Neural Architecture Search: An overview

Federico Minutoli
DIBRIS
University of Genoa
fede97.minutoli@gmail.com

Massimiliano Ciranni
DIBRIS
University of Genoa
massimiliano.ciranni@outlook.com

Abstract

Recent advances in deep learning research have given birth to really powerful architectures, which have been used with great success in a wide range of fields. Despite designing neural architectures has become easier, a lot of human expertise is still required to find optimal solutions and often comes as the result of extended trial and error cycles. This has led to a rapid increase in the complexity of architectures (e.g., ResNet with its many skip connections) that has left a large segment of the machine learning community wondering "*How can someone come out with such an architecture?*" (J. Langford referring to *DenseNet*) [1]. This is a hot-topic in machine learning right-now, which is being addressed by neural architecture search (NAS), a novel branch of automated machine learning that aims at automating architecture engineering. The core idea behind a NAS system is to feed it with a dataset and a task (classification, regression, etc.) as input, expecting it to find an architecture by navigating through a search space of all possible architectures trying to maximize a reward-based search strategy. In this paper we will provide an overview of the many questions that NAS tries to answer, also addressing some of the main limitations that plague it at the moment (i.e., GPU time, combinatorial explosion, domain shift, etc.). Then we will provide a brief outline of the main approaches followed by a more in depth analysis on two of them: differentiable search spaces (DNAS) and (efficient) reinforcement learning (ENAS). In the end, we will present a naive toy implementation of a controller-based NAS system tailored at sampling multi-layer perceptrons (MLPs) on a handful of datasets, against Scikit-learn's pre-made MLP classifier.

1 Introduction

The success that deep learning gained in the last few years is mostly due to its capability of automating the feature engineering process, allowing to extract hierarchical features in an end-to-end fashion, rather than manually design features from data. New opportunities made available by deep learning methods imply that there is a high demand of deep learning expertise and architecture engineering, as we see models becoming more and more complex, while still manually designed. Deep learning engineers are expected to have an intuitive understanding of what architecture might work best for what situation, but this is rarely the case. DNNs are complex systems, and the number of architectural variants one could come up with are literally infinite; consider for example a famous image recognition model, such as ResNet [2]: it comes with many flavours in terms of the number of hidden layers (50, 101, 152) and configurations for its skip-connections. It is obvious that there are many ways in which a model like this could be tweaked just by adding/removing convolutional blocks or skip-connections and, in general, when we look at the best performing models, it is almost never clear why certain design choices have been made in that specific manner. A logical step to avoid the process of manual designing neural networks is to automate it by building a system that given a dataset provides a neural network architecture that can be implemented, trained, evaluated and eventually serve its purpose; this is something that, in comparison, has been already addressed and

Advanced Machine Learning (10184) - (University of Genoa, 2020) - Genoa, Italy.

solved for classical machine learning algorithms (under the name of hyper-parameter optimization [3]), by means of grid search, random search, *Bayesian optimization*, meta-learning, to name a few, but when considering deep learning architectures, the problem becomes much harder to deal with. The process of automating architecture engineering took the name of *Neural Architecture Search* (***NAS***), a branch of automated machine learning (AutoML [4]) which was first introduced by Zoph et al. (2016) [5] and quickly turned into an extremely popular area of research, with a seemingly exponential increase in the number of papers written on the subject, as Figure 1 shows. On top of that all the main manufacturers seem pretty active at the moment, with the aim of uncovering the full potential of such systems, like Google for large-scale object classification [6], Tesla for automatic configuration assessment in automotive [7] and Microsoft with its NNI's toolkit [8] which offers implementations for most of the current state-of-the-art approaches in NAS.

In the next sections you will find a comprehensive overview of the main components that compose a NAS system and its main limitations (Section 2), a detailed description of two algorithms, DARTS and ENAS, which have received a lof attention recently (Section 3), followed by our proposed naive implementation of a controller-based NAS system for multi-layer perceptrons (Section 4) and, finally, a few noticeable results that we could gather out of it (Section 5.1) and conclusions (Sections 6).

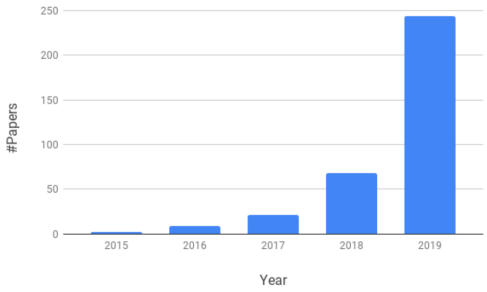

Figure 1: NAS papers per year based on the literature list on `automl.org` [2]

2 Problem setting analysis

A NAS system can be defined in terms of three main components (see Figure 2)

- *Search space*, shapes the type of valid networks that can be designed.
- *Search strategy*, defines the approach to navigate through the search space.
- *Performance estimation strategy*, looks for ways to evaluate the performance of a plausible network solely from its design without constructing or training it.

