Learning to reinforcement learn for Neural Architecture Search

Jorge Gomez Robles j.gomez.robles@student.tue.nl
Joaquin Vanschoren j.vanschoren@tue.nl
Department of Mathematics and Computer Science
Eindhoven University of Technology
Eindhoven 5612 AZ, The Netherlands

Abstract

Reinforcement learning (RL) is a goal-oriented learning solution that has proven to be successful for Neural Architecture Search (NAS) on the CIFAR and ImageNet datasets. However, a limitation of this approach is its high computational cost, making it unfeasible to replay it on other datasets. Through meta-learning, we could bring this cost down by adapting previously learned policies instead of learning them from scratch. In this work, we propose a deep meta-RL algorithm that learns an adaptive policy over a set of environments, making it possible to transfer it to previously unseen tasks. The algorithm was applied to various proof-of-concept environments in the past, but we adapt it to the NAS problem. We empirically investigate the agent's behavior during training when challenged to design chain-structured neural architectures for three datasets with increasing levels of hardness, to later fix the policy and evaluate it on two unseen datasets of different difficulty. Our results show that, under resource constraints, the agent effectively adapts its strategy during training to design better architectures than the ones designed by a standard RL algorithm, and can design good architectures during the evaluation on previously unseen environments. We also provide guidelines on the applicability of our framework in a more complex NAS setting by studying the progress of the agent when challenged to design multi-branch architectures.

Keywords: Neural Architecture Search, Deep Meta-Reinforcement Learning, Image Classification

1. Introduction

Neural networks have achieved remarkable results in many fields, such as that of Image Classification. Crucial aspects of this success are the choice of the neural architecture and the chosen hyperparameters for the particular dataset of interest; however, this is not always straightforward. Although state-of-the-art neural networks can inspire the design of other architectures, this process heavily relies on the designer's level of expertise, making it a challenging and cumbersome task that is prone to deliver underperforming networks.

In an attempt to overcome these flaws, researchers have explored various techniques under the name of Neural Architecture Search (NAS) (Elsken et al., 2018). In NAS, the ultimate goal is to come up with an algorithm that takes any arbitrary dataset as input and outputs a well-performing neural network for some learning task of interest, so that we can accelerate the design process and remove the dependency on human intervention. Nevertheless, coming up with a solution of this kind is a complicated endeavor where researchers have to deal with several aspects such as the type of the networks that they

consider, the scope of the automation process, or the search strategy applied. A particular search strategy for NAS is reinforcement learning (RL), where a so-called *agent* learns how to design neural networks by sampling architectures and using their numeric performance on a specific dataset as the reward signals that guide the search. Popular standard RL algorithms such as Q-learning or Reinforce have been used to design state-of-the-art Convolutional Neural Networks (CNNs) for classification tasks on the CIFAR and ImageNet datasets (Pham et al., 2018; Cai et al., 2018; Zhong et al., 2018; Zoph and Le, 2016; Baker et al., 2016), but little attention is paid to deliver architectures for other datasets. In an attempt to fill that gap, a suitable alternative is deep meta-RL (Wang et al., 2016; Duan et al., 2016), where the *agent* acts on various environments to learn an *adaptive* policy that can be transferred to new environments.

In this work, we apply deep meta-RL to NAS, which, to the best of our knowledge, is a novel contribution. The environments that we consider are associated with standard image classification tasks on datasets with different levels of hardness sampled from a meta-dataset (Triantafillou et al., 2019). Our main experiments focus on the design of chain-structured networks and show that, under resource constraints, the resulting policy can adapt to new environments, outperform standard RL, and design better architectures than the ones inspired by state-of-the-art networks. We also experiment with extending our approach to the design of multi-branch architectures so that we can give directions for future work.

The remainder of this report is structured as follows. First, in Section 2, we introduce the preliminary concepts required to understand our work. Next, in Section 3, we discuss the related work for both reinforcement learning and NAS. In Section 4, we formally introduce our methodology, and in Section 5, the framework developed to implement it. In Section 6, we define the experiments, and in Section 7, we show the results. Finally, in Section 8, the conclusions are set out.

2. Preliminaries

2.1 Reinforcement learning

Reinforcement learning (RL) is an approach to automate goal-directed learning (Sutton and Barto, 2012). It relies on two entities that interact with each other: an *environment* that delivers information of its *state*, and an *agent* that using such information learns how to achieve a *goal* in the environment. The interaction is a bilateral communication where the agent performs *actions* to modify the *state* of the environment, which responds with a numeric *reward* measuring how good the action was to achieve the *goal*. Typically, the sole interest of the agent is to improve its decision-making strategy, known as the *policy*, to maximize the total reward received over the whole interaction trial since this will lead it to the desired goal. More strictly, RL is formalized using finite Markov Decision Processes (MDPs) as in Definition 1 borrowed from Duan et al. (2016), resulting in the agent-environment interaction illustrated in Figure 1.

Definition 1 (Reinforcement Learning) *We define a discrete-time finite-horizon discounted MDP* $M = (\mathcal{X}, \mathcal{A}, \mathcal{P}, r, \rho_0, \gamma, T)$*, in which* $\mathcal{X}$ *is a state set,* $\mathcal{A}$ *an action set,* $\mathcal{P} : \mathcal{X} \times \mathcal{A} \times \mathcal{X} \mapsto \mathbb{R}_+$ *a transition probability distribution,* $r : \mathcal{X} \times \mathcal{A} \mapsto [-R_{max}, R_{max}]$

a bounded reward function, $\rho_0 : \mathcal{X} \mapsto \mathbb{R}_+$ *an initial state distribution,* $\gamma \in [0, 1]$ *a discount factor, and* T *the horizon.* Reinforcement learning *typically aims to optimize a stochastic policy* $\pi_\theta : \mathcal{X} \times \mathcal{A} \mapsto \mathbb{R}_+$ *by maximizing the expected reward, modeled as* $\eta(\pi_\theta) = \mathbb{E}_\tau[\sum_{t=0}^{T} \gamma^t r(x_t, a_t)]$*, where* $\tau = (s_0, \alpha_0, ...)$ *denotes the whole trajectory,* $x_t \in \mathcal{X}$*,* $x_0 \sim \rho_0(x_0)$*,* $a_t \in \mathcal{A}$*,* $a_t \sim \pi_\theta(a_t|x_t)$*, and* $x_{t+1} \sim \mathcal{P}(x_{t+1}|x_t, a_t)$*.*

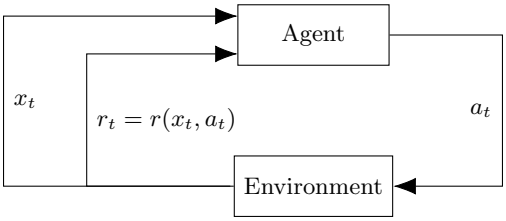


Figure 1: Graphic representation of the reinforcement learning interaction. Every time the agent performs an action a_t, the environment modifies its state x_{t-1} to x_t, computes the reward r_t and sends both values to the agent, who uses them to optimize its policy.

2.2 Neural Architecture Search

Neural Architecture Search (NAS) is the process of automating the design of neural networks. In order to formalize this definition, it is convenient to refer to the survey of Elsken et al. (2018), which characterize a NAS work with three variables: the *search space*, the *search strategy*, and the *performance estimation strategy*. Figure 2 illustrates the interaction between these variables.

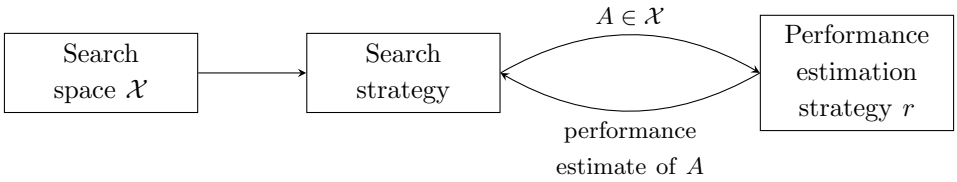


Figure 2: An illustration of the three NAS variables interacting (Elsken et al., 2018). At any moment during the search, the *search strategy* samples an architecture A from the *search space* and sends it to the *performance estimation strategy*, which returns the performance estimate. By design, the *search space* and the *performance estimation strategy* are named after the variables in Definition 1 since they are typically equivalent in NAS within the reinforcement learning setting.

The *search space* is the set of architectures considered in the search process. It is possible to define different spaces by constraining attributes of the networks, such as the maximum depth allowed, the type of layers to use, or the connections permitted between layers. A common abstraction inspired in popular networks is to separate the search spaces in *chain* structures and *multi-branch* structures that can be either *complete* neural networks or *cells* that can be used to build more complex networks, as illustrated in Figure 3.

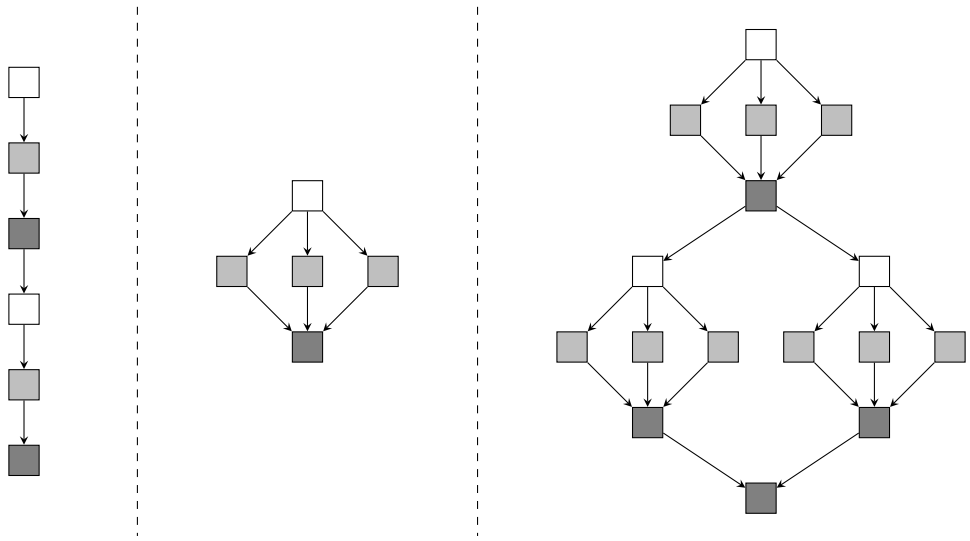


Figure 3: Examples of networks belonging to different search spaces. On the left, a *chain-structured* network. On the center, a *multi-branch* network. On the right, the same multi-branch structure used as a *cell* repeated multiple times to build a more complex network.

On the other hand, the *search strategy* is simply the algorithm used to perform the search. The choices range from naive approaches such as random search to more sophisticated ones like reinforcement learning (Baker et al., 2016; Zoph and Le, 2016), evolutionary algorithms (Real et al., 2018), or gradient descent search (Liu et al., 2018).

Lastly, the *performance estimation strategy* is the function used to measure the goodness of the sampled architectures. Formally, it is a function $R_D : \mathcal{X} \mapsto \mathbb{R}$ evaluating an architecture on a dataset D. The *vanilla* estimation strategy is the test accuracy after training of a network, but different alternatives are proposed to try delivering an accurate estimate in a short time since expensive training creates a bottleneck in the search process.

