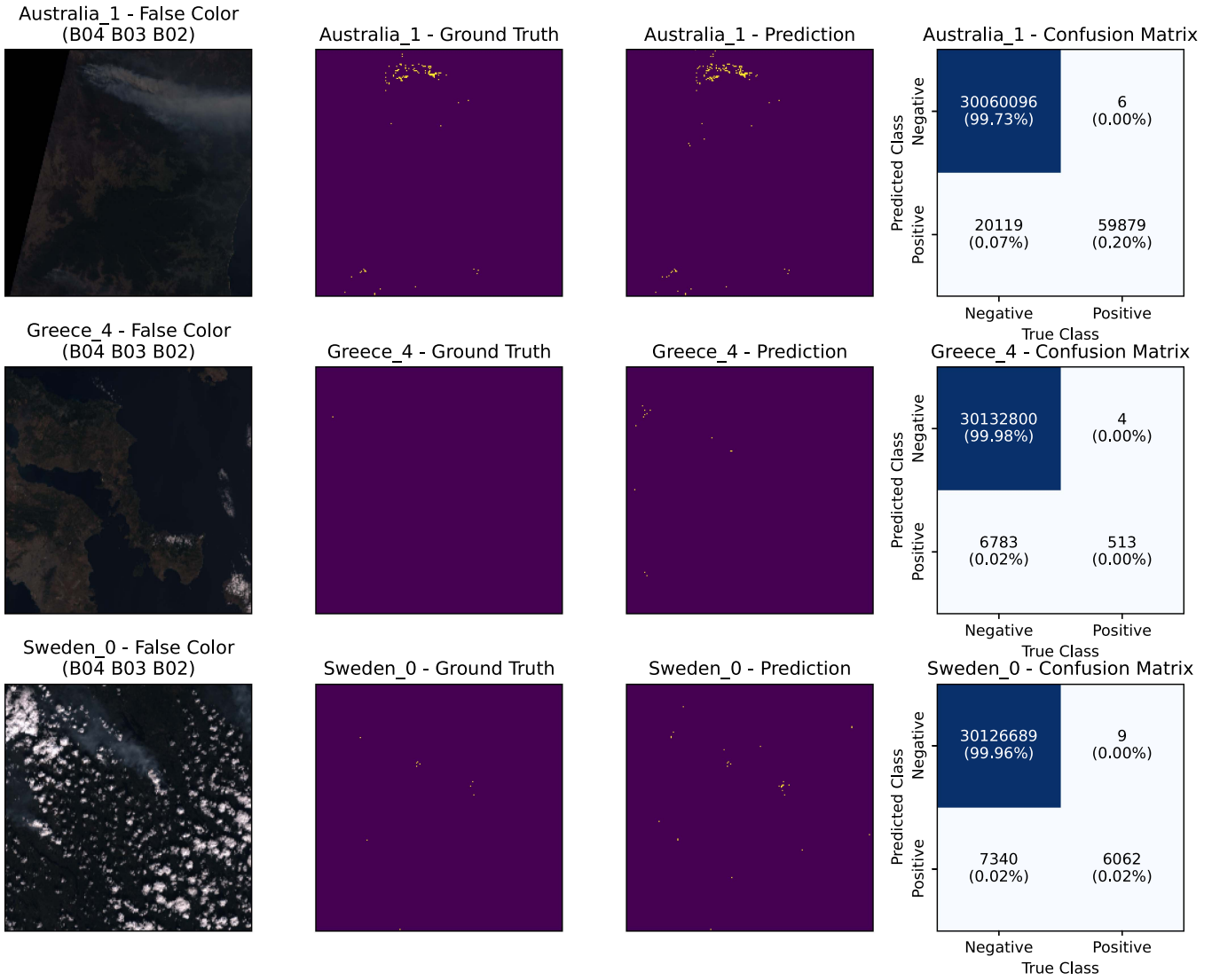


Fig. 16. Three example satellite images with predictions. The black area in the top-left corner of the "Australia_1" image contains no valid data, and such areas were not considered in the creation of the dataset. The confusion matrices concern the entire image, not just the pixels that were used in the dataset.

While performing inference on the training set, our firmware averages a round-trip latency of 0.984 ms per sample, corresponding to a throughput of 1016.01 samples per second. This includes the time it takes to send data from the laptop to the device, and the time it takes to receive the data again.

To analyze the feasibility of deploying this system on a nanosatellite we use a DC energy analyzer while the device is operating. Our energy analyzer is a JS110 JouleScope [70]. This should give us insight into the power consumption of the device, both in terms of overall consumed power, as well as peak power consumption. Fig. 17 shows an energy trace while our firmware is running on the Google Coral Micro Dev Board [17]. Fig. 17(a) shows idle power consumption, while the device is waiting for a command, while Fig. 17(b) shows the power consumption while the device is performing inference. The graph shows that while idle, the device consumes approximately 640 mW, which jumps to 780 mW average while performing inference. The idle power draw for the device is relatively high, which we attribute to several factors. First, the firmware we use has not been optimized for reducing power consumption. Such optimizations include the heavy use of sleep states and disabling the TPU when it is not in use. We also note that the communication system used to transfer data to the device (Ethernet-over-USB) has a relatively high power draw, with even dedicated application-specific integrated circuits drawing in the dozens of milliwatts [71]. Despite this seemingly high power consumption, the Dev Board Micro's datasheet [17] references average power peaks of up to 3 W, which is significantly higher than what we observe. Our power consumption being significantly lower is likely explained by the fact that the neural networks we execute are significantly simpler than the complex CNNs used for computer vision tasks.