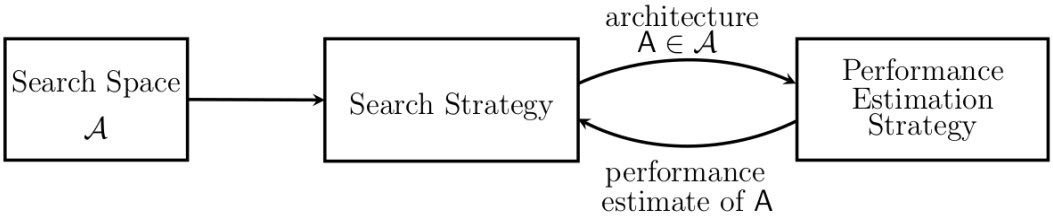

Figure 2: Scheme of a generic NAS system. [9]

2.1 Search space

The search space defines which neural architectures a NAS approach might discover in principle, that is, the domain $\mathcal{D}$ of valid architectures it could navigate towards (see an example in Figure 3). A relatively simple search space is the space of *chain-structured neural networks*, a sequence of N layers, where the i-th layer L_i receives its input from the (i-1)-th layer and sends it to the (i+1)-th through a function $g(L_{i-1}^{out})$. The search space is then parametrized by: (i) the (maximum) number of layers N (possibly unbounded); (ii) the type of operation every layer executes, e.g.,

[2]The literature list on NAS papers at `www.automl.org` is manually curated and lists all NAS paper that we are aware of starting from early 2017 up to summer 2020.

[3]Implied in Zoph et al. (2016) first NAS system [5]

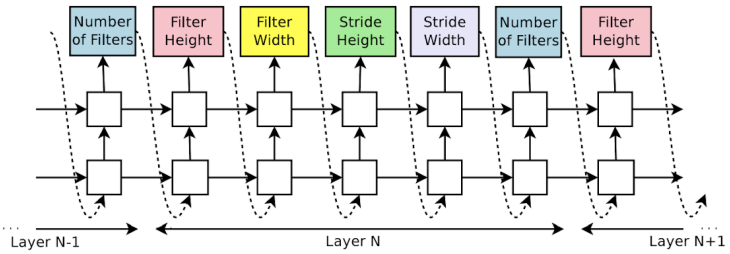

Figure 3: The search space for CNNs design [3]

pooling, convolution, or more advanced operations like depth-wise separable convolution; and (iii) hyper-parameters associated with the operation, e.g., number of filters, kernel size and strides for a convolutional layer or even number of units for fully-connected networks. Note that the parameters from (iii) are conditioned on (ii), hence the parametrization of the search space is not fixed-length, but a conditional space. Recent work on NAS (Real et al., 2019) incorporates modern design elements known from hand-crafted architectures, such as skip connections, which allow to build complex, *multi-branch networks* (i.e., residual networks, dense networks, etc.). In this case the input of layer i can be instead formally described as a function $g(L_{i-1}^{out}, ..., L_0^{out})$ combining previous layers outputs, in turn resulting in a function with significantly more degrees of freedom. The motif of these approaches is to predict an architecture without previous knowledge of what a good architecture should even look like to begin with, which opens up room to novel architectures and to a better interpretability of the current human-made ones; hence, they've been given the name of *macro-architecture* search NAS.

The biggest drawback of the approach mentioned above was the time it took to navigate through the search space before coming up with a definite solution. Zoph et al. (2016) had to use 800 GPUs for 28 days (with a cost of approx. 130k US$) to navigate through the entire search space before coming up with the best architecture. There was clearly a need for a way to design controllers that could navigate the search space more intelligently and, motivated by hand-crafted architectures consisting of repeated motifs, Zoph et al. (2018) proposed to search for such motifs, dubbed *cells*, rather than for whole architecture. A typical cell is modeled as a *Direct Acyclic Graph* (DAG) consisting of n nodes. Each node is a latent representation (like 2D kernels in CNNs) while directed edges connecting each node map to operations, such as pooling, or batch normalization.

Zoph et al. (2018) would rather optimize two different kind of cells: a normal cell that preserves the dimensionality of the input and a reduction cell which reduces the spatial dimension. The final architecture is then built by stacking these cells in a predefined manner, dictated by human bias. This approach has brought great benefits to NAS systems research, since the search space of cells consists of significantly less layers than whole architectures, with an estimated speed-up of a factor 7, coming from an order of magnitude less parameters. Also, architectures built from cells can more easily be transferred or adapted to other data sets by simply varying the number of cells and filters used within a model. Indeed, Zoph et al. (2018) transferred cells optimized on CIFAR-10 to ImageNet and achieved state-of-the-art performance, whereas domain shift, due to an intrinsic bias towards the current dataset, has always been a worry-some problem for macro-architectures NAS systems. A few questions rose, though, regarding how while using a cell-based, or *micro-architecture*, search space the macro-architecture would be chosen: how many cells shall be used and how should they be connected in the model? Ideally, both the macro-architecture and the micro-architecture should be optimized jointly instead of solely optimizing the micro-architecture. Until recently this proposition has been omitted, as given the undeniable advantages micro-architecture has shown over macro-architecture, a lot of the community has just shifted towards designing such systems, instead of trying to understand why they outperform macro-architecture ones on many different tasks. Indeed starting from [10], more cost-efficient macro-architecture NAS systems have been designed with the intention of bridging the gap between these two opposite flavours.