3. Related work

3.1 Reinforcement learning

The key to reinforcement learning is the algorithm used to optimize the *policy* π_θ (see Definition 1). Through the years, researchers have proposed different algorithms, such as Reinforce (Williams, 1992), Q-learning (Watkins and Dayan, 1992), Actor-Critic (Konda, 2002), Deep-Q-Network (DQN) (Mnih et al., 2013), Trust Region Policy Optimization (TRPO) (Schulman et al., 2015), and Proximal Policy Optimization (PPO) (Schulman et al., 2017). These algorithms have successfully solved problems in a variety of fields, from robotics (Kober et al., 2013) and video games (Bouzy and Chaslot, 2006; Mnih et al., 2013) to traffic-control (Arel et al., 2010) or computational resources management (Mao et al., 2016), showing the power and utility of the reinforcement learning framework. Despite its success, a theoretical flaw of RL is that the policy learned only captures the one-to-one *state-action* relation of the environment in question, making it necessary to perform in-

dividual runs for every new environment of interest. The latter is a costly trait of this standard form of RL since it typically requires thousands of steps to converge to an optimal policy.

A research area that addresses this problem is meta-RL, in which agents are trained to learn transferable policies that do not require training from scratch on new problems. We identify two types of meta-RL algorithms: the ones that learn a *good* initialization for the neural networks representing the policy[1], and the ones that learn policies that can adapt their decision-making strategy to new environments, ideally without further training. First, Model Agnostic Meta-Learning (MAML) (Finn et al., 2017) learns an initialization that allows few-shot learning in RL; however, it relies on the strong assumption of working with a distribution of similar environments, which cannot always be guaranteed, thus limiting its scope. Second, two algorithms have been proposed to learn policies that, once deployed, can adapt their decision-making strategy to new environments, ideally without further training: *Learning to reinforcement learn* (Wang et al., 2016) and RL^2 (Duan et al., 2016). These methods aim to learn a more sophisticated policy modeled by a Recurrent Neural Network (RNN) that captures the relation between states, actions, and meta-data of past actions. What emerges is a policy that can adapt its strategy to different environments. The main difference between the two works is the set of environments considered: for Wang et al. (2016) they come from a parameterized distribution, whereas for Duan et al. (2016) they are relatively unrelated. The latter is an essential advantage over MAML, making them more suitable for scenarios where a distribution of environments cannot be guaranteed. Third, Simple Neural AttentIve Learner (SNAIL) extends the idea behind *Learning to reinforcement learn* and RL^2 by using a more powerful temporal-focused model than the simple RNN. We note that none of these approaches have been applied to NAS.

3.2 Neural Architecture Search

As introduced earlier in Section 2.2, it is possible to address Neural Architecture Search (NAS) in different ways. Remarkable results have been achieved by applying optimization techniques such as Bayesian optimization, evolutionary algorithms, gradient-based search, and reinforcement learning. We are interested in reinforcement learning for NAS, due to the variety of works that have achieved state-of-the-art results. For other work in NAS, we refer to the survey of Elsken et al. (2018).

Although the ultimate goal of NAS is to come up with a straightforward fully-automated solution that can deliver a neural architecture for a machine learning task on any dataset of interest, there exist several factors that impede that ambition. Perhaps the most important of these factors is the high computational cost of NAS with reinforcement learning, which imposes constraints on different elements that impact the scope of the solutions. The first bottleneck is the computation of the reward, which typically is the test accuracy of the sampled architectures after training. Because of the expensiveness of such computation, researchers have proposed various *performance estimation strategies* to avoid expensive training procedures, and they have also imposed some constraints over the *search space* considered so that a lower number of architectures get sampled and evaluated. For the

1. The term *deep* meta-reinforcement learning comes from the usage of *deep* models, such as neural networks, to represent the policy to be learned.

first aspect, we observe several relaxations: reducing the number of training epochs (Baker et al., 2016; Zhong et al., 2018), sharing of weights between similar networks (Cai et al., 2018; Pham et al., 2018), or entirely skipping the train-evaluation procedure by predicting the performance of the architectures (Zhong et al., 2018). Although these alternatives have successfully reduced the computation time, they pay little attention to the effect of their potentially unfair estimations on the decision-making of the agent, and therefore, one should treat them carefully. On the other hand, for the *search space*, crucial choices are the cardinality of the space and the complexity of the architectures. Some researchers opt for ample spaces with various types of layers and no restrictions in their connections (Zoph and Le, 2016; Zoph et al., 2017; Zhong et al., 2018), whereas others prefer them smaller, such as a chain-structured space (Baker et al., 2016), or a multi-branch space modeled as a fully-connected graph with low cardinality (Pham et al., 2018).

It is important to note that an approach can dramatically reduce its computation time with relaxations on the search space alone. For instance, the same methodology can decrease its computational cost by a factor of 7 (28 days to 4 days, with hundreds of GPUs in both cases) if the space is restricted in the types of layers and the number of elements allowed in the architectures (Zoph and Le, 2016; Zoph et al., 2017). Furthermore, a constrained search space used jointly with some performance estimation strategy can reduce the cost to only 1 day with 1 GPU such as in BlockQNN-V2 (Zhong et al., 2018) and ENAS (Pham et al., 2018); however, this drastic reduction of the computational time should be treated with caution. In the case of BlockQNN-V2, the estimation of the performance of the networks (i.e., accuracy at a given epoch) depends on a surrogate prediction model that is not studied in detail by the authors, thus leaving room for potentially wrong predictions. On the other hand, a recent investigation (Singh et al., 2019) shows that the quality of the networks delivered by ENAS is not a consequence of reinforcement learning, but of the search space, which contains a majority of well-performing architectures that can be explored with a less expensive procedure such as random search, therefore losing its character of *artificially smart* search.

Another factor impacting a NAS with reinforcement learning work is the *input dataset* used. Although they usually transfer the best CIFAR-based architecture designed by the agent to the ImageNet dataset (Baker et al., 2016; Zoph and Le, 2016; Zoph et al., 2017; Cai et al., 2018; Pham et al., 2018), none of them make the agent design networks for other datasets. Furthermore, none of the works give insight on how using a different dataset could affect the complexity of the search. We believe that the lack of study for other datasets is ascribed to the costly task-oriented design of the reinforcement learning algorithms used, Q-learning and Reinforce, that requires to train the agent from scratch for every environment (i.e., a dataset) of interest. The authors do not justify the choice of these algorithms; hence, it would be desirable to study other reinforcement learning algorithms in the same NAS scenarios.

4. Methodology

We aim to improve the performance of Neural Architecture Search (NAS) with reinforcement learning (RL) by using meta-learning. We, therefore, build a meta-RL system that can learn across environments and adapt to them. We split the system into two components: the

NAS variables and the reinforcement learning framework. For the reinforcement learning framework, we make use of a deep meta-RL algorithm that follows the same line of *Learning to reinforcement learn* (Wang et al., 2016) and RL^2 (Duan et al., 2016), with some minor adaptations in the meta-data employed and the design of the episodes. The environments that we consider are neural architecture design tasks for different datasets sampled from the meta-dataset collection (Triantafillou et al., 2019). On the other hand, for the NAS elements, we work with a slightly modified version of the search space of BlockQNN (Zhong et al., 2018) and similarly, we use the test accuracy after early-stop training as the reward associated with the sampled architectures. In the remainder of the section, we discuss these elements further.

4.1 The Neural Architecture Search elements

As described in Section 2.2, three NAS variables characterize a research work in this area: the search strategy, the search space, and the performance estimation strategy. In our case, we constrain the search strategy to a deep meta-reinforcement learning algorithm that we explain in detail in Section 4.2, and thus, here we only elaborate on the remaining two.

4.1.1 The search space

The set of architectures considered in our work is inspired by BlockQNN (Zhong et al., 2018), which defines the search space as all architectures that can be generated by sequentially stacking $d \in \mathbb{N}$ vectors from a so-called Network Structure Code (NSC) space containing encodings of the most relevant layers for CNNs. An NSC vector has information of the type of a layer, the value of its most important hyperparameter, its position on the network, and the allowed incoming connections (i.e., the inputs) so that it becomes possible to represent any architecture as a list of NSCs. The NSC definition is flexible in that it can easily be modified or extended and, moreover, it allows us to define an equivalent discrete action space for the reinforcement learning agent as described in Section 4.2.2.

In Table 1, we present the NSC space for our implementation. Given a list of NSC vectors representing an architecture, the network is built following the next rules: firstly, based on BlockQNN's results, if a *convolution* layer is found then a Pre-activation Convolutional Cell[2] (PCC) (He et al., 2016) with 32 units[3] is used; secondly, the *concatenation* and *addition* operations create padded versions of their inputs if they have different shapes; thirdly, if at the end of the building process the network has two or more leaves then they get merged with a *concatenation* operation[4]. Figure 4 illustrates these rules.

4.1.2 The performance estimation strategy

Our estimation of the long-term performance of the designed networks closely follows the early-stop approach of BlockQNN-V1 (Zhong et al., 2018), but we ignore the penalization

2. The PCC stacks a ReLU, a convolution, and a batch normalization unit.
3. The selection of the number of units is made to reduce the cost of the training of the networks.
4. The last two rules do not apply for chain-structured architectures since no merge operations are needed.
5. The kernel size is an attribute for the convolutions, whereas for the pooling elements it refers to the layer's pool size.

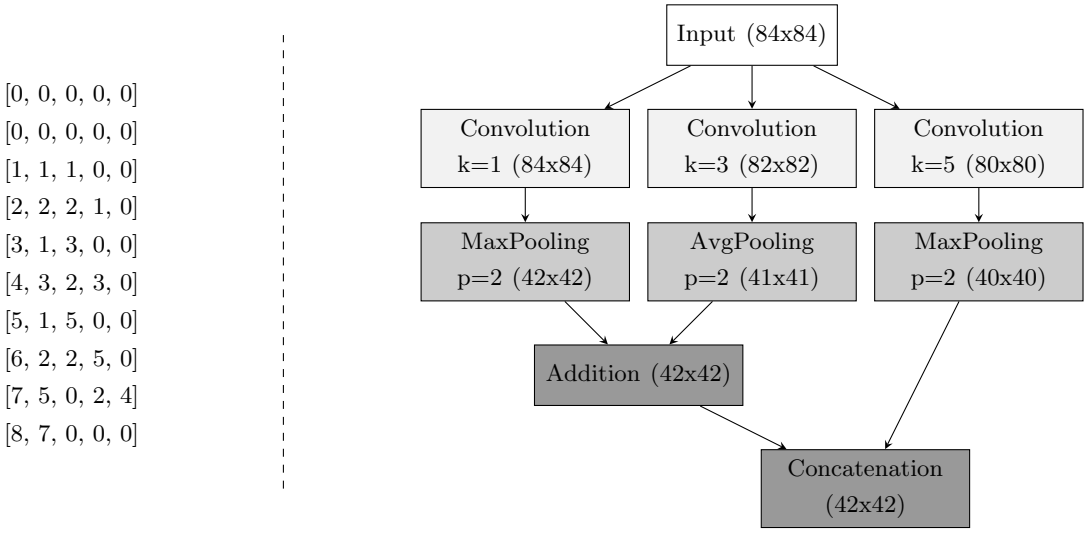


Figure 4: Example of an architecture sampled from the search space of our approach. On the left, a list of Neural Structure Codes (NSCs); on the right, the corresponding network after the application of the rules. For the sake of simplicity, in this example, the convolutions are assumed to have one filter only.