We also compare our neural network against three other machine learning models in Fig. 18: a linear (ridge) classifier, a decision tree, and an SVM. All three models were taken from the scikit-learn Python library. The linear classifier fitted both intercept and slope, with α set to 1. For the decision tree, the

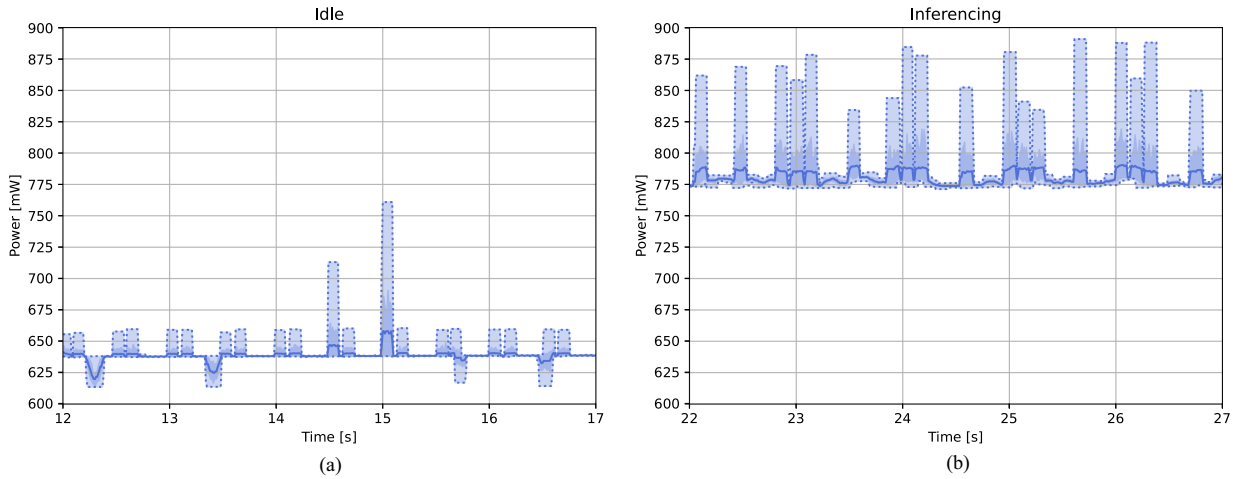


Fig. 17. Waveform trace of the power consumption of the Google Coral Micro Dev Board while our firmware is running, sampled at 100 Hz. Statistics are computed over a 0.1-s window. The mean is given as a solid blue line, and the minimum and maximum as a dotted blue line. The area between the minimum and maximum, and the area spanning a 95% confidence interval around the mean are filled in. (a) Device is idle, waiting for a command. (b) Device is servicing repeated inference requests.

maximum depth was set to 5. The SVM used a linear kernel with γ set to auto, and "C" set to 10. For all three models, the seed used for random number generation was set to 0. All other settings were left on their default values. From this comparison, we can see that all models perform very well on the dataset, with every model attaining an F1 score of over 99% on each of the datasets. Compared to our model, each of the other models obtains a higher number of false positives on each of the datasets, while obtaining a lower number of false negatives. This results in our neural network outperforming the linear classifier in terms of F1 score, but underperforming compared to the other models. The most likely explanation for the strong performance of such simple models is the algorithm that was used to obtain labeled data. As we mentioned in Section III-A, we used the algorithm from [40] to label the satellite images from the THRawS dataset. As mentioned in Section II-C, this is a relatively basic algorithm, built on the comparison of reflectance ratios against thresholds. This results in a straight-forward relationship between features and labels, making the classification problem easier.

V. Discussion

This work demonstrated how the NAS agent proposed in [15] can be used to design active fire detection systems. While this proved an interesting use-case, one possible point of improvement is the data that were used. We opted to use Massimetti et al.'s [40] algorithm to label Sentinel-2 satellite images. While this allowed us to easily gather a sufficient amount of labeled images, the quality of these labels is limited. Given the simplicity of the labeling algorithm, most machine learning algorithms can easily approximate the labeling algorithm, thus resulting in seemingly strong performance from the classifier, as we saw with all machine learning models we evaluated achieving F1 scores in excess of 99%. To fully evaluate the capability of the developed active fire detection system, an analysis should be made that considers a different source of labeled data that does not suffer from these issues, such as [72].