2.2 Search strategy

The search strategy details how to explore the search space (which often happens to be exponentially large or even unbounded). A good search strategy should be able to balance the trade-off between finding well-performing architectures in short time and premature convergence to a region of sub-

optimal architectures, whose importance is so keen, that most of the work gone into neural architecture search has been innovations for this part of the problem: finding out which optimization methods work best, and how they can be changed or tweaked to make the search process churn out better results faster and with consistent stability. There have been several approaches attempted to navigate the space of neural architectures, including random search, gradient-based methods, Bayesian optimization, reinforcement learning (RL) and evolutionary methods.

2.2.1 Reinforcement learning

Reinforcement learning is the science of optimal decision making. When an infant plays, waves its arms, it has no explicit teacher, but it does have a direct sensor motor connection to its environment. Exercising this connection produces a wealth of information about cause and effect, about consequences of actions, and about what to do in order to achieve goals. This is the key idea behind RL: given an environment which represents the outside world to the agent and an agent that takes actions, receives observations from the environment that consists of a reward for his action and information of his new state. That reward informs the agent of how good or bad was the taken action, and the observation tells him what is his next state in the environment. The agent tries to figure out the the best actions to take or the optimal way to behave in the environment in order to carry out his task in the best possible way, essentially functioning as an active learning paradigm. Defined as such, in RL there is no concept of supervision. Feedback from the environment might be delayed over several time steps, it's not necessarily instantaneous. This means that the data in RL cannot be consider as i.i.d since the agent might spend some time exploring and wandering in the environment and not see other parts which might be interesting to learn the optimal behavior. So time really matters, the agent must explore pretty much every part of the environment to be able to take the right actions. All of this is formally explained in terms of a policy π, regulated by a parameter θ, according to which the agent tries to find the optimal trajectory τ across the environment he is in by taking a sequence of actions $[a_1, ..., a_T]$ from its corresponding current state $[s_1, ..., s_T]$. Hence, the objective of RL is to maximize the policy gradient w.r.t. θ, according to the following gradient descent:

$$\theta_{t+1} = \theta_t + \nabla_\theta J(\theta)$$

where $J(\theta)$ can be expressed in terms of an expectation on future rewards, taking into account the sum of probabilities of the trajectory τ conditioned by θ and the current rewards at time t:

$$J(\theta) = \mathbb{E}[\sum_{t=1}^{T} R(a_t|s_t; \pi_\theta)] = \sum_{\tau} p(\tau; \theta) R(\tau)$$

which turns the objective function into finding the policy (parameter) θ that induces a trajectory $\tau = [(s_1, a_1), ..., (s_T, a_T)]$ that maximizes the expected rewards:

$$\theta^* = \arg\max_\theta J(\theta) = \arg\max_\theta \sum_{\tau} p(\tau; \theta) R(\tau)$$

Being such a powerful tool, not to mention the greatest success in AI history, that is DeepMind's AlphaGo ruthlessly managing to defeat Lee Sedol in 2016, the first NAS paper from Zoph et al. (2016) [5] implied the use a recurrent network to generate the model descriptions of neural networks and train this RNN with reinforcement learning to maximize the expected accuracy of the generated architectures on a validation set which rivaled human made state-of-the-art models on the CIFAR-10 dataset' test accuracy. Many RL-based NAS systems have been proposed since then, but the general idea to frame a NAS as reinforcement learning problem is to associate the generation of a neural architecture to be the agent's action, with the action space identical to the search space. The agent's reward is based on an estimate of the performance of the trained architecture on unseen data and the state can be seen as a summary of the actions sampled so far. Different RL approaches differ in how they represent the agent's policy and how they optimize it: Zoph and Le (2017) use a recurrent neural network (RNN) policy to sequentially sample a string that in turn encodes the neural architecture trained on the popular REINFORCE policy gradient algorithm (Williams, 1992). Often, the models built with the sole objective of a high validation accuracy end up being high in complexity–meaning a greater number of parameters, more memory required, and higher inference times. An important

contribution with to combat these issues was made by Hsu et al. (2018), where they proposed MONAS, which introduces a multi-task objective that not only tries to maximize validation accuracy, but also to minimize power consumption.

2.2.2 Progressive NAS

Progressive NAS is designed to address the task of finding the best network cell for a given task (micro-search). The first iteration of a progressive NAS generates all possible cells made of just one block and creates proxy CNNs formed by a single cell stacked for a predetermined number of times n. Validation accuracies from proxy CNNs and their respective cell structure are used to train a surrogate function implemented through a LSTM-based network, that will learn to predict the performance of a candidate cell structure without the need of building and training it. As iterations follow, blocks are incrementally added to the original 1-block cells in a greedy manner following a simple heuristic space. The performance predictor has the role of pruning out all those newly found structures that are not promising enough, keeping just the top K cells for proxy CNNs generation and predictor training. When the desired number of per-cell blocks, B, is reached, the best performing cell is stacked n times to form the final full CNN.