Name	**Index**	**Type**	**Kernel size**[5]	**Predecessor 1**	**Predecessor 2**
Convolution	**T**	1	{1, 3, 5}	**K**	∅
Max Pooling	**T**	2	{2, 3}	**K**	∅
Average Pooling	**T**	3	{2, 3}	**K**	∅
Addition	**T**	5	∅	**K**	**K**
Concatenation	**T**	6	∅	**K**	**K**
Terminal	**T**	7	∅	∅	∅

Table 1: The subset of the NSC space used, presented as in BlockQNN (Zhong et al., 2018). The changes with respect to the original BlockQNN space are: a) the *identity* operator (**Type** 4) is omitted; b) the pool size values changed from the original set $\{1, 3\}$ to $\{2, 3\}$ because a pool size of 1 does not contribute to any reduction. The set $\mathbf{T} = \{1, 2 \ldots, d\}$ refers to the position of each layer in the network, where d is the maximum depth, and $\mathbf{K} = \{0, 1, 2, \ldots, \text{current layer index} - 1\}$ the index of its predecessor.

of the network's FLOPs and density since we have empirically ascertained that it is too strict when the classification task is difficult (i.e., when low accuracy values are expected).

The choice of an early-stop strategy is made to help reduce the computational cost of our approach. In short, for every sampled architecture $\mathcal{N}$ a prediction module is appended, and the network is then trained for a small number of epochs to obtain its accuracy on a test set, which is the final estimation of its long-term performance. The datasets considered are balanced, and their train and test splits are designed beforehand (see Section 4.2.3).

As in BlockQNN, for the prediction module we stack a fully-connected dense layer with 1024 units and ReLU activation function, a dropout layer with rate of 0.4, a dense layer

with the number of units equals to the desired number of classes to predict and linear activation function, and a softmax that outputs the probabilities per class. The training is performed to minimize the cross-entropy loss using the Adam Optimizer (Kingma and Ba, 2015) with the parameters used in BlockQNN: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon_{\text{ADAM}} = 10e^{-8}$, and $\alpha_{\text{ADAM}} = 0.001$ that is reduced by a factor of 0.2 every five epochs. After training, the network is evaluated on a test set by fixing the network's weights and selecting the class with the highest probability to be the final prediction per observation in the set so that the standard accuracy $\text{ACC}_{\mathcal{N}}$ can be returned.

4.2 The reinforcement learning framework

The deep-meta-RL framework that we propose is different from standard RL in two main aspects. First, the *agent* is challenged to face more than one *environment* during training, and second, the distribution over the *reward* domain learned by the agent is now dependant on the whole history of *states*, *actions*, and *rewards*, instead of the simple *state-action* pairs. In the remainder of the section, we describe each of the RL elements.

4.2.1 The states

A state $x_i \in \mathcal{X}$ is a multidimensional array of size $d \times 5$, storing d NSC vectors sorted by layer index. While this representation is programmatically easy to control, it is not ideal in a machine learning setting. In particular, we note that every element of an NSC vector is a categorical variable. Therefore, when required, every NSC vector in x_t is transformed as follows: the layer's type[6] is encoded into a one-hot vector of size 8, the predecessors into a one-hot vector of size $(d + 1)$, and the kernel size into a one-hot vector of size $(k + 1)$ with $k = \max(\text{kernel\_size})$. The transformation ignores the layer index because the state implicitly incorporates the information of the position of each layer due to sorting. This encoding results in a multidimensional array[7] of size $d \times (2d + k + 11)$. Figure 5 illustrates this transformation.

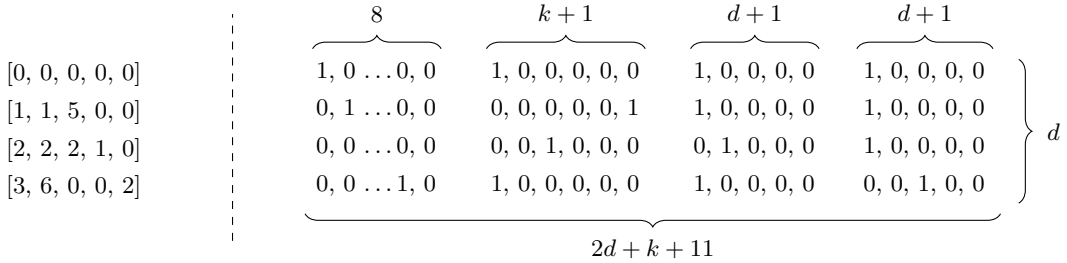


Figure 5: The different representations of a state. In this example, a state x_t contains $d = 4$ NSC vectors. On the left, a network as a list of NSC vectors; on the right, the same network in its encoded representation. In our work, $k = 5$ as observed in Table 1.

6. This size is the result of having 7 types of layers (see Section 4.1.1) plus the type 0 representing an empty layer.
7. When working with chain-structured networks the second predecessor is always omitted, reducing the dimensionality of the encoding to $d \times (d + k + 10)$.

4.2.2 The action space

We formulate the action space $\mathcal{A}$ as a discrete space of 14 actions listed in Table 2. Each action $a_i \in \mathcal{A}$ can either append a new element from the NSC space to a state $x_j \in \mathcal{X}$ or control two pointers, p_1 and p_2, for the indices of the predecessors to use for the next NSC vector. We note that for the chained-structured networks no pointers are required since the predecessor is always the previous layer, and neither do the merging operations *addition* and *concatenation*, making it possible to reduce the action space to 8 actions only.

Action ID	**Description**
A0	Add *convolution* with kernel_size = 1, using predecessor p_1
A1	Add *convolution* with kernel_size = 3, using predecessor p_1
A2	Add *convolution* with kernel_size = 5, using predecessor p_1
A3	Add *max-pooling* with pool_size = 2, using predecessor p_1
A4	Add *max-pooling* with pool_size = 3, using predecessor p_1
A5	Add *avg-pooling* with pool_size = 2, using predecessor p_1
A6	Add *avg-pooling* with pool_size = 3, using predecessor p_1
A7	Add *terminal* state.
A8	Add *addition* with predecessors p_1 and p_2
A9	Add *concatenation* with predecessors p_1 and p_2
A10	Shift p_1 one position up (i.e., $p_1 = p_1 + 1$)
A11	Shift p_1 one position down (i.e., $p_1 = p_1 - 1$)
A12	Shift p_2 one position up (i.e., $p_2 = p_2 + 1$)
A13	Shift p_2 one position down (i.e., $p_2 = p_2 - 1$)

Table 2: The action space proposed, which is compliant with the NSC space of section 4.1.1.

4.2.3 The environments

In our work, an environment is a neural architecture design task for image classification on a specific dataset of interest. The goal for an agent on this environment is to come up with the best architecture possible after interacting for a certain number of time-steps. At any time-step t, the environment's state is $x_t \in \mathcal{X}$, which is the NSC representation of a neural network N_t. The reward $r_t \in [0, 1]$ associated with x_t is a function of the network's accuracy $\text{ACC}_{N_t} \in [0, 1]$ (Section 4.1.2). The initial state x_0 of the environment is an empty architecture.

An agent can interact with the environment through a set of episodes by performing actions $a_t \in \mathcal{A}$. In our terminology, an *episode* is the trajectory from a reset of the environment's state until a termination signal. The environment triggers the termination signal in the following cases: a) the predecessors p_1 and p_2 are out of bounds after the execution of a_t, b) a_t is a *terminal* action, c) x_t contains d NSC elements (the maximum depth) after performing a_t, d) the total number of actions executed in the current episode is higher than a given number τ, or e) the action led to an invalid architecture. The agent-environment interaction process is formalized in Algorithm 1.

Algorithm 1 Agent-environment interaction

procedure INTERACT(Agent, Environment, Dataset, t_{max}, σ)
    $done \leftarrow$ False
    $t \leftarrow 0$
    Environment.reset_to_initial_state()
    while $t < t_{max}$ **do**
        $a_t \leftarrow$ Agent.get_next_action()
        $x_t \leftarrow$ Environment.update_state(a_t)
        $N \leftarrow$ Environment.build_network(x_t)
        $\text{ACC}_{N_t} \leftarrow$ N.accuracy(Dataset)
        if a_t is shifting **then**
            $r_t \leftarrow \sigma \cdot \text{ACC}_{N_t}$
        else
            $r_t \leftarrow \text{ACC}_{N_t}$
        $done \leftarrow$ Environment.is_termination()
        Agent.learn($x_t, a_t, r_t, done$)
        if $done$ **then**
            Environment.reset_to_initial_state()
            $done \leftarrow$ False

As mentioned in the beginning of Section 4, we work with more than one environment. Specifically, we define five environments, each one associated with a different dataset sampled from the meta-dataset collection (Triantafillou et al., 2019). The datasets are listed in Table 3 and they were selected as explained in Appendix A. All datasets have balanced classes. In order to evaluate the accuracy of a network N_t, for any dataset we perform a deterministic 1/3 train-test split and follow the pre-processing that has been initially proposed by the meta-dataset authors so that the images are resized to a shape of $84 \times 84 \times 3$ using bilinear interpolation.

Dataset ID	Dataset name	Usage	N classes	N observations
aircraft	FGVC-Aircraft	Validation	100	10000
cu_birds	CUB-200-2011	Validation	200	11788
dtd	Describable Textures	Train	47	5640
omniglot	Omniglot	Train	1623	32460
vgg_flower	VGG Flower	Train	102	8189

Table 3: List of datasets considered for the environments. They are sampled from the meta-dataset (Triantafillou et al., 2019) as explained in Appendix A.

.

4.2.4 Deep meta-reinforcement learning

Our deep meta-RL approach, illustrated in Figure 6, is based on the work of Wang et al. (2016) and Duan et al. (2016). They propose to learn a policy that, in addition to the *state-*

action pairs of standard RL, uses the current time-step in the agent-environment interaction (the temporal information) as well as the previous action and reward. In this way, the agent can learn the relation between its past decisions and the current action. However, we introduce a modification in the temporal information, by considering the relative step within an episode instead of the global time-step so that the agent can capture the relation between changes in a neural architecture.