Section IV-C focused on the training of performance prediction models. Here, we noted that despite their intrinsic strengths at dealing with graph data, graph neural networks (GNNs) performed relatively poorly at the performance prediction task, only achieving a Kendall's Tau ranking correlation of around 24%, compared to gradient boosted trees' 53%. The GNN model we used is a relatively simple model based on GCNs [64], and more complex models, such as graph attention transformers [73], will likely perform better. GNNs are still an active area of research, and we expect that there are many improvements that could be made to our GNN architecture that would result in better predictions.

Section IV-C also showed that while many performance predictors were able to achieve satisfactory performance, there is still room for improvement. This is particularly evident when comparing against other performance prediction models in the NAS literature. For example, the three models used in the NAS-Bench-301 benchmark [24] (GIN, XGB, and LGB) all obtained R^2 scores of over 0.8 on their dataset. Compare this to our best model, a gradient boosted tree, which achieved an R^2 score of approximately 0.55. The fact that we are trying to predict postquantization performance metrics likely resulted in a more complex regression problem, at least partially explaining the performance gap. We expect that the performance of our predictors could be improved through the use of better feature engineering. In their paper, Kadlecová et al. [26] showed that the computation of a set of fairly simple features of the underlying architectures can result in a strong feature set to train performance predictors on. We hypothesize that using such an improved feature set would likely also lead to stronger predictive performance in our case.

The reinforcement learning agent originally proposed by Cassimon et al. [15] and used in this work proved effective, as shown in Section IV-D. While random search and random walks provided a very strong baseline when it comes to reducing the total number of trainable parameters, their ability to design neural networks that can perform accurate

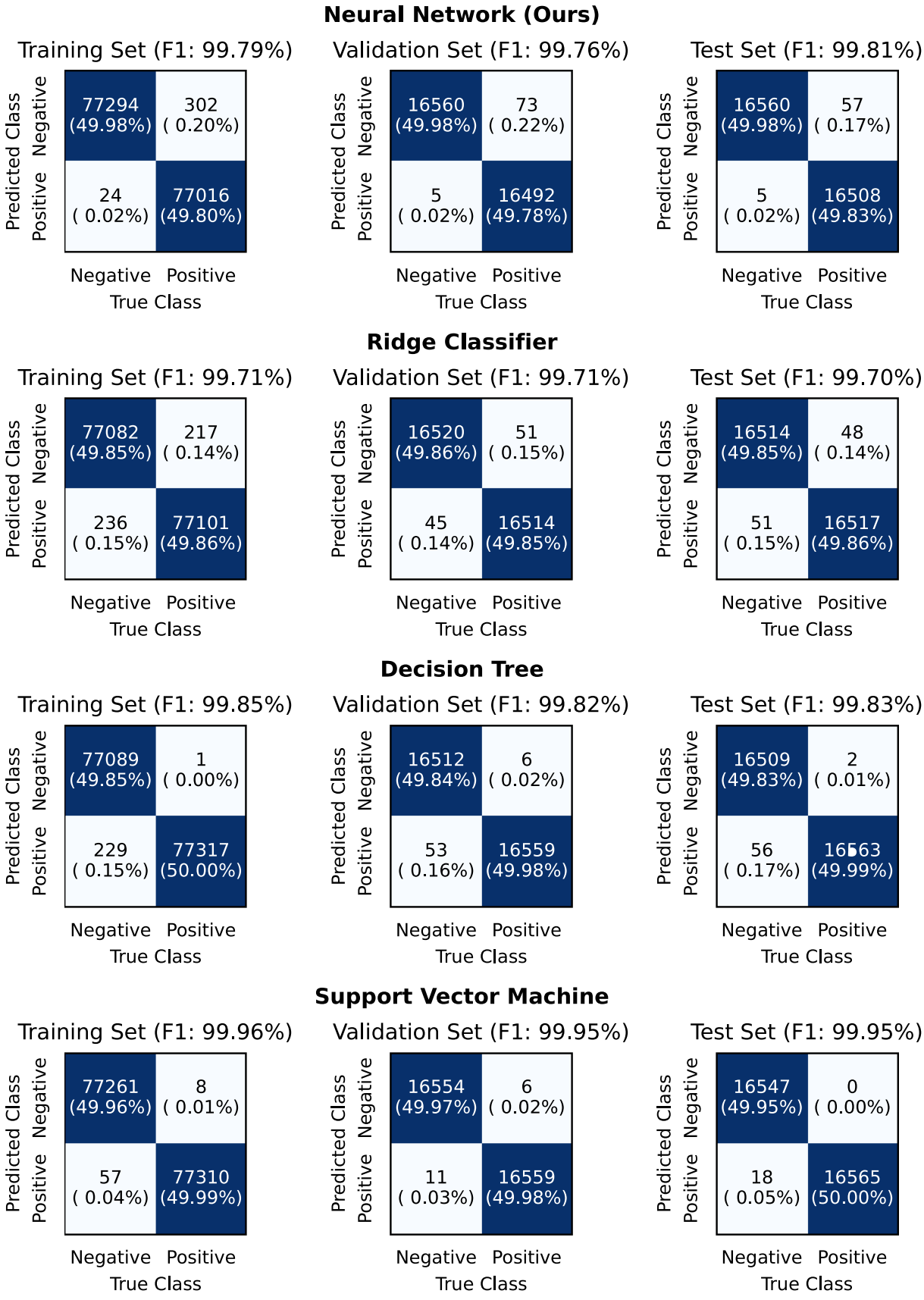


Fig. 18. Comparison of our neural network (top) against three other machine learning models.