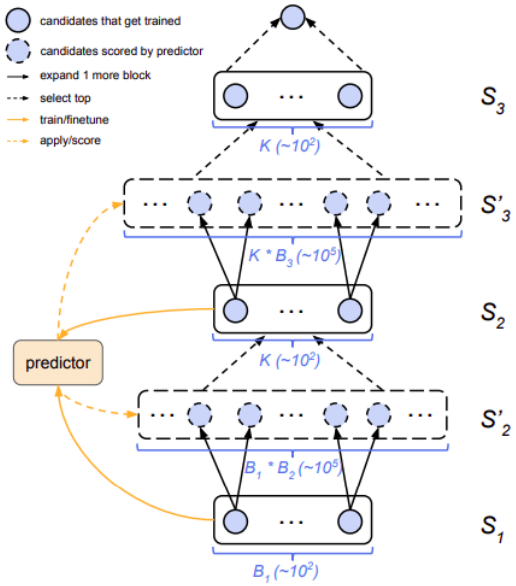

Figure 4: Scheme of the main PNAS steps. [9]

2.2.3 Differentiable NAS

Differentiable NAS is a small family of NAS systems which approach the problem of architecture search from a different angle. Instead of searching over a discrete set of candidate architectures, DNAS models leverage a continuous relaxation of the search space, so that the architecture can be optimized with respect to its validation set performance by gradient descent. The data efficiency of gradient-based optimization, as opposed to inefficient black-box search, allows differentiable methods to achieve competitive performance with state-of-the-art models using orders of magnitude less computation resources. For this reason, this family of NAS models is sometimes referred to as *super-network NAS*, as these models were implemented to challenge the main state-of-the-art networks performing complex vision tasks. The idea of searching architectures within a continuous domain was already explored in [11], [12], [13] and [14], but it was only with the introduction of DARTS [15] that this approach gained its current popularity.

2.2.4 Others

A strong alternative to using RL are aforementioned neuro-evolutionary approaches that use evolutionary algorithms for optimizing the neural architecture. Evolutionary algorithms evolve a population of models, i.e., a set of (plausible) networks; in every evolution step, at least one model from the population is sampled and serves as a parent to generate off-springs by applying mutations to it. Neuro-evolutionary methods differ in how they sample parents, update populations, and generate offsprings by either applying tournament selection (Goldberg and Deb, 1991) or follow a multi-objective Pareto front to sample parents. Floreano et al. (2008) claimed that gradient-based methods outperform evolutionary methods for the optimization of neural network weights, and that evolutionary approaches should only be used to optimize the architecture itself, yet Stanley et al. (2019) highlighted the ability to account for diversity in genetic algorithms, which increases the chance of finding novel architectures and also allows for massively parallel exploration.

Bayesian optimization is one of the most popular methods for hyperparameter optimization, but it has not been applied to NAS by many groups since typical BO toolboxes are based on Gaussian processes and focus on low-dimensional continuous optimization problems. NASBOT from Kandasamy et al. (2018) is still worth mentioning for its efficiency.

While not exactly in the domain of architecture search, an important consideration in designing neural architectures is that of model compression. Several approaches have been adapted over time: quantization, pruning, knowledge distillation, etc. It is worth mentioning Stier et al. (2019)'s work, who proposed a game theoretic approach based on Shapley values to create efficient topologies. MorphNet [16] by Google follows such topologies, and after being applied on Inception V2 trained on ImageNet, it has scored the same test accuracy, while consuming around 10% less FLOPS.

2.3 Performance estimation

The main goal of NAS is to find architectures that achieve high predictive performances on a certain task with the ability to generalize when dealing with unseen data. Performance estimation refers exactly to the process of estimating this performance: the simplest option is to perform a standard training and validation of a generated architecture on data, and measure its performances, but this is unfortunately computationally expensive and limits the number of architectures that can be explored. Much recent research therefore has been focusing on developing methods to address this issue by reducing the cost of these performance estimations. A simple approach relies on low fidelity estimates obtained by training for fewer epochs or on a small subset of data as is done by Real et al. (2019). This approach could work if we could be sure that the relative ranking of architectures does not change due to the low fidelity evaluations, but recent research has shown this not to be the case. Another strategy is based on extrapolating from learning curves: architectures that are predicted to perform poorly from the learning curve in the initial few epochs can be terminated early to speed up the search process. Liu et al. (2018), on the other hand, don't use learning curves but suggest training a surrogate model (mentioned in 2.2.2), referred to as an *accuracy predictor*, used to predict the performance of an architecture based on properties extrapolated from other novel architectures in an adversarial manner. Other approaches rely on *parameter sharing*, either in the form of network morphism, were child networks are warm-started from surrogate parent networks' weights, or in the form of one-shot architecture search, popularized by Pham et al. (2018). The latter represents a family of NAS methods which attempt to train one super-architecture which subsumes all the other architectures in the search space. In order to do so parameters are shared between all the networks that are created by the search space navigation strategy, which also allows for fast evaluation since the weights of the super-graph can be inherited to evaluate the various different architectures without training them. While one-shot NAS has reduced the computational resources required for neural architecture search, it is unclear how the inheritance of weights from the super-graph affects the performance of the sub-graphs and the kind of biases it introduces into architectures, as [1] mentioned. Recent explorations on the effect of reduced training in NAS [17] have also discovered high rank correlations between various fully and partially trained connected architectures, whereas others [18] have had success optimizing a NAS in limited GPU time following a gene expression approach.

3 Algorithmic overview

3.1 DARTS

In 2019 Liu, Simonyan and Yang proposed DARTS [15], a differentiable architecture search method based on the continuous relaxation of typically discrete NAS search spaces. It also leverages game theoretic concepts for the optimization of neural architectures, as it poses the architecture search problem as a bi-level optimization, where one optimization problem is embedded in another.