Formally, let $\mathcal{D}$ be a set of Markov Decision Processes (MDPs). Consider an agent embedding a Recurrent Neural Network (RNN) - with internal state h - modeling a policy π. At the start of a *trial*, a new task $m_i \in \mathcal{D}$ is sampled, and the internal state h is set to zeros (empty network). The agent then executes its action-selection strategy for a certain number t_{max} of discrete time-steps, performing n episodes of maximum length l depending on the environment's rules. At each step t (with $0 \leq t \leq t_{max}$) an action $a_t \in A$ is executed as a function of the observed history $H_t = \{x_0, a_0, r_0, c_0, ..., x_{t-1}, a_{t-1}, r_{t-1}, c_{t-1}, x_t\}$ (set of states $\{x_s\}_{0 \leq s \leq t}$, actions $\{a_s\}_{0 \leq s < t}$, rewards $\{r_s\}_{0 \leq s < t}$, episode-related steps $\{c_s\}_{0 \leq s \leq l}$) and a reward r_t is obtained. At the very beginning of the trial, the action a_0 is sampled at random from a uniform distribution of all actions available, and the state x_0 is given by the environment's rules. The RNN's weights are trained to maximize the total discounted reward accumulated during each *trial*. The evaluation consists of resetting h and fixing π to run an interaction with a new MDP $m_e \notin \mathcal{D}$.

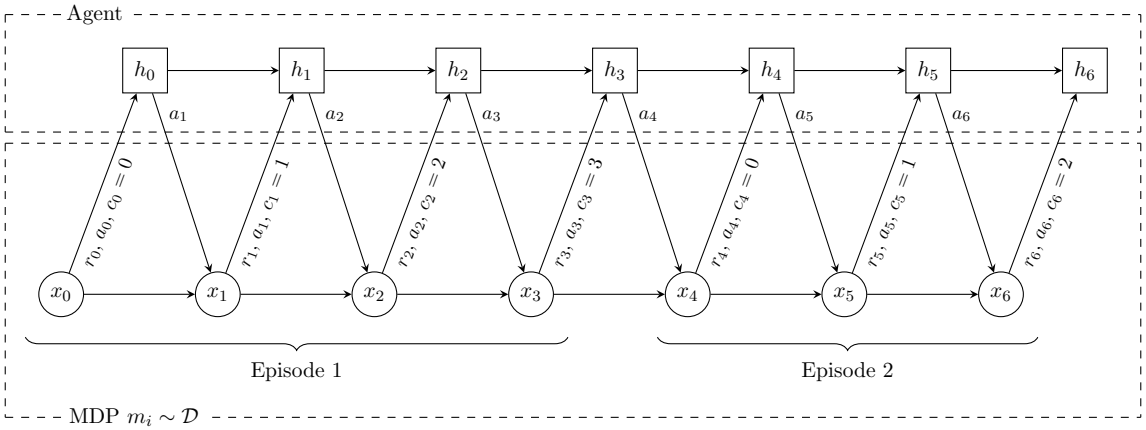


Figure 6: A graphic representation, inspired by the RL^2 illustration (Duan et al., 2016), of our deep meta-reinforcement learning framework. In this example, the trial consists of $t_{max} = 6$ time-steps, and the agent is able to complete two episodes of different length. c_s is a counter of the current step in the episode and it gets reset at the start of any new episode. The states x_0 and x_4 are shown to be different, although in practice the initial state of an episode could always be the same.

4.2.5 The policy optimization algorithm

Similarly to Wang et al. (2016), we make use of the synchronous Advantage Actor-Critic (A2C) (Mnih et al., 2016) with one worker. As it can be observed in Figure 7, the only

change in the A2C network is in the input of the recurrent unit, so that the updates of the network's parameters remain unchanged.

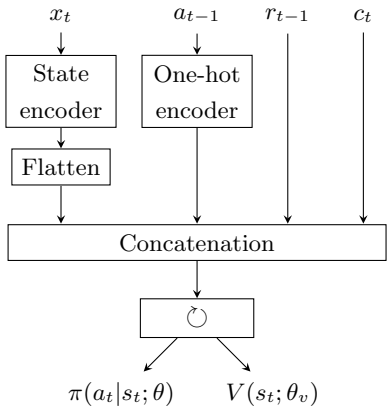


Figure 7: Illustration of the *meta*-A2C architecture. In our implementation, the "State encoder" follows the procedure explained in Section 4.2.1, and the recurrent layer is an LSTM with 128 units.

Formally, let t be the current time step, $s_t = x_t \cdot a_{t-1} \cdot r_{t-1} \cdot c_t$ a concatenation of inputs, $\pi(a_t|s_t;\theta)$ the policy, $V(s_t;\theta_v)$ the value function, H the entropy, $j \in \mathbb{N}$ the horizon, $\gamma \in (0, 1]$ the discount factor, η the regularization coefficient, and $R_t = \sum_{i=0}^{j-1} \gamma^i r_{t+i}$ the total accumulated return from time step t. The gradient of the objective function is:

$$\nabla_\theta \log \pi(a_t|s_t;\theta) \underbrace{(R_t - V(s_t;\theta_v))}_{\text{Advantage estimate}} + \underbrace{\eta \nabla_\theta H(\pi(s_t;\theta))}_{\text{Entropy regularization}} \tag{1}$$

As it is usually the case for A2C, the parameters θ and θ_v are shared except for the ones in output layers. For a detailed description of the algorithm, we refer to the original paper (Mnih et al., 2016).

5. Evaluation framework

The current Neural Architecture Search (NAS) solutions lack a crucial element: an open-source framework for reproducibility and further research. Specifically for NAS with reinforcement learning, it would be desirable to build on a programming interface that allows researchers to explore the effect of different algorithms on the same NAS environment. In an attempt to fill this gap, we have developed the NASGYM[8], a python OpenAI Gym (Brockman et al., 2016) environment that can jointly be used with all the reinforcement learning algorithms exposed in the OpenAI baselines (Dhariwal et al., 2017).

We make use of the object-oriented paradigm to abstract the most essential elements of NAS as a reinforcement learning problem, resulting in a system that can be extended to perform new experiments, as displayed in Figure 8. Although the defaults in the NASGYM are

8. Source code available at: github.com/gomerudo/nas-env

the elements in our methodology, the system allows us to easily modify the key components, such as the performance estimation strategy, the action space, or the Neural Structure Code space. We also provide an interface to use a database of experiments that can help to store previously computed rewards, thus reducing the computation time of future trials. All the deep learning components are built with TensorFlow v1.12 (Abadi et al., 2015).

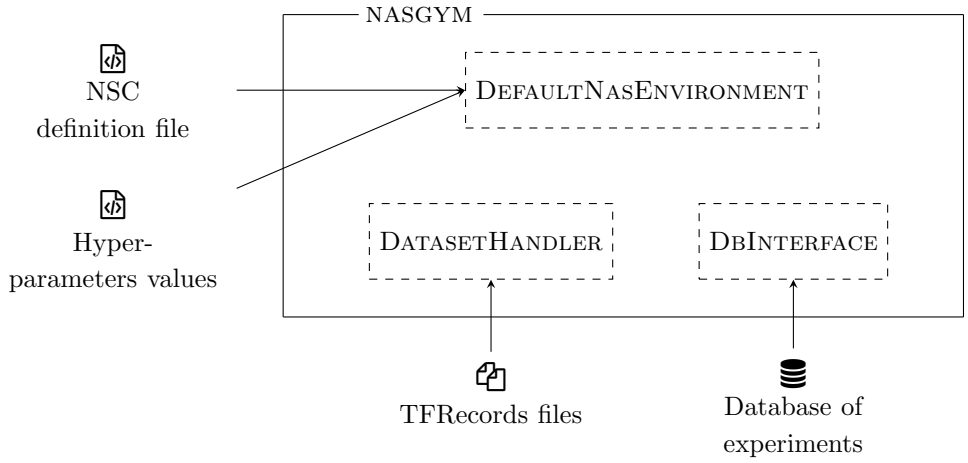


Figure 8: An sketch of the system built to perform our research. The nasgym package contains a default NAS environment whose states and actions are designed according to the Neural Structure Code (NCS) space defined in a *.yml* file. The hyperparameters for all the machine learning components are defined in a *.ini* file. Internally, the environment makes use of a dataset handler that reads TFRecords files and sends them as inputs to the neural architectures. A simple database of experiments is used to store experiments in a local file, although the logic can be easily be extended to support a more robust database system.

Additionally to the nasgym, we implement the meta version of the A2C algorithm on top of the OpenAI baselines[9]. We believe that this software engineering effort will help to compare, reproduce, and develop future research in NAS.

6. Experiments

To evaluate our framework, we conduct three experiments. The first two aim to study the behavior of the agent when challenged to design chain-structured networks, and the third one is intended to observe its behavior in the multi-branch setting. We empirically assess the quality of the networks designed by the agent through episodes, the ability of the agent to adapt to each environment, and the runtimes of the training trials.

6.1 Chain-structured networks

Experiment 1: evolution during training. The agent learns from the three train environments listed in Table 3, using deep meta-RL. It starts in the *omniglot* environment, continues in *vgg_flower*, and finishes in *dtd* so that it faces increasingly harder classification

9. Source code available at: github.com/gomerudo/openai-baselines

tasks (see Appendix A), and the policy learned in one environment is reused in the next one. The agent interacts with each environment for $t_{max} = 8000$, $t_{max} = 7000$, and $t_{max} = 7000$, respectively so that the agent spends more time in the first environment to develop its initial knowledge. We compare against two baselines: random search and DeepQN with experience replay, where the agent learns a new policy on each environment (i.e., it does not re-use the policy between trials) for $t_{max} = 6500, 5500,$ and 7000, respectively. Due to resources and time constraints, all t_{max} values were empirically selected according to the behaviour of the rewards (see Section 7). The most relevant hyper-paremeters are set as follows:

- Environment
  - $d = 10$. The maximum depth of a neural architecture.
  - $\tau = 10$. The maximum length of an episode.
- A2C hyper-parameters
  - $j = 5$. The number of steps to perform before updating the A2C parameters (see Equation 4.2.5). We set the value to half the maximum depth of the networks to allow the agent to learn before the termination of an episode.
  - $\gamma = 0.9$. The discount factor for the past actions.
  - $\eta = 0.01$. The default in the OpenAI baselines (Dhariwal et al., 2017).
  - $\alpha = 0.001$. The A2C learning rate set as in *Learning to reinforcement learn* (Wang et al., 2016).
- DeepQN
  - Experience buffer size = $\frac{t_{max}}{2}$.
  - Target model's batch size = 20.
  - ϵ with linear decay from 1.0 to 0.1. The parameter controlling the exploration of the agent.
  - $\alpha = 0.0005$. The default learning rate in the OpenAI baselines (Dhariwal et al., 2017).
- Training of the sampled networks
  - batch size = 128.
  - epochs = 12. The value used in BlockQNN (Zhong et al., 2018).

Experiment 2: evaluation of the policy. We fix the policy obtained in Experiment 1. The agent interacts with the evaluation environments, *aircraft* and *cu_birds*, and deploys its decision-making strategy to design a neural architecture for each dataset. The interaction runs for $t_{max} = 2000$ to study the performance of the policy in short evaluation trials. At the end of the interaction, we select the best two architectures per environment (i.e., the ones with the highest reward) and train them on the same datasets but applying a more intensive procedure as follows. First, we augment the capacity of the architectures by changing the number of filters in the convolution layers according to the layer's depth; i.e., number of units = 2^{4+i} with i being the current count of convolutions while building the network (e.g., number of units = $32 \rightarrow 64 \rightarrow 128$). Second, we stack the prediction module as described in Section 4.1.2, but we increase the number of units in the first dense layer

to 4096, we use a learning rate with exponential decay, and we train the network for 100 epochs. Since the datasets that we use are resized to a shape of 84x84x3, it is not fair to compare our resulting accuracy values with those of state-of-the-art architectures that assume a higher order of shape (Cui et al., 2018; Guo and Farrell, 2018; Hu and Qi, 2019), and neither is to train our networks (which are designed for a given input size) with bigger images. Hence, based on the baselines of Hu and Qi (2019), we use a VGG-19 network (Liu and Deng, 2015) with only two blocks as our baseline on both datasets.