classification is limited with local search and reinforcement learning outperforming them across all query budgets. The use of both local search and reinforcement learning also revealed some weaknesses in our performance prediction models, in the form of the discovery of adversarial samples in, respectively, 99% and 16% of traces. Despite this, the final neural network architecture designed by the reinforcement learning agent demonstrated strong performance on the task at hand and did so within a limited computational budget. Our results also highlight the robustness of the reinforcement learning agent in the face of an inaccurate performance prediction model and a diffuse distribution of post-quantization F1 scores.

In Section IV-E, we deployed the best neural network designed by our reinforcement learning agent onto a Google Coral Dev Board Micro device. When using TensorFlow Lite to deploy our model to the development board, the lack of support for division operations presented a challenge. While trying to circumvent this issue, we tried using an approximation algorithm, such as the Goldschmidt [74] algorithm, but ran into issues when using this approach too. This forced us to eventually move the division operations to the ARM M7 Core's FPU. It is likely that further improvements in power consumption could be realized if the complete preprocessing operations (including divisions) were executed on the EdgeTPU. When executing the model on the device, inference latency is very low, but the power consumption, even when idle, is relatively high, around 640 mW. Given the limited size of the model, it is likely possible to execute the model on the ARM core exclusively, allowing the EdgeTPU to be completely disabled, further lowering power consumption. This would also eliminate the overhead that is created by transferring data from the ARM core to the EdgeTPU, further lowering inference latency.

Finally, another interesting perspective arises when the results of the comparative analysis in Fig. 18 are taken together with the histogram from Fig. 6. Despite the fact that very simple machine learning models, such as a linear classifier, converge with F1 scores exceeding 99%, a large section of the neural network design space failed to achieve similar F1 scores. This highlights that the neural network architecture design is a nontrivial process, even when the underlying process being modeled itself is fairly simple, further supporting the need for NAS algorithms, such as the one used in this article.

VI. Conclusion

This article covered the use of a reinforcement learning-based NAS strategy to design power-efficient neural networks for the application of detecting active wildfires from satellite imagery. Such NAS systems could allow engineers and researchers to more quickly and easily develop neural networks for processing remote sensing data, lessening the need for expertise in neural network design, and speeding up the design process. We found that the proposed reinforcement learning approach can successfully design neural networks that are accurate at detecting active wildfires, and that use limited computational resources. This work is the first to validate this reinforcement learning approach to NAS in a practical use-case, and in doing so, leaves many avenues open for future research, including a refinement of the multiobjective reinforcement learning strategy used, improved performance estimation algorithms, and the tackling of more complex remote sensing problems.

References

[1] J. Hrisko, P. Ramamurthy, and J. E. Gonzalez, "Estimating heat storage in urban areas using multispectral satellite data and machine learning," *Remote Sens. Environ.*, vol. 252, 2021, Art. no. 112125. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0034425720304983

[2] F. Tonion, F. Pirotti, G. Faina, and D. Paltrinieri, "A machine learning approach to multispectral satellite derived bathymetry," *ISPRS Ann. Photogrammetry, Remote Sens. Spatial Inf. Sci.*, vol. V-3-2020, pp. 565–570, 2020. [Online]. Available: https://isprs-annals.copernicus.org/articles/V-3-2020/565/2020/

[3] C. Cavallo, M. Nones, M. N. Papa, M. Gargiulo, and G. Ruello, "Monitoring the morphological evolution of a reach of the Italian Po river using multispectral satellite imagery and stage data," *Geocarto Int.*, vol. 37, no. 25, pp. 8579–8601, 2022, doi: 10.1080/10106049.2021.2002431.

[4] A. Vali, S. Comai, and M. Matteucci, "Deep learning for land use and land cover classification based on hyperspectral and multispectral Earth observation data: A review," *Remote Sens.*, vol. 12, no. 15, 2020, Art. no. 2495. [Online]. Available: https://www.mdpi.com/2072-4292/12/15/2495

[5] A. Daghouri, Y. El Hachimi, A. Ouhammam, M. A. Chanoui, S. El Hani, and H. Mahmoudi, "Investigating the power budget of a 3 U nanosatellite designed for Earth observation," in *Proc. IEEE 10th Int. Workshop Metrol. Aerosp.*, 2023, pp. 574–579.

[6] S. Dahbi et al., "Power budget analysis for a LEO polar orbiting nanosatellite," in *Proc. Int. Conf. Adv. Technol. Signal Image Process.*, 2017, pp. 1–6.

[7] G. Giuffrida et al., "The ϕ-sat-1 mission: The first on-board deep neural network demonstrator for satellite Earth observation," *IEEE Trans. Geosci. Remote Sens.*, vol. 60, 2022, Art. no. 5517414.