DARTS attempts to solve three main problems that are typically encountered in NAS:

- Extremely large search spaces (*micro-search approach*).
- Discrete search space causing expensive optimization (*continuous relaxation*).
- Every candidate architecture must be trained from scratch (*parameter sharing*).

3.1.1 Search space

Following the recent advances in neural architecture search, DARTS embraces micro-search, looking for the best layer configuration instead of taking into consideration a complete multi-layer architecture. As already mentioned in section 2.1, these basic building blocks are defined as *cells*.

A DARTS cell is represented as a DAG, in which each node $x^{(i)}$ is a latent representation (e.g. a 2D kernel inside a CNN) and each directed edge (i, j) is associated with some operation $o^{(i,j)}$, for example a pooling operation, that transforms $x^{(i)}$. Each intermediate node is computed from all of its predecessors, as in eq.(1)

$$x^{(j)} = \sum_{i<j} o^{(i,j)}(x^{(i)}) \tag{1}$$

It is important to point out that a *zero* operation is always included into the search space, as it is needed in order to express the lack of connection between two nodes inside a cell.

Let $\mathcal{O}$ be a set of candidate operations, where each operation o represents some function $o(\cdot)$ to be applied to a node $x^{(i)}$. We also denote as N the number of nodes that we want our cell to be composed of. Given two nodes $x^{(i)}$, $x^{(j)}$ the directed edge (i, j) connecting them is parameterized by a vector $\alpha^{(i,j)} \in \mathbb{R}^{|\mathcal{O}|}$.

In order to obtain a continuous search space, the categorical choice of a assigning a particular operation to an edge (i, j) is relaxed to a *softmax* over all possible operations (2).

$$\bar{o}^{(i,j)}(x) = \sum_{o \in \mathcal{O}} \frac{exp(\alpha_o^{(i,j)})}{\sum_{o' \in \mathcal{O}} exp(\alpha_{o'}^{(i,j)})} o(x) \tag{2}$$

This has two major consequences: (i), the operation of an edge is the sum of all the operations placed on that edge, weighted by the *softmax* over $\alpha^{(i,j)}$; (ii), the task of architecture search reduces to learning a set of continuous variables α so that $\alpha = \{\alpha^{(i,j)} | i \neq j \wedge i, j \in [1, N]\}$, from which probability distributions of what operations are to be chosen for each edge are learned. Therefore, *finding the optimal cell* is to find the best operation to assign on each edge. The great advantage of DARTS is that each different choice is not investigated independently, but every possible operation is mixed so their weights are actually shared and "trained together". When the search ends, a discrete architecture can be extracted by replacing each *mixed operation* with the most likely one, i.e. $o^{(i,j)} = \arg\max_{o \in \mathcal{O}} \alpha^{(i,j)}$.

3.1.2 Optimization

The goal of DARTS is to jointly learn the architecture α and the weights w within all the mixed operations. While in RL based methods the performances onto the validation set are used as rewards to train a controller, DARTS is able to optimize the validation loss using gradient descent.

Let $\mathcal{L}_{train}$ and $\mathcal{L}_{val}$ be the training and validation loss respectively. Both losses depend both on current architecture α and on weights w in the cell operations and nodes. DARTS aims to find an

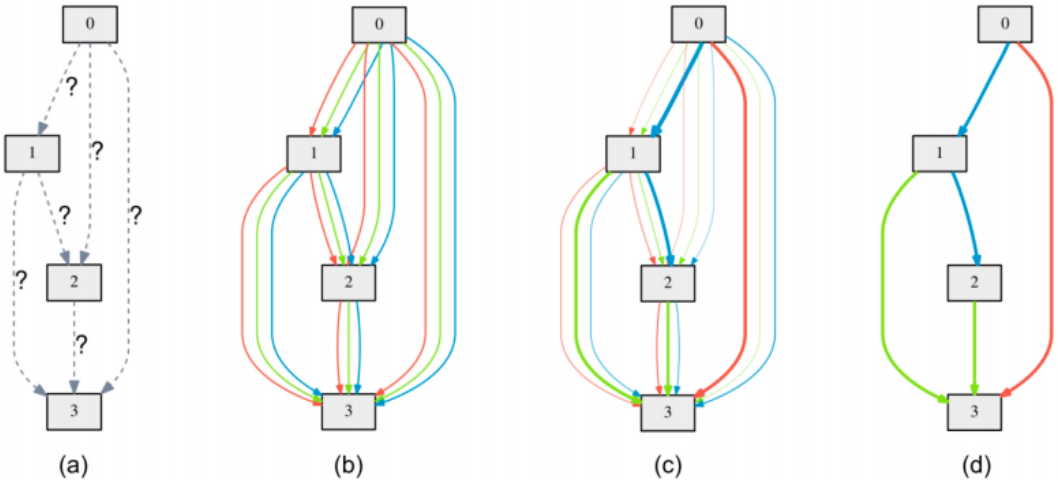

Figure 5: Overview of DARTS, as presented in [15]: Operations on the edges are initially unknown. (b) Continuous relaxation of the search space by placing a mixture of candidate operations on each edge. (c) Joint optimization of the mixing probabilities and the network weights by solving a bi-level optimization problem. (d) Inducing the final architecture from the learned mixing probabilities.

α^* that minimizes the validation loss $\mathcal{L}_{val}(w^*, \alpha^*)$, where $w^* = \arg\min_w \mathcal{L}_{train}(w, \alpha^*)$ are the weights associated to the architecture and are obtained by minimizing the training loss.