6.2 Multi-branch networks

Experiment 3: training on a more complex environment. In this experiment, we extend the search space to multi-branch architectures. We consider the *omniglot* environment only. The goal here is to observe the ability of the agent to design multi-branch networks through time; i.e., the number of multi-branch structures generated through training. The interaction runs for $t_{max} = 20000$ time-steps because more exploration is required due to the larger action space. The hyper-parameters are the same as in Experiment 1, except that $\tau = 20$ and $j = 10$ because the trajectories are longer due to the shifting of the pointers controlling the predecessors, and batch size = 64 because the concatenation operation can generate networks that require more space in memory. We train the agent from scratch two times varying the parameter $\sigma \in [0.0, 0.1]$ (see Section 4.2.3) to study its effect encouraging the generation of multi-branch structures.

7. Results

Experiment 1: evolution during training

Figure 9 shows the evolution of the *best reward* and the *accumulated reward* per episode (representing the quality of the neural architectures), as well as the *episode length* (in a chain-structured network this represents the number of layers). We observe that, in the first environment (*omniglot*), our deep meta-RL agent performs worse than DeepQN. Nevertheless, in the second and third environments (*vgg_flower*, *dtd*), the agent performs better than the baselines from the very first steps, and more consistently through all episodes (showing less variance). DeepQN only catches up after many more episodes, although it exhibits a faster learning curve, which we ascribe to the linear exploration that makes it to end exploration sooner.

Figure 10 shows the entropy of the policy during training over the different environments, which in the A2C algorithm is related to the level of exploration by the agent (more exploration leads to high entropy). In the first environment, our agent explores the environment for a significant number of time-steps, which translates to the slow increase observed in Figure 9a. In the second environment, the exploration drops down quickly, except for a short period with increased exploration (time-steps 9005 to 12005). In the last and hardest environment, the agent re-explores the environment to adapt its strategy, leading to a reduction of the episode length (depth of the networks) and, consequently, the accumulated reward does not appear to increase due to the shorter episodes. We believe that exploration causes the drops in episode length in *vgg_flower* and *dtd* (Figures 9e and 9h).

In Figure 11, the proportion of the actions performed by the agent during training is shown. We see that it deployed different strategies per environment. Specifically, we note the changes in proportion for actions A0 (*convolution* with a kernel of size 1), A3 (*max-pooling* with pool size of 2), and A7 (the *terminal* state) when the environment switched from *vgg_flower* to *dtd*, suggesting that the agent preferred different layers and depth according to the dataset.

Finally, Table 4 shows the running times per environment for each RL algorithm. Here, we do not observe significant differences considering that once transferred, the policy of the deep meta-RL agent designs deeper and more costly networks, as observed in Figures 9e and 9h.

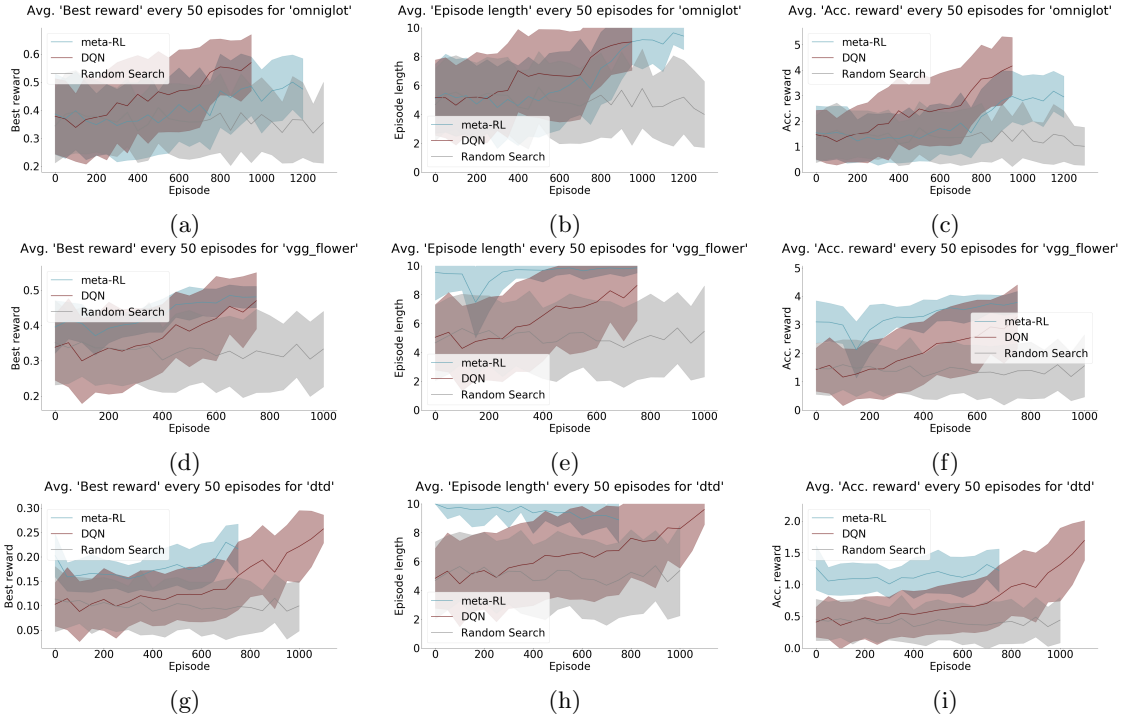


Figure 9: Evolution of training episodes through time from different perspectives, showing the means and ±1 standard deviations for every 50 episodes. Since the different techniques can build networks of different depths per episode, the number of episodes executed per environment may differ.

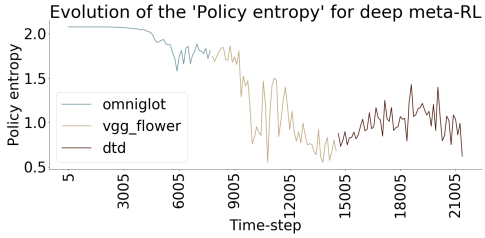


Figure 10: Policy entropy through environments in Experiment 1.

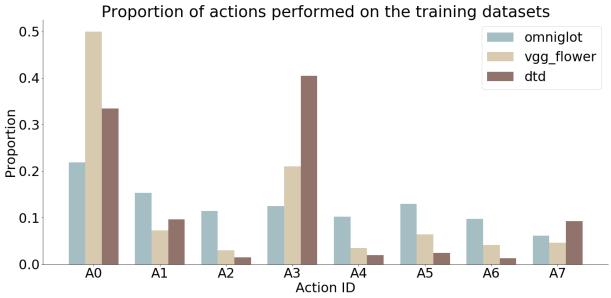


Figure 11: Proportion of actions performed by the agent per dataset in Experiment 1. The labels in the x-axis match the IDs in Table 2.

Dataset	Deep meta-RL	DQN
omniglot	11 days 9h	6 days 14h
vgg_flower	7 days 23h	5 days 15h
dtd	6 days 17h	6 days 4h
Total	26 days 1h	18 days 9h

Table 4: Running times per dataset during training. All experiments ran on a single NVIDIA Tesla K40m GPU.

Experiment 2: evaluation of the policy

The results of replaying the learned policy on completely new datasets are displayed in Figure 12, and the corresponding runtimes are listed in Table 5. They show that the agent immediately finds a good solution (with a deep network), and rewards remain consistent; however, it does not improve over time, which warrants further study (see Section 8). Moreover, the strategy deployed by the agent is not different on each dataset, as it is observed in Figure 13. We confirm that the strategies are not significantly different by performing a Wilcoxon signed-rank test with the null hypothesis that the two related paired samples come from the same distribution. The output is a p-value $= 0.48$ with 95% confidence.

Dataset	Runtime
aircraft	2 days 6h
cu_birds	2 days 22h
Total	5 days 4h

Table 5: Running times for the evaluation of the deep meta-RL agent. All experiments ran on a single NVIDIA Tesla K40m GPU.

As we mentioned in Section 6, another result of interest is the performance of the best networks designed by the agent when they follow more intensive training. Table 6 shows the accuracy values obtained. We note that the networks achieve low accuracy, in the majority of the cases worse than random guessing. An important observation is that these low values

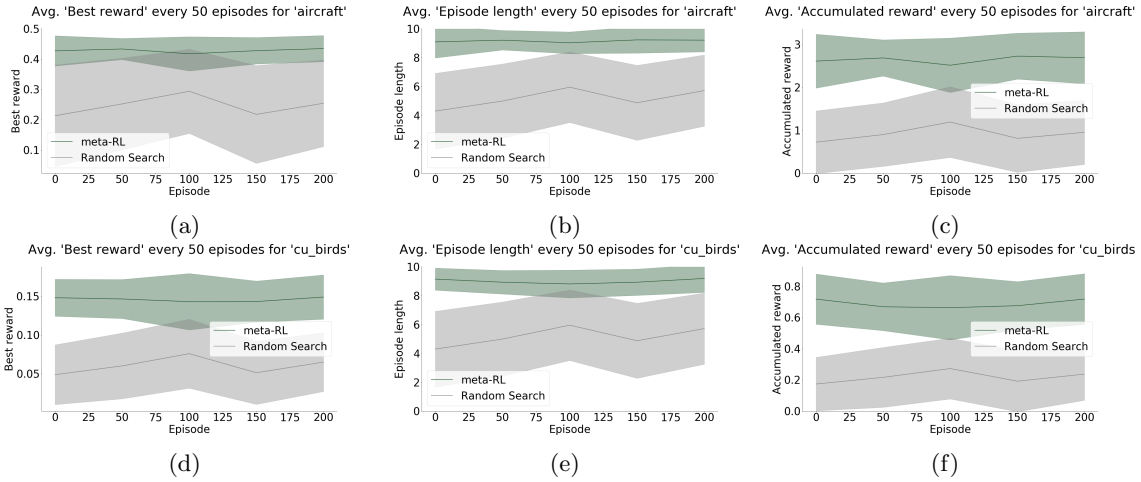


Figure 12: Evolution of evaluation episodes through time from different perspectives, showing the means and ± 1 standard deviations for every 50 episodes.

Figure 13: Proportion of actions per dataset during evaluation. The labels in the x-axis match the IDs in Table 2.

can be a consequence of the relaxation made in the shape of the images. Whereas state-of-the-art architectures on both *aircraft* and *cu_birds* work with shapes greater than 200×200, we use a smaller version of 84×84 that might lead to loss of information. Moreover, state-of-the-art results for these datasets are usually obtained after data augmentation and use deeper and more complex networks with multi-branch structures (Cui et al., 2018; Guo and Farrell, 2018; Hu and Qi, 2019). However, in this experiment, we do not consider any of the latter aspects since we work under resource constraints that force us to make relaxations, and thus a lower accuracy can be expected.