[8] S. Falkner, A. Klein, and F. Hutter, "BOHB: Robust and efficient hyperparameter optimization at scale," in *Proc. 35th Int. Conf. Mach. Learn.*, J. Dy and A. Krause, Eds., Jul. 2018, vol. 80, pp. 1437–1446. [Online]. Available: https://proceedings.mlr.press/v80/falkner18a.html

[9] B. Zoph and Q. Le, "Neural architecture search with reinforcement learning," in *Proc. Int. Conf. Learn. Representations*, 2017, pp. 1–16. [Online]. Available: https://openreview.net/forum?id=r1Ue8Hcxg

[10] H. Liu, K. Simonyan, and Y. Yang, "DARTS: Differentiable architecture search," in *Proc. Int. Conf. Learn. Representations*, 2019, pp. 1–13. [Online]. Available: https://openreview.net/forum?id=S1eYHoC5FX

[11] H. Pham, M. Guan, B. Zoph, Q. Le, and J. Dean, "Efficient neural architecture search via parameters sharing," in *Proc. 35th Int. Conf. Mach. Learn.*, 2018, vol. 80, pp. 4095–4104. [Online]. Available: https://proceedings.mlr.press/v80/pham18a.html

[12] T. Elsken, J. H. Metzen, and F. Hutter, "Efficient multi-objective neural architecture search via lamarckian evolution," in *Proc. Int. Conf. Learn. Representations*, 2019, pp. 1–23. [Online]. Available: https://openreview.net/forum?id=ByME42AqK7

[13] K. Kandasamy, W. Neiswanger, J. Schneider, B. Poczos, and E. P. Xing, "Neural architecture search with Bayesian optimisation and optimal transport," in *Proc. Int. Conf. Adv. Neural Inf. Process. Syst.*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., 2018, vol. 31, pp. 2020–2029. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2018/file/f33ba15effa5c10e873bf3842afb46a6-Paper.pdf

[14] C. White et al., "Neural architecture search: Insights from 1000 papers," 2023, *arXiv:2301.08727*.

[15] A. Cassimon, S. Mercelis, and K. Mets, "Scalable reinforcement learning-based neural architecture search," *Neural Comput. Appl.*, vol. 37, no. 1, pp. 231–261, Jan. 2025, doi: 10.1007/s00521-024-10445-2.

[16] A. Cassimon, S. Mercelis, and K. Mets, "Task adaptation of reinforcement learning-based NAS agents through transfer learning," 2024. [Online]. Available: https://arxiv.org/abs/2412.01420

[17] Dev Board Micro Datasheet, LLC Google, version 1.0, 2022. [Online]. Available: https://coral.ai/static/files/Coral-Dev-Board-Micro-datasheet.pdf

[18] E. Real et al., "Large-scale evolution of image classifiers," in *Proc. 34th Int. Conf. Mach. Learn.*, 2017, vol. 70, pp. 2902–2911.

[19] M. Javaheripi et al., "Litetransformersearch: Training-free neural architecture search for efficient language models," in *Proc. Int. Conf. Adv. Neural Inf. Process. Syst.*, S. Koyejo, S. Mohamed, A. Agarwal, D.