It is clear that the optimization of α depends on the optimization of w, implying that DARTS optimization task is actually a bi-level optimization problem where α is the upper-level variable and w is the lower-level one.

$$\min_{\alpha} \mathcal{L}_{val}(w^*(\alpha), \alpha) \text{ s.t. } w^*(\alpha) = \arg\min_{w} \mathcal{L}_{train}(w, \alpha) \tag{3}$$

Technically the inner optimization could be solved only by training current weights w until convergence, but this is only approximated by doing just a single training step, meaning that $\nabla_\alpha \mathcal{L}_{val}(w^*(\alpha), \alpha) \approx \nabla_\alpha \mathcal{L}_{val}(w - \xi \nabla_w \mathcal{L}_{train}(w, \alpha), \alpha)$, with ξ being the learning rate of the inner optimization. Then upper and lower optimizations are performed in alternated order, as summarized in algorithm 1. The final cell architecture is stacked for a predetermined number of time to form a larger and complete neural network that is then trained and tested.

Algorithm 1: DARTS – Differentiable Architecture Search

Create a mixed operation $\bar{o}^{(i,j)}$ parametrized by $\alpha^{(i,j)}$ for each edge (i, j)
while *not converged* **do**
  1. Update architecture α by descending $\nabla_\alpha \mathcal{L}_{val}(w - \xi \nabla_w \mathcal{L}_{train}(w, \alpha), \alpha)$
  2. Update weights w by descending $\nabla_w \mathcal{L}_{train}(w, \alpha)$
end
Derive the final architecture based on the learned α.

During their experiments, following what Zoph et al. did in 2018 for NASNet [19], two kinds of different cells are searched: *normal cells* and *reduction cells*, with the only difference that operations on reduction cells have stride 2 and are typically placed at 1/3 and 2/3 of the total depth of the network. The architecture encoding α then becomes a pair of sets $(\alpha_{normal}, \alpha_{reduce})$, where α_{normal} is shared just between normal cells while α_{reduce} is shared only among reduction cells.

Moreover, DARTS has been able to automatically build both convolutional and recurrent cells, reaching state of the art performances in image recognition and language modeling tasks. This by being orders of magnitude faster than most of currently available NAS systems, with the exception of ENAS, which has comparable performances and computation time.

3.2 ENAS

In 2018 H. Phame, B. Zoph et al. proposed a novel neural architecture search approach tailored at fast and inexpensive automatic model design, which goes by the name of ENAS [20]. It actually shares with DARTS most of the aims, like turning the navigation through diverse large search spaces into a feasible operation and avoiding losing weights on the candidate architectures, i.e. *child models*, as soon as they've been trained to start over from scratch. It is a controller-based implementation, where the controller, in the form of a recurrent neural network (typically a LSTM), uses the performance of its samples from previous steps as a guiding signal to find more promising architectures.

The main contribution from its introduction is that it pushed the boundaries of NAS systems training without massively dedicated hardware. Indeed, despite being a few % points shy of many human-made networks, it is on-par with other NAS systems performance, while dramatically outperforming them in terms of computational time: depending on the task, ENAS scored a time gain between 3 to 5 orders of magnitude, as it was able to traverse the local regions within a provided macro-architecture space in less than 18 hours on a single GeForce GTX 1080Ti, which go down to a stunning 10 hours for language modeling on the Penny Treebank dataset (Marcus et al., 1994).

3.2.1 Search space

The core idea behind ENAS is the observation that all of the graphs which NAS ends up iterating over can be viewed as sub-graphs of a larger graph (also know in the literature as *super-network*). In other words, we can represent NAS's search space using a single directed acyclic graph (DAG), whose sub-graphs map generic architectures that could be sampled out of it. In other words, ENAS's DAG can be seen has the superposition of all its valid child models in the search space, where nodes represent local computations (i.e., convolutional layer, max or average pooling, etc.), each with their own parameters, and edges represent the flow of information (see Figure 6). The parameters at node-level computations' are used only when that specific computation is activated, which in turn allows ENAS's children to share weights in the search space.

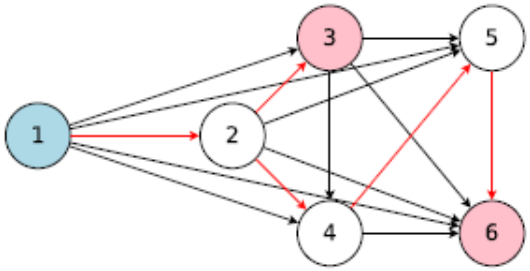

Figure 6: The above DAG represents the entire search space while the red arrows define a model in the search space, which is decided by a controller. Here, node 1 is the input to the model whereas nodes 3 and 6 are the model's outputs.

3.2.2 Designing layers

The design of layers changes depending on whether ENAS is trying to produce convolutional or recurrent neural networks, but the baseline task for the controller remains the same. Indeed at each time step (assuming we are at a certain hidden state h_i within the LSTM) he has two decisions to take: select which nodes should be connected to h_i-th layer as inputs, and fix the activation function or the local computation, for recurrent and convolutional networks, respectively. Being so similar in nature, and having mentioned both for the sake of completeness, we will just present how ENAS' controller designs recurrent networks. Let us define $N = 4$ the number of blocks of decision the LSTM has to go through, in a limited setting, where x_t is the input signal for a recurrent cell and h_{t-1} the hidden state from the previous step, as you can see from Figure 7.