Despite the low values, the architectures for the two datasets designed by the deep meta-RL agent outperformed by a significant amount the shortened version of VGG19. This shows that by using the learned policy it is possible to find better architectures than one inspired by state-of-the-art networks. A final observation is that the best architecture found by the agent during training did not become the best final network, thus exhibiting that early-stop can underestimate the long-term performance of the networks, which also warrants future work.

Dataset	Deep meta-RL (1st)	Deep meta-RL (2nd)	VGG19-like
aircraft	49.18 ± 1.2	**50.11 ± 1.02**	30.85 ± 10.82
cu_birds	23.97 ± 1.28	**24.24 ± 0.90**	6.66 ± 1.98

Table 6: Accuracy values of the best architectures after a more intense training. Every reported accuracy value is the mean ± 2 standard deviations of five independent trainings. For the sake of completeness, we show the designed networks in Appendix B

Experiment 3: training on a more complex environment

Figure 14 shows the evolution of the *best reward*, *episode length*, and *accumulated reward* during the multi-branch experiment on *omniglot*. We do not observe differences in the behavior of the agent when using different σ values, but we note that it took longer to output meaningful rewards (around episode 3000) when compared to Experiment 1, causing extended runtimes as shown in Table 7.

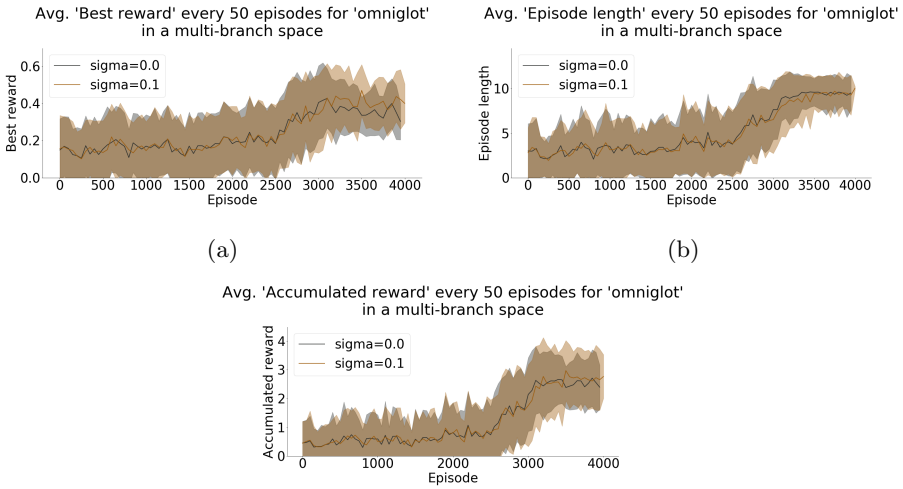


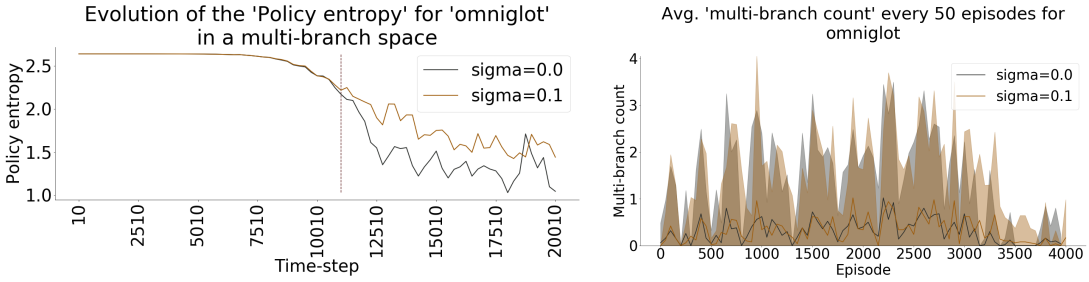
Figure 14: Evolution of training episodes for the multi-branch experiment from different perspectives. The plots show the means and ±1 standard deviations for every 50 episodes.

	Runtime
$\sigma = 0.0$	13 days 9h
$\sigma = 0.1$	15 days 14h
Total	28 days 23h

Table 7: Runtimes for the training of the deep meta-RL agent in the multi-branch search space for *omniglot*. All experiments ran on a single NVIDIA Tesla K40m GPU.

However, we stated in Section 6.2 that our main interest in this experiment is to study whether or not the agent can explore multi-branch structures. Figure 15a shows the en-

tropy of the policy through time-steps, and Figure 15b the count of multi-branch structures through episodes. We note that during exploration the appearance of multi-branch structures is more likely, and after episode 3000 (represented by the vertical line in Figure 15a), when exploration drops down, the multi-branch structures become less frequent. Furthermore, we found that the proportion of multi-branch vs. chain-structured networks is only 1:10, meaning that the agent did not explore multi-branch structures aggressively, and settled for chain-structured networks instead. The latter is supported by the proportion of actions displayed in Figure 16, where the actions A8-13 (related to multi-branch structures) are the least frequent.



(a) (b)

Figure 15: The exploration of the agent through time. (a) The entropy of the policy through time-steps. The vertical line cuts the horizontal axis at the time-step where the episde 3000 starts. (b) The count of multi-branch structures explored by the agent, showing the mean $\pm$ 1 standard deviation every 50 episodes.

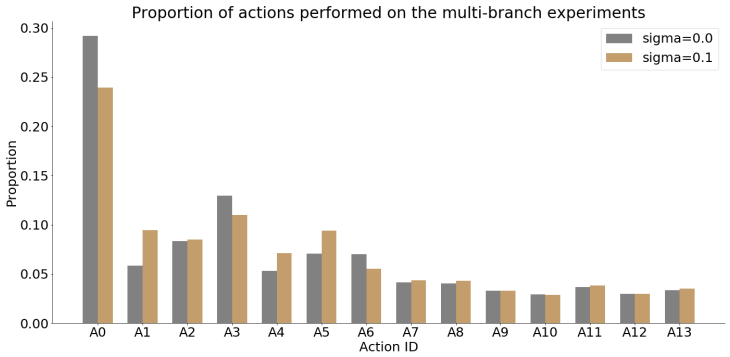


Figure 16: Proportion of actions taken by the agent in the multi-branch experiments. The labels in the x-axis match the IDs in Table 2.

We believe that a multi-branch space requires us to handle differently how the predecessors of the NSC vectors are selected (see Section 4.2.2). Some alternatives are: defining heuristics to chose the predecessors instead of using the shifting operations, assigning other rewards to the actions related to the predecessors, and modifying the hyper-parameters of the A2C network to encourage more exploration of the agent in the beginning of the deep meta-RL training.

8. Conclusions and future work

In this work, we presented the first application of deep meta-RL in the NAS setting. Firstly, we investigated the advantages of deep meta-RL against standard RL on the relatively simple scenario of chain-structured architectures. Despite resource limitations (1 GPU only), we observed that a policy learned using deep meta-RL can be transferred to other environments and quickly designs architectures with higher and more consistent accuracy than standard RL. Nevertheless, standard RL outperforms meta-RL when both learn a policy from scratch. We also note that the meta-RL agent exhibited adaptive behavior during the training, changing its strategy according to the dataset in question. Secondly, we analyzed the adaptability of the agent during evaluation (i.e., when the policy's weights are fixed) and the quality of the networks that it designs for previously unseen datasets. In our experiment, the agent was not able to adapt its strategy to different environments, but the performance of the networks delivered was better than the performance of a human-designed network, showing that the knowledge developed by our agent in the training environments is meaningful in others. Thirdly, we extended our approach to a more complex NAS scenario with a multi-branch search space. In this setting, the meta-RL agent was not able to deeply explore the multi-branch structures and settled for chain-structured networks instead.

We conclude that deep meta-RL does provide an advantage over standard RL when transferring is enabled, and it can effectively adapt its strategy to different environments during training. Moreover, the policy learned can be used to deliver meaningful and well-performing architectures on previously unseen environments without further training. We believe that it is possible to strengthen our deep-meta RL framework in future work. Specifically, we propose to investigate the following aspects under more powerful computational resources:

- *Hyper-parameter tunning of the A2C components.* In Experiment 1, we observed that the learning progress of the meta-RL agent is slow. We also noticed a long exploration window in the first environment. In order to improve these aspects, we propose to tune the hyper-parameters according to the next intuitions:
    - j: the parameter controlling the number of steps before a learning update. We suggest reducing this value to speed up learning.
    - η: the entropy regularization. Experiments varying the range of this hyper-parameter are required to observe its impact on the learning curve. Also, different values could be used depending on the hardness of the environments.
    - α: the learning rate. We suggest exploring decay functions for the learning rate to encourage faster learning after exploration.
- *Duration of the agent-environment interaction.* In Experiment 2, the policy did not exhibit adaptive behavior. A possible explanation is that the training trials were relatively short when compared to other reinforcement learning applications. Training the agent for longer trials could help improve the adaptation of the policy during evaluation.
- *The action space in the multi-branch setting.* In Experiment 3, we observed that the agent was not able to explore the multi-branch space sufficiently and settled for

chain-structured networks instead. Although hyper-parameter tuning could also help encourage exploration of the multi-branching actions, we believe that redefining the actions is a more suitable area of improvement. In that line, we recommend exploring heuristics based on the number of connections to select the predecessors.

- *The datasets and the performance estimation strategy.* In all experiments, we observed a low accuracy of the networks in the datasets. Since we worked with constrained resources, we applied relaxations to the datasets and the performance estimation strategy to reduce the computational cost, which could have affected the accuracy of the networks. Future work can focus on designing a different set of environments with images with a smaller size, optimizing the performance estimation strategy per dataset, and investigating alternatives to reduce the cost of computing the rewards associated to the networks.
- *Transforming other standard RL algorithms to a meta-RL version.* The transformation of the A2C algorithm to a meta-RL implementation required to change the input passed to the policy and to rely on a recurrent unit to learn the temporal dependencies between actions. This transformation is possible on other standard RL algorithms, which would help study different meta-RL approaches to NAS.
- *Benchmarking of other RL on the same NAS environments.* In Section 5, we introduced the system developed to conduct our experiments, which allows to easily play other RL algorithms from the OpenAI baselines on the same NAS environments. We believe that this system will help to encourage research in these directions so that the benefits of different RL algorithms on NAS can be studied in detail.

Acknowledgments

The authors thank the *SURF* cooperative for kindly providing the required computational resources.

References

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `http://tensorflow.org/`. Software available from tensorflow.org.

I. Arel, C. Liu, T. Urbanik, and A. G. Kohls. Reinforcement learning-based multi-agent system for network traffic signal control. *IET Intelligent Transport Systems*, 4(2):128–135, June 2010. ISSN 1751-956X. doi: 10.1049/iet-its.2009.0070.

Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. *CoRR*, abs/1611.02167, 2016. URL `http://arxiv.org/abs/1611.02167`.

B. Bouzy and G. Chaslot. Monte-carlo go reinforcement learning experiments. In *2006 IEEE Symposium on Computational Intelligence and Games*, pages 187–194, May 2006. doi: 10.1109/CIG.2006.311699.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

Han Cai, Jiacheng Yang, Weinan Zhang, Song Han, and Yong Yu. Path-level network transformation for efficient architecture search. *CoRR*, abs/1806.02639, 2018. URL `http://arxiv.org/abs/1806.02639`.