Belgrave, K. Cho, and A. Oh, Eds., 2022, vol. 35, pp. 24254–24267. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2022/file/9949e6906be6448230cdba9a4cb2d564-Paper-Conference.pdf
[20] K. N. Pujari, S. S. Miriyala, P. Mittal, and K. Mitra, "Better wind forecasting using evolutionary neural architecture search driven green deep learning," *Expert Syst. Appl.*, vol. 214, 2023, Art. no. 119063. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417422020814
[21] R. Asthana, J. Conrad, Y. Dawoud, M. Ortmanns, and V. Belagiannis, "Multi-conditioned graph diffusion for neural architecture search," *Trans. Mach. Learn. Res.*, vol. 3, pp. 1–23, 2024. [Online]. Available: https://openreview.net/forum?id=5VotySkajV
[22] M. Li, J. Y. Liu, L. Sigal, and R. Liao, "GraphPNAS: Learning probabilistic graph generators for neural architecture search," *Trans. Mach. Learn. Res.*, vol. 11, pp. 1–21, 2023. [Online]. Available: https://openreview.net/forum?id=ok18jj7cam
[23] C. Ying, A. Klein, E. Christiansen, E. Real, K. Murphy, and F. Hutter, "NAS-bench-101: Towards reproducible neural architecture search," in *Proc. 36th Int. Conf. Mach. Learn.*, K. Chaudhuri and R. Salakhutdinov, Eds., Long Beach, CA, USA, 2019, vol. 97, pp. 7105–7114. [Online]. Available: http://proceedings.mlr.press/v97/ying19a.html
[24] J. N. Siems, L. Zimmer, A. Zela, J. Lukasik, M. Keuper, and F. Hutter, "NAS-bench-301 and the case for surrogate benchmarks for neural architecture search," 2021. [Online]. Available: https://openreview.net/forum?id=1flmvXGGJaa
[25] S. Lu et al., "PINAT: A permutation invariance augmented transformer for NAS predictor," in *Proc. AAAI Conf. Artif. Intell.*, Jun. 2023, vol. 37, no. 7, pp. 8957–8965. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/26076
[26] G. Kadlecová et al., "Surprisingly strong performance prediction with neural graph features," in *Proc. 41st Int. Conf. Mach. Learn.*, 2024, pp. 22771–22816. [Online]. Available: https://openreview.net/forum?id=EhPpZV6KLk
[27] J. Mellor, J. Turner, A. Storkey, and E. J. Crowley, "Neural architecture search without training," in *Proc. 38th Int. Conf. Mach. Learn.*, M. Meila and T. Zhang, Eds., 2021, pp. 7588–7598. [Online]. Available: https://proceedings.mlr.press/v139/mellor21a.html
[28] X. Dong and Y. Yang, "NAS-Bench-201: Extending the scope of reproducible neural architecture search," in *Proc. Int. Conf. Learn. Representations*, 2020, pp. 1–16. [Online]. Available: https://openreview.net/forum?id=HJxyZkBKDr
[29] H. Cai, L. Zhu, and S. Han, "ProxylessNAS: Direct neural architecture search on target task and hardware," in *Proc. Int. Conf. Learn. Representations*, 2019, pp. 1–13. [Online]. Available: https://openreview.net/forum?id=HylVB3AqYm
[30] Z. Zheng, Y. Zhong, A. Ma, and L. Zhang, "FPGA: Fast patch-free global learning framework for fully end-to-end hyperspectral image classification," *IEEE Trans. Geosci. Remote Sens.*, vol. 58, no. 8, pp. 5612–5626, Aug. 2020.
[31] P. Gamba, Pavia University Dataset, 2013. [Online]. Available: https://www.ehu.eus/ccwintco/index.php/Hyperspectral_Remote_Sensing_Scenes#Pavia_Centre_and_University
[32] Y. Wu and K. He, "Group normalization," in *Proc. Eur. Conf. Comput. Vis.*, 2018, pp. 3–19.
[33] Salinas Dataset, 2019. [Online]. Available: https://www.ehu.eus/ccwintco/index.php/Hyperspectral_Remote_Sensing_Scenes
[34] H. I. A. Lab, CASI University of Houston Dataset, "IEEE GRSS data fusion challenge," 2018. [Online]. Available: https://hyperspectral.ee.uh.edu/?page_id=1075
[35] M. Wu, Q. Huang, T. Sui, B. Peng, and M. Yu, "A remote sensing spectral index guided bitemporal residual attention network for wildfire burn severity mapping," *IEEE J. Sel. Topics Appl. Earth Observ. Remote Sens.*, vol. 17, pp. 17187–17206, 2024.
[36] C. Key and N. Benson, "Landscape Assessment: Ground measure of severity, the Composite Burn Index; and remote sensing of severity, the normalized burn ratio," *FIREMON: Fire Effects Monit. Inventory Syst.*, 2006, Art. no. LA1-51.
[37] O. Günay, K. Taşdemir, B. Uğur Töreyin, and A. E. Çetin, "Fire detection in video using LMS based active learning," *Fire Technol.*, vol. 46, no. 3, pp. 551–577, Jul. 2010, doi: 10.1007/s10694-009-0106-8.
[38] P. Barmpoutis, P. Papaioannou, K. Dimitropoulos, and N. Grammalidis, "A review on early forest fire detection systems using optical remote sensing," *Sensors*, vol. 20, no. 22, 2020, Art. no. 6442. [Online]. Available: https://www.mdpi.com/1424-8220/20/22/6442
[39] J. Florath and S. Keller, "Supervised machine learning approaches on multispectral remote sensing data for a combined detection of fire and burned area," *Remote Sens.*, vol. 14, no. 3, 2022, Art. no. 657. [Online]. Available: https://www.mdpi.com/2072-4292/14/3/657
[40] F. Massimetti, D. Coppola, M. Laiolo, S. Valade, C. Cigolini, and M. Ripepe, "Volcanic hot-spot detection using SENTINEL-2: A comparison with MODIS–MIROVA thermal data series," *Remote Sens.*, vol. 12, no. 5, 2020, Art. no. 820. [Online]. Available: https://www.mdpi.com/2072-4292/12/5/820
[41] G. Meoni, R. D. Prete, F. Serva, A. D. Beussche, O. Colin, and N. Longépé, "Unlocking the use of raw multispectral earth observation imagery for onboard artifical intelligence," *IEEE J. Sel. Top. Appl. Earth Observ. Remote Sens.*, vol. 17, pp. 12521–21537, 2024, doi: 10.1109/JSTARS.2024.3418891.
[42] W. Xu and M. J. Wooster, "Sentinel-3 SLSTR active fire (AF) detection and FRP daytime product - algorithm description and global intercomparison to MODIS, VIIRS and Landsat AF data," *Sci. Remote Sens.*, vol. 7, 2023, Art. no. 100087. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2666017223000123
[43] P. Coppo et al., "SLSTR: A high accuracy dual scan temperature radiometer for sea and land surface monitoring from space," *J. Modern Opt.*, vol. 57, no. 18, pp. 1815–1830, 2010, doi: 10.1080/09500340.2010.503010.
[44] W. Xu, M. J. Wooster, J. He, and T. Zhang, "First study of Sentinel-3 SLSTR active fire detection and FRP retrieval: Night-time algorithm enhancements and global intercomparison to MODIS and VIIRS AF products," *Remote Sens. Environ.*, vol. 248, 2020, Art. no. 111947. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0034425720303175
[45] S. Maillard, M. S. Khan, A. Cramer, and E. K. Sancar, "Wildfire and smoke detection using YOLO-NAS," in *Proc. IEEE 3rd Int. Conf. Comput. Mach. Intell.*, 2024, pp. 1–5.
[46] "YOLO-NAS by Deci achieves SOTA performance on object detection using neural architecture search," Deci AI Inc., 2023. [Online]. Available: https://web.archive.org/web/20240519041339/https://deci.ai/blog/yolo-nas-object-detection-foundation-model/
[47] D. Rashkovetsky, F. Mauracher, M. Langer, and M. Schmitt, "Wildfire detection from multisensor satellite imagery using deep semantic segmentation," *IEEE J. Sel. Topics Appl. Earth Observ. Remote Sens.*, vol. 14, pp. 7001–7016, 2021.
[48] O. Ronneberger, P. Fischer, and T. Brox, "U-Net: Convolutional networks for biomedical image segmentation," in *Proc. Int. Conf. Med. Image Comput. Comput.-Assist. Interv.*, N. Navab, J. Hornegger, W. M. Wells, and A. F. Frangi, Eds. Cham, Switzerland, 2015, pp. 234–241.
[49] R. D. Prete, G. Meoni, N. Longepe, F. Serva, O. Colin, and A. D. Beusscher, "THRawS," *Zenodo*, May 8, 2023, doi: 10.5281/zenodo.7908728.
[50] N. Pettorelli, *The Normalized Difference Vegetation Index*. New York, NY, USA: Oxford Univ. Press, 2013.
[51] L. Cicala, C. V. Angelino, N. Fiscante, and S. L. Ullo, "Landsat-8 and Sentinel-2 for fire monitoring at a local scale: A case study on Vesuvius," in *Proc. IEEE Int. Conf. Environ. Eng.*, 2018, pp. 1–6.
[52] C. White, A. Zela, R. Ru, Y. Liu, and F. Hutter, "How powerful are performance predictors in neural architecture search?," in *Proc. Int. Conf. Adv. Neural Inf. Process. Syst.*, 2021, pp. 28454–28469. [Online]. Available: https://proceedings.neurips.cc/paper/2021/file/ef575e8837d065a1683c022d2077d342-Paper.pdf
[53] K. Yu, R. Ranftl, and M. Salzmann, "How to train your super-net: An analysis of training heuristics in weight-sharing NAS," 2020. [Online]. Available: https://openreview.net/forum?id=txC1ObHJ0wB
[54] L. Li and A. Talwalkar, "Random search and reproducibility for neural architecture search," in *Proc. 35th Uncertainty Artif. Intell. Conf.*, R. P. Adams and V. Gogate, Eds., Jul. 2020, pp. 367–377. [Online]. Available: https://proceedings.mlr.press/v115/li20c.html
[55] E. Liberis, L. Dudziak, and N. D. Lane, "μNAS: Constrained neural architecture search for microcontrollers," in *Proc. 1st Workshop Mach. Learn. Syst.*, New York, NY, USA, 2021, pp. 70–79, doi: 10.1145/3437984.3458836.
[56] Y. Akhauri and M. Abdelfattah, "On latency predictors for neural architecture search," in *Proc. Conf. Mach. Learn. Syst.*, P. Gibbons, G. Pekhimenko, and C. D. Sa, Eds., 2024, pp. 512–523. [Online]. Available: https://proceedings.mlsys.org/paper_files/paper/2024/file/f03cb785864596fa5901f1359d23fd81-Paper-Conference.pdf
[57] D. M. Roijers, P. Vamplew, S. Whiteson, and R. Dazeley, "A survey of multi-objective sequential decision-making," *J. Artif. Intell. Res.*, vol. 48, pp. 67–113, 2013.