- First, the controller samples an activation function that helps it start out from h_0 defining the current state $h_t = tanh(x_t \cdot W^{(x)} + h_{t-1}W_1^{(h)})$ where each weight matrix $W_{(i)}$ has to be learnt and shared across epochs.

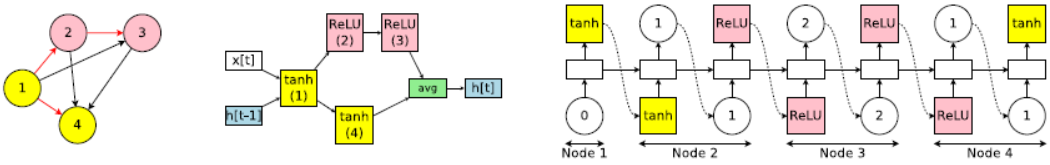

Figure 7: An example of a recurrent cell in our search space with 4 computational nodes. Left: The computational DAG that corresponds to the recurrent cell. The red edges represent the flow of information in the graph. Middle: The recurrent cell. Right: The outputs of the controller RNN that result in the cell in the middle and the DAG on the left. Note that nodes 3 and 4 are never sampled by the RNN, so their results are averaged and are treated as the cell's output.

- For any other node in the graph, the controller samples a previous index (which will function as input) and an activation function. By choosing always a previous index, they are forcing which parameter matrices can be used, because given a pair of indexes j, l with $j < l$ there will always be an independent matrix $W_{l,j}^{(h)}$ since l won't ever function as input to j. Therefore, in ENAS all recurrent cells in a space share the same set of parameters, which is a testament to parameter sharing not being a bad idea, at all.
- For the output, they simply average all the loose ends, i.e., the nodes that didn't get selected as input to any other node (h_3 and h_4 in the above example).

The search space defined as such includes an exponential number of configurations, which, by assuming 12 nodes (N) and 4 plausible activation functions for each of them, returns $4^N \times N!$ or, approximately, 10^{15} models. Similar numbers can be achieved in the case of convolutional layers, despite them allowing for skip connections through the prediction of previous indexes, via concatenation on depth of inputs, in case one layer has more than one.

3.2.3 Optimization

With the above definition of ENAS the optimization unrolls on a bi-level step, where the training of the controller is alternated to training of the sampled architectures. Also, the controller, which samples through softmax classifiers, evolves in an autoregressive fashion as the the decisions it took at the previous step will be fed as its input embeddeding into the next step; this means that at the first iteration the LSTM will receive an empty input, compliant with the idea that it has absolutely no clue of how a good or bad architecture should look like. There are then two sets of learnable parameters to take into account: θ, the controller parameters, and w, the shared parameters of the child models. The controller will perform an alternating stochastic gradient descent (SGD) by fixing one to descent the other, and viceversa. In particular at first the controller will fix the policy $\pi(m;\theta)$ and perform SGD on w to minimize the expected loss $\mathbb{E}_{m\sim\pi}[\mathcal{L}(m;w)]$, which is typically chosen as the cross-entropy loss, where m is a sample drawn from the distribution π. The gradient descent is computed using the Monte-Carlo estimate, approximating the expected loss with the empirical loss on the sequence of drawn models emulating the true distribution:

$$\nabla_w \mathbb{E}_{m\sim\pi}(m;\theta)[\mathcal{L}(m;w)] \approx \frac{1}{M}\sum_{i=1}^{M} \nabla_w \mathcal{L}(m_i, w) \tag{4}$$

The above result provides an unbiased estimate of the expectation gradient, but still introduces higher variance than standard SGD with m fixed. By experimenting, they found out an interesting result, that is, the above descent performs pretty well even if $M = 1$, i.e. a single sample m is used to update the gradient, even though they didn't have a clear answer yet to why that's the case. w is then updated through an entire run of the training data on the drawn sample m. When w has been updated, the other half of the bi-level optimization takes place: by fixing w the model tries to update the policy parameters θ following the policy gradient in order to maximize the expected reward $\mathbb{E}_{m\sim\pi(m;\theta)}[\mathcal{R}(m,w)]$ for some reward function R [4]; they suggest to evaluate the goodness of R

[4]They use REINFORCE algorithm, so they would actually need an intermediate step to compute the cumulative discounted reward

validation set rather than on the training set as a proxy to encourage ENAS to look for models in the search space that generalize well. In order to actually asses the best model, the one which scores the higher reward is taken and re-trained from scratch for 150 epochs. Of course, by retraining all the M models the table of rewards could vary by a good margin, but considering only the one with the initial highest reward is a good greedy approach that gave them a nice balance between accuracy and computational time, as ENAS has actually discovered a novel architecture on the Penny Treebank dataset, with state-of-the-art capabilities.