Yin Cui, Yang Song, Chen Sun, Andrew Howard, and Serge J. Belongie. Large scale fine-grained categorization and domain-specific transfer learning. *CoRR*, abs/1806.06193, 2018. URL `http://arxiv.org/abs/1806.06193`.

Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. Openai baselines. `https://github.com/openai/baselines`, 2017.

Yan Duan, John Schulman, Xi Chen, Peter L. Bartlett, Ilya Sutskever, and Pieter Abbeel. Rl^2: Fast reinforcement learning via slow reinforcement learning. *CoRR*, abs/1611.02779, 2016. URL `http://arxiv.org/abs/1611.02779`.

Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural Architecture Search: A Survey. *arXiv e-prints*, art. arXiv:1808.05377, Aug 2018.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. *CoRR*, abs/1703.03400, 2017. URL `http://arxiv.org/abs/1703.03400`.

Pei Guo and Ryan Farrell. Fine-grained visual categorization using PAIRS: pose and appearance integration for recognizing subcategories. *CoRR*, abs/1801.09057, 2018. URL `http://arxiv.org/abs/1801.09057`.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. *CoRR*, abs/1603.05027, 2016. URL `http://arxiv.org/abs/1603.05027`.

Tao Hu and Honggang Qi. See better before looking closer: Weakly supervised data augmentation network for fine-grained visual classification. *CoRR*, abs/1901.09891, 2019. URL `http://arxiv.org/abs/1901.09891`.

Diederik P Kingma and Jimmy Lei Ba. Adam: A method for stochastic gradient descent. *ICLR: International Conference on Learning Representations*, 2015.

Jens Kober, J. Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *Int. J. Rob. Res.*, 32(11):1238–1274, September 2013. ISSN 0278-3649. doi: 10.1177/0278364913495721. URL http://dx.doi.org/10.1177/0278364913495721.

Vijaymohan Konda. *Actor-critic Algorithms*. PhD thesis, Cambridge, MA, USA, 2002. AAI0804543.

Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: differentiable architecture search. *CoRR*, abs/1806.09055, 2018. URL http://arxiv.org/abs/1806.09055.

S. Liu and W. Deng. Very deep convolutional neural network based image classification using small training sample size. In *2015 3rd IAPR Asian Conference on Pattern Recognition (ACPR)*, pages 730–734, Nov 2015. doi: 10.1109/ACPR.2015.7486599.

Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, HotNets '16, pages 50–56, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4661-0. doi: 10.1145/3005745.3005750. URL http://doi.acm.org/10.1145/3005745.3005750.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL http://arxiv.org/abs/1312.5602.

Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016. URL http://arxiv.org/abs/1602.01783.

Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. *CoRR*, abs/1802.03268, 2018. URL http://arxiv.org/abs/1802.03268.

Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. Regularized evolution for image classifier architecture search. *CoRR*, abs/1802.01548, 2018. URL http://arxiv.org/abs/1802.01548.

John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015. URL http://arxiv.org/abs/1502.05477.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL http://arxiv.org/abs/1707.06347.

Prabhant Singh, Tobias Jacobs, Sebastien Nicolas, and Mischa Schmidt. A Study of the Learning Progress in Neural Architecture Search Techniques. *arXiv e-prints*, art. arXiv:1906.07590, Jun 2019.

Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: an introduction*. The MIT Press, 2012.

Eleni Triantafillou, Tyler Zhu, Vincent Dumoulin, Pascal Lamblin, Kelvin Xu, Ross Goroshin, Carles Gelada, Kevin Swersky, Pierre-Antoine Manzagol, and Hugo Larochelle. Meta-dataset: A dataset of datasets for learning to learn from few examples. *CoRR*, abs/1903.03096, 2019. URL http://arxiv.org/abs/1903.03096.

Jane X. Wang, Zeb Kurth-Nelson, Dhruva Tirumala, Hubert Soyer, Joel Z. Leibo, Rémi Munos, Charles Blundell, Dharshan Kumaran, and Matthew Botvinick. Learning to reinforcement learn. *CoRR*, abs/1611.05763, 2016. URL http://arxiv.org/abs/1611.05763.

Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1992. ISSN 1573-0565. doi: 10.1007/BF00992698. URL https://doi.org/10.1007/BF00992698.

Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. In *Machine Learning*, pages 229–256, 1992.

Zhao Zhong, Zichen Yang, Boyang Deng, Junjie Yan, Wei Wu, Jing Shao, and Cheng-Lin Liu. Blockqnn: Efficient block-wise neural network architecture generation. *CoRR*, abs/1808.05584, 2018. URL http://arxiv.org/abs/1808.05584.

Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. *CoRR*, abs/1611.01578, 2016. URL http://arxiv.org/abs/1611.01578.

Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. *CoRR*, abs/1707.07012, 2017. URL http://arxiv.org/abs/1707.07012.

Appendix A. Selection of the datasets

The deep meta-reinforcement learning framework that we implement requires a set of environments associated to image classification tasks. In order to design these environments, we rely on the meta-dataset (Triantafillou et al., 2019), a collection of 10 datasets with a concrete sampling procedure designed for meta-learning in few-shot learning image classification. In our setting, the datasets are intended for standard image classification, thus we redefine the sampling strategy. Our interest is in using small but yet challenging datasets that allow us to save computational resources without making the Neural Architecture Search (NAS) trivial.

In Table 8 the original datasets in the collection are listed. We select the ones that are smaller than CIFAR-10 (60K observations), which is the reference for NAS. The datasets satisfying the criterion are *aircraft*, *cu_birds*, *dtd*, *omniglot*, *traffic_sign* and *vgg_flower*. We want to evaluate the hardness of these six datasets to define a sampling procedure from the collection, and thus we perform a short and individual deep meta-reinforcement learning trial with $t_{max} = 200$ for each dataset. Since at the beginning of the trial the agent does not develop any significant knowledge, its sampling of architectures is random. In Figure 17 the boxplot and barplot of the obtained accuracy values are presented, and in Table 9 the running time per experiment is shown.

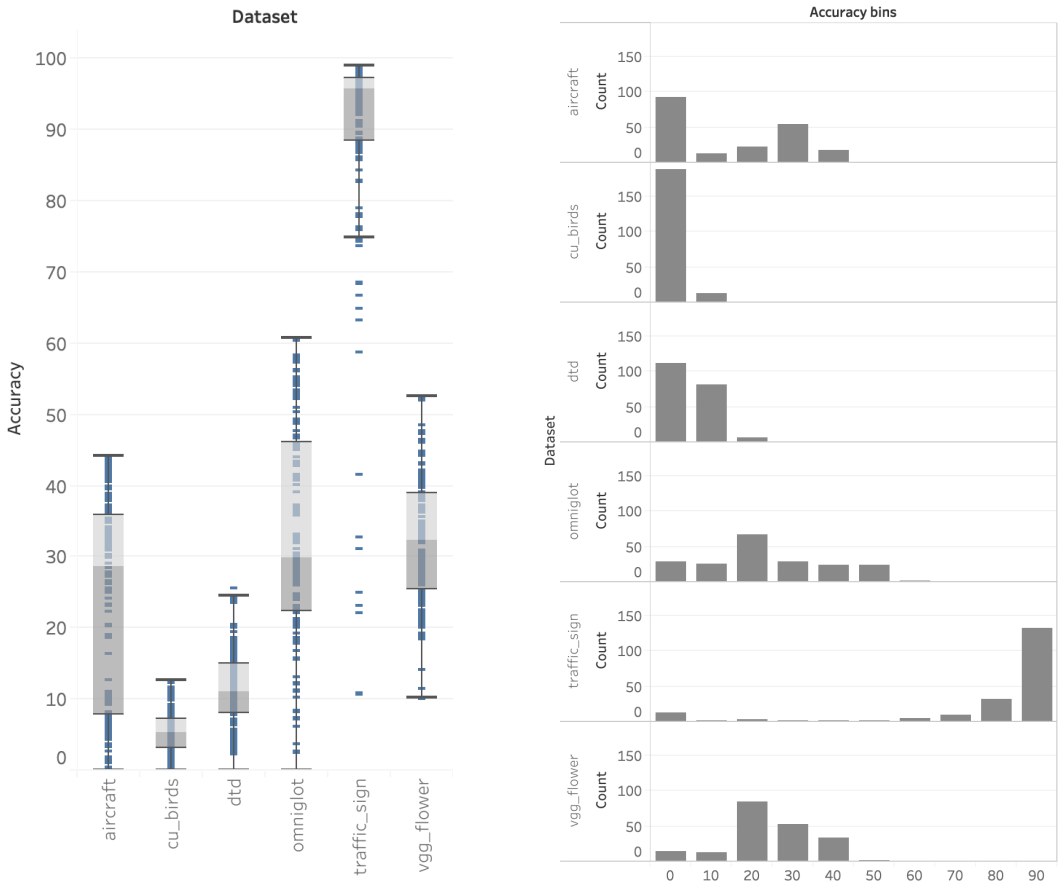
A simple exploratory analysis suggests three types of datasets: a "trivial" dataset with high accuracy values with simple networks (*traffic_sign*), two "hard" datasets with low accuracy values (all values below 30%: *dtd* and *cu_birds*), and three "medium" datasets with more diversity of accuracy values (median around 30% and broader interquartile range: *aircraft*, *omniglot*, *vgg_flower*). On the other hand, for the running times, we can observe that *aircraft* and *cu_birds* result in the most expensive runs. Considering the computation time, and the hardness of the classification tasks, we defined the sampling presented in Table 3. Our training datasets have different levels of hardness and reported the least costly runs.

Dataset ID	Dataset name	N classes	N observations
aircraft	FGVC-Aircraft	100	10000
cu_birds	CUB-200-2011	200	11788
dtd	Describable Textures	47	5640
fungi	FGVCx Fungi	1394	89760
ilsvrc_2012	ImageNet	1000	1280764
mscoco	Common Objects in Context	80	330000
omniglot	Omniglot	1623	32460
quickdraw	Quick, Draw!	345	50426266
traffic_sign	German Traffic Sign Recognition Benchmark	43	39209
vgg_flower	VGG Flower	102	8189

Table 8: The original meta-dataset (Triantafillou et al., 2019) with the number of classes and observations after conversion with the official source code.

Dataset ID	Time
aircraft	9h49m
cu_birds	16h20m
dtd	5h38m
omniglot	3h38m
traffic_sign	4h33m
vgg_flower	4h56m

Table 9: Running times of a deep meta-RL trial with $t_{max} = 200$, used to study the hardness and cost of each dataset.



(a) (b)

Figure 17: Different visualizations of the early-stop accuracy values obtained to study the hardness of the datasets.

Appendix B. Networks designed by the deep meta-RL agent during training and evaluation

Here we show the best architectures designed by the agent in the three experiments. Figure 18 shows the best architecture per datasets during training (*omniglot*, *vgg_flower*, and

dtd). Figure 19 and 20 show the best two architectures during evaluation for *aircraft* and *cu_birds* respectively. Figure 21 shows the best architectures for the multi-branch experiment. For each architecture we report the early-stop accuracy obtained.