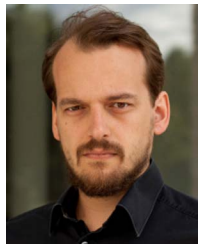
[58] R. Rădulescu, P. Mannion, D. M. Roijers, and A. Nowé, "Multi-objective multi-agent decision making: A utility-based analysis and survey," *Auton. Agents Multi-Agent Syst.*, vol. 34, 2019, Art. no. 10. [Online]. Available: https://link.springer.com/article/10.1007/s10458-019-09433-x
[59] M. Abadi et al., "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. [Online]. Available: https://www.tensorflow.org/
[60] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychol. Rev.*, vol. 65, no. 6, 1958, Art. no. 386.
[61] M. Shkolnik et al., "Robust quantization: One model to rule them all," in *Proc. Int. Conf. Adv. Neural Inf. Process. Syst.*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., Curran Associates, Inc., 2020, pp. 5308–5317. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/3948ead63a9f2944218de038d8934305-Paper.pdf
[62] F. Pedregosa et al., "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, 2011.
[63] C. R. Harris et al., "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020, doi: 10.1038/s41586-020-2649-2.
[64] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *Proc. Int. Conf. Learn. Representations*, 2017, pp. 1–14. [Online]. Available: https://openreview.net/forum?id=SJU4ayYgl
[65] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *Proc. 30th Int. Conf. Mach. Learn.*, 2013, Art. no. 3. [Online]. Available: https://ai.stanford.edu/amaas/papers/relu_hybrid_icml2013_final.pdf
[66] D. Kingma, "Adam: A method for stochastic optimization," in *Proc. Int. Conf. Learn. Representations*, 2015, pp. 1–13. [Online]. Available: https://arxiv.org/abs/1412.6980v5
[67] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, "Dueling network architectures for deep reinforcement learning," in *Proc. 33rd Int. Conf. Mach. Learn.*, M. F. Balcan and K. Q. Weinberger, Eds., New York, NY, USA, 2016, pp. 1995–2003. [Online]. Available: https://proceedings.mlr.press/v48/wangf16.html
[68] D. Horgan et al., "Distributed prioritized experience replay," in *Proc. Int. Conf. Learn. Representations*, 2018, pp. 1–19. [Online]. Available: https://openreview.net/forum?id=H1Dy---0Z
[69] C. Szegedy et al., "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2015, pp. 1–9.
[70] J. LLC, "Joulescope — DC energy analyzer measures current & voltage," 2024. [Online]. Available: https://www.joulescope.com/
[71] G.-M. Sung, Z.-Y. Li, and C.-P. Yu, "Ethernet–USB bridge application-specific integrated circuit incorporating the user datagram protocol and address resolution protocol," *IEEE Access*, vol. 12, pp. 106874–106883, 2024.
[72] "Cal fire perimeter database," California Dept. Forestry Fire Protection, 2025. [Online]. Available: https://www.fire.ca.gov/what-we-do/fire-resource-assessment-program/fire-perimeters
[73] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in *Proc. Int. Conf. Learn. Representations*, 2018, pp.1–12. [Online]. Available: https://openreview.net/forum?id=rJXMpikCZ
[74] R. E. Goldschmidt, "Applications of division by convergence," Ph.D. dissertation, Massachusetts Inst. Technol., Cambridge, MA, USA, 1964.

Amber Cassimon received the master's degree in electronics and ICT engineering technology from the University of Antwerp, Antwerp, Belgium, in 2019.

She is a Ph.D. Researcher with the Faculty of Applied Engineering Sciences, University of Antwerp. Her master's thesis was regarding the use of neural architecture search to design neural networks for embedded devices. Her research focuses on the use of reinforcement learning agents for NAS, focusing on the design of neural networks suitable for environments with limited resources.

Phil Reiter received the B.Eng. degree in computer engineering with specialization in biotechnology from McGill University, Montreal, QC, Canada, in 2006 and the M.Sc. degree with distinction in machine learning and deep learning from the University of Strathclyde, Glasgow, U.K., in 2020.

He is a Senior Researcher and Project Manager advancing extreme low-power and biologically inspired AI for extreme environments in the IDLab research group, collaborating both with imec and the University of Antwerp, Antwerp, Belgium. He has more than 15 years of research and industry experience working in beyond leading-edge semiconductor and machine intelligence domains.



Siegfried Mercelis received the master's degrees in music production from the HOGENT University of Applied Sciences and Arts, Ghent, Belgium, in 2008, and the Engineering (electronics and ICT) from KdG University of Applied Sciences and Arts, Antwerp, Belgium, in 2012, and the Ph.D. degree in applied engineering from the University of Antwerp, Antwerp, Belgium, in December 2016.

From 2012 to 2016, he was employed with Van den Berghe R&D under a Baekeland Ph.D. mandate on the subject of optimizing and parallelizing real-time media applications. He is an Assistant Professor with the University of Antwerp. He is currently employed as an Assistant Professor and a Manager of the adaptive intelligence program with the University of Antwerp. His team of 30+ AI researchers is committed to bridging the gap between academic AI research and industry in domains, such as chemical process control, autonomous shipping, smart buildings, logistics, and mobility.

Kevin Mets received the M.Sc. degree in computer science—information and communication technology and the Ph.D. degree in computer science, on the topic optimal and scalable management of smart power grids with electric vehicles, from Ghent University, Ghent, Belgium, in 2009 and 2015, respectively.

He is currently an Assistant Professor with the Faculty of Applied Engineering, University of Antwerp, Antwerp, Belgium, and a Member of the IDLab research group. His research interests include graph neural networks and deep reinforcement learning with a particular focus on topics, such as transfer learning and continual learning.