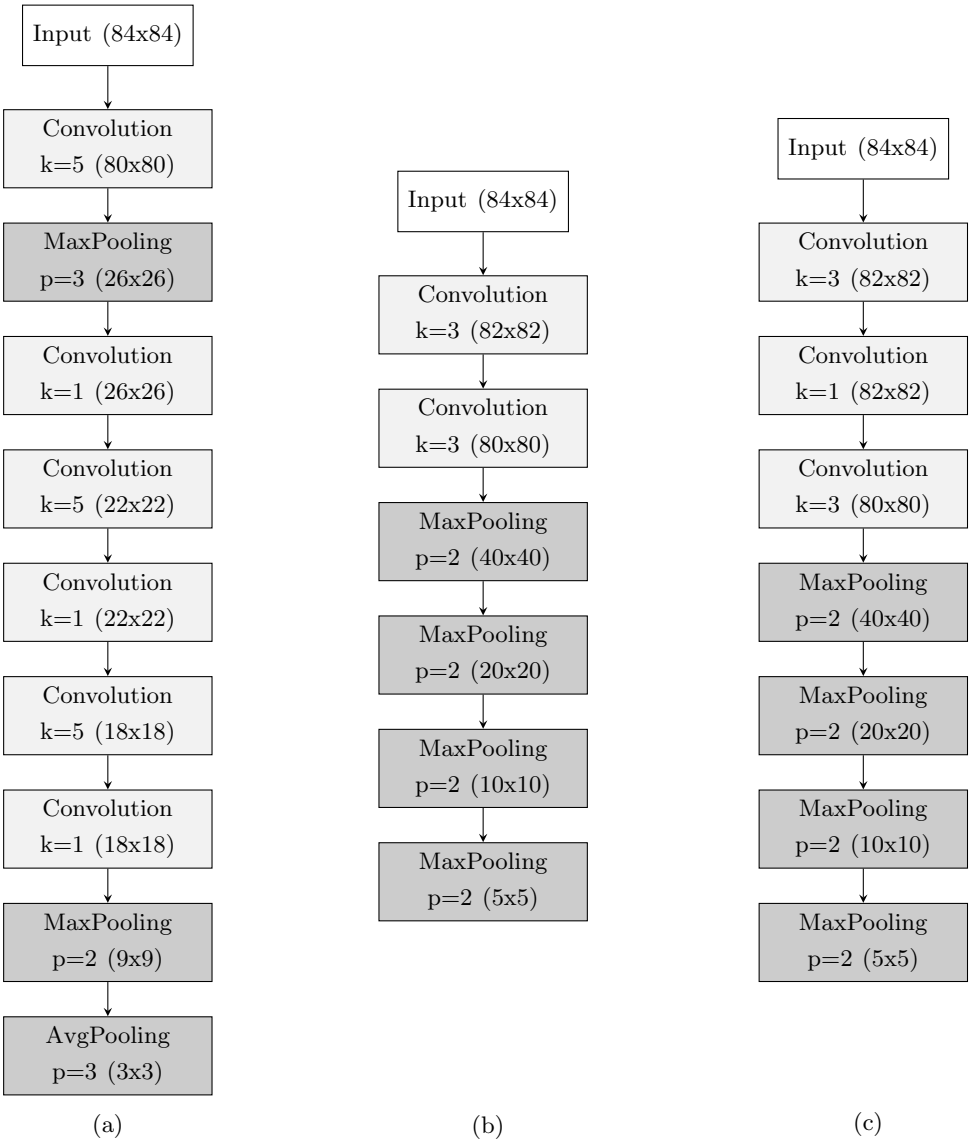


Figure 18: Best architectures designed for the training datasets. (a) The best architecture for *omniglot*, with early-stop accuracy of 67.11. (b) The best architecture for *vgg_flower*, with early-stop accuracy of 57.15. (c) The best architecture for *dtd*, with early-stop accuracy of 29.43

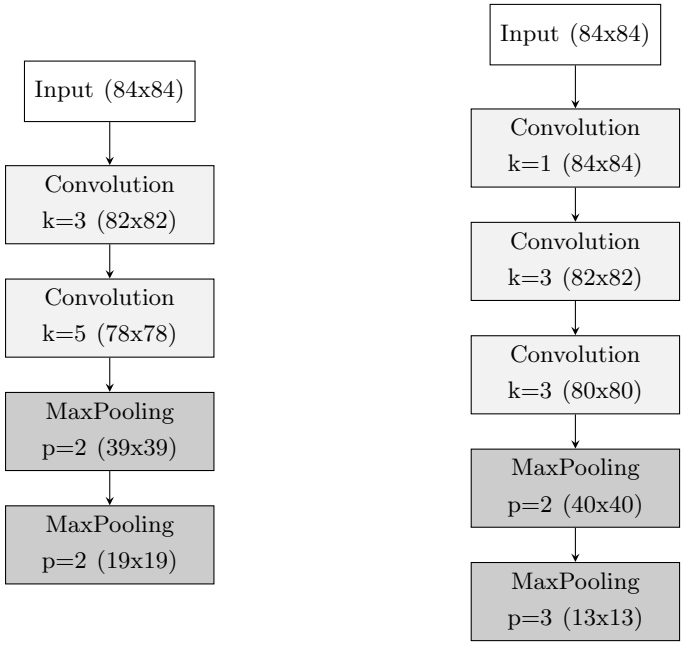


Figure 19: Best architectures designed for *aircraft* during evaluation of the policy. (a) The best architecture with early-stop accuracy of 48.22. (b) The second-best architecture with early-stop accuracy of 47.95

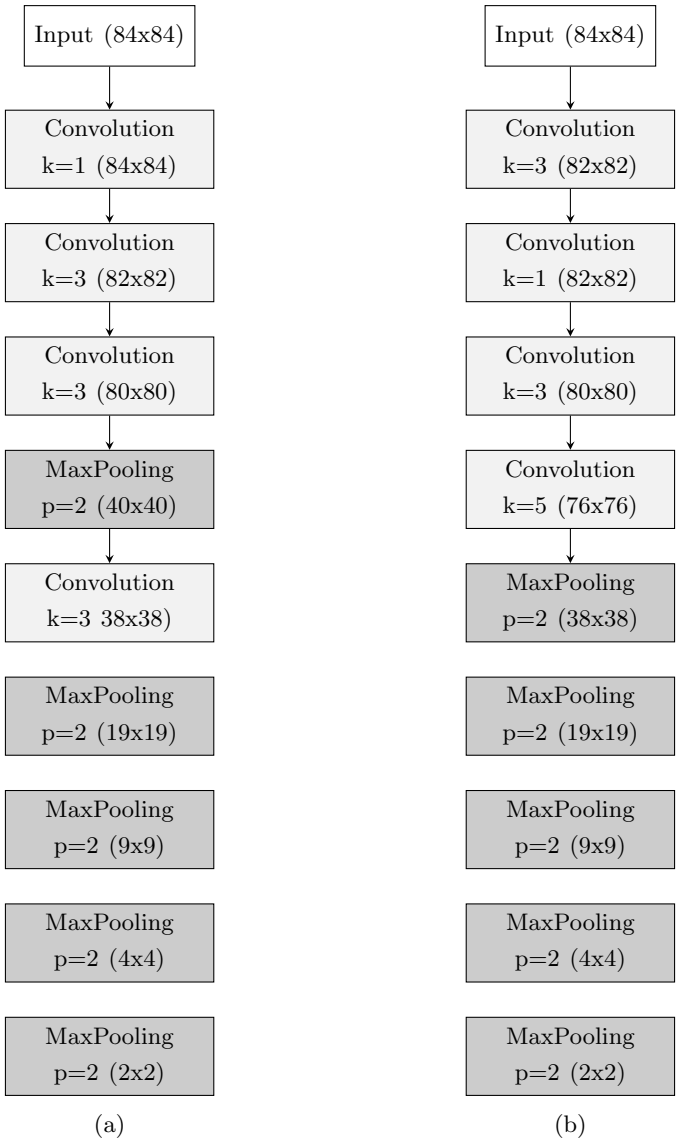


Figure 20: Best architectures designed for *cu_birds* during evaluation of the policy. (a) The best architecture with early-stop accuracy of 19.22. (b) The second-best architecture with early-stop accuracy of 19.06

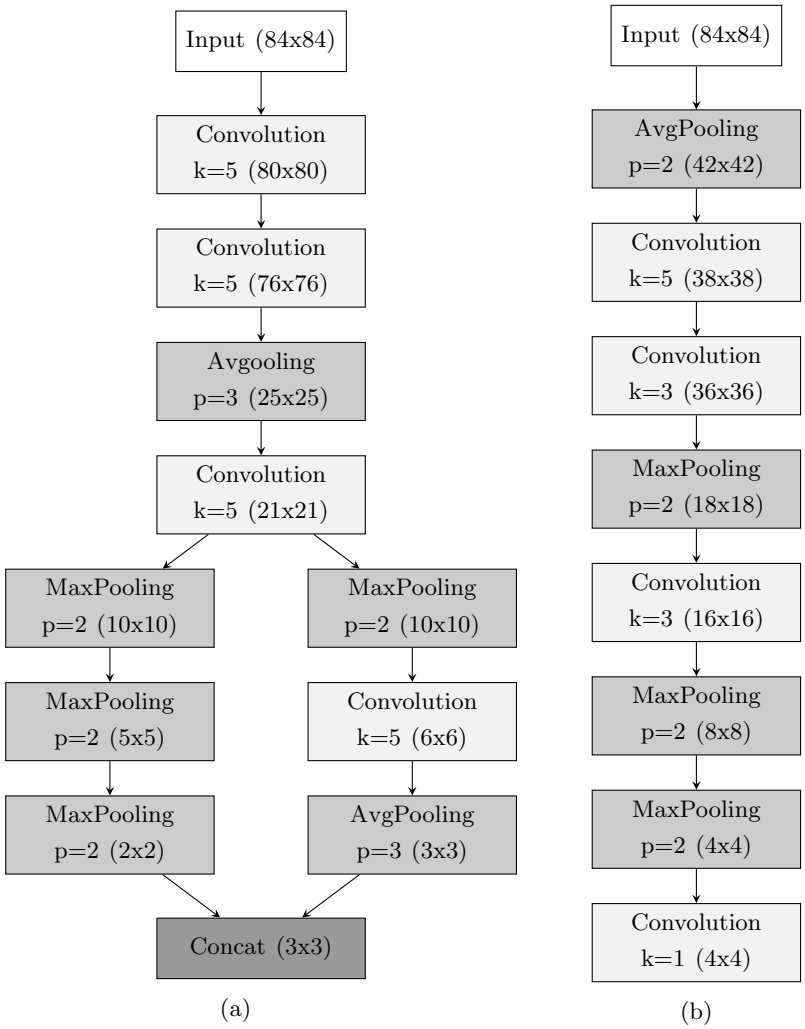


Figure 21: Best architectures designed for during the experiment in a multi-branch search space. (a) The best architecture when $\sigma = 0.0$, with early-stop accuracy of 66.10. (b) The best architecture when $\sigma = 0.1$, with early-stop accuracy of 66.45