Extensive experiments have shown super promising results for ENAS, as it scored a 2.89% test error against DenseNet's 2.56 on CIFAR-10, while comprising less than half of their parameters. Also, the importance of the controller being trained at the same time to the child models weights w has been assessed (by preventing the controller from training the performance would decrease by up to 5%), as well as the tendency of ENAS to always look for a local minimum in the macro region at its disposal. As we've seen, as opposed to DARTS, ENAS can actually deal with both micro- and macro-search with good performance, since a computational DAG, albeit different, is used when local cells needs to be predicted instead of the overall architecture, as well; despite that, how you would choose the macro-architecture to feed it with, in the case of a micro-search setting for ENAS, doesn't have a precise answer yet. All in all, ENAS has left its mark on NAS research since it has paved the way to a lot of the current exploration in finding efficient ways to deal with macro-search assessment.

4 Proposed implementation

We provide as a part of this survey a naive implementation of a controller-based NAS system[5], whose main objective is sampling multi-layer perceptrons (MLPs) capable of solving a given task. This can be seen as a toy problem, since much more complex models should be sampled to tackle real-world tasks, for example CNNs, and those would need a much deeper reasoning on how to structure the search space, how to describe every single convolutional layer, and so on. The choice of searching for MLPs also convinced us to use a macro-search approach, as cell searching would have been a useless complication, given the scope of the implementation.

This NAS system is based on a LSTM model whose goal is to sample a probability distribution for every possible layer accounted by the search space, and then extract the next predicted layer through a random choice based on this distribution. At every epoch, the controller calls the MLP generator for a number of times equal to the desired number of architectures to sample per-epoch. The layers of these architectures are sampled in terms of unique indexes compliant to a vocabulary, that maps to a specific pair in the form of (*n. of neurons, activation function)*.

When all of the layers for a single architecture have been sampled, the whole architecture is decoded back to actual Keras layers, which are concatenated in order to obtain a trainable model. After the sampling phase, each architecture is trained on its dataset of choice for approximately 10 epochs with an early stopping of 3. This training procedure can actually take quite some time depending on the complexity of the dataset, so this must be one of the deciding factors of both the number of sampling epochs the controller performs and the number of architectures being sampled per-epoch.

4.1 Design choices

Many of the design choices we made were based primarily on the original NAS paper [21] and its more recent evolution, ENAS [20]. Alongside the classical RL based NAS mechanics, we also implemented a one-shot architecture search feature, meaning that identical pairs of layers (also addressed as *bigrams*) share weights connecting them. In order to do so we use a dictionary that stores bigrams of the layers sampled so far and their corresponding weights, that will be transferred to newly sampled architectures whenever the opportunity arises.

The code should be self-explanatory and structured in modular Python files, but we will still comment on a few key notes worth of mention. Regarding the controller, more LSTM layers could have been implied to sample the probability distributions, but we didn't test it out for time constraints. A few questions which may actually sound more appealing are what the true role of the NAS' controller is and how the LSTM itself would actually be trained. Its main role is to assure that the sequences

[5]Source code can be found on the repository at `https://github.com/ResonantFilter/AMLProject-NAS`

being sampled by the MLP generator are valid and that no repeating ones are being sampled to avoid wasting important computational resources on regions of the search space which have already been explored. But how well the controller traverses the search space depends on how well it is trained to follow the more optimal directions. After the first few sequences have been generated, trained and evaluated they will form a dataset the controller is fed with. In essence, after every controller epoch, a new dataset that collects the architectures sampled since the first epoch is created for the controller to learn from, and this should make it learn how to discriminate within its hidden states between architectures that perform well from those that do not.

The actual training of the controller must be dependant from the validation accuracies obtained by its generated architectures, thus its loss function implements the aforementioned REINFORCE algorithm. In our case, the controller functions as a *model-free agent*, that is an agent that doesn't try to understand the environment, but rather just focuses on the policy π.

4.2 Accuracy predictor

Our implementation opens up for the possibility to introduce an accuracy predictor, a network parallel to the LSTM itself, which turns the model into an adversarial model by accounting for an optimization that doesn't focuses solely on the above REINFORCE loss function, but also on how well the accuracy predictor is becoming in predicting the goodness of the sampled architectures on their task. Indeed the predictor is implemented by using a single dense layer which will share weights with the LSTM layer of the sequence layer, fed with the true validation accuracies reported by the previously sampled architectures at time t, thus letting it construct an internal representation of the architectures that allows it to understand the properties that characterize a good architecture as opposed to a bad one, without the need to train them for those 10 epochs, as a proxy to their validation accuracies. The controller, on the other hand, will try to navigate the search space in a way that also allows it to generate architectures not easily predictable by the predictor.

Despite the adversion of the predictor may lead to architectures with lower validation accuracies on some tasks, its usage is still desired to help sampling architectures that generalize better.

5 Results

5.1 Data and configurations

We conducted our experiments on three datasets:

1. MNIST Digits [22]
2. Wine-Quality
3. 10 Speakers, a dataset that has been used as part of a project for "Speech Processing and Recognition" course, containing audio features from 10 different speakers.

In particular we did extensive testing using "MNIST Digits" dataset, while "Wine-Quality" and "10 speakers" were included as additional playgrounds. Then we defined a search space for MLPNAS that can be summarized as follows:

- Possible layer widths: [8, 16, 32, 64, 128, 256, 512]
- Possible activation functions: [sigmoid, tanh, ReLU, ELU]
- Maximum architecture length: 10 layers

We also included the chance of sampling dropouts layer, for which dropout rate has been set constant to 0.2, for the sake of simplicity. The final layer is decided depending on the given task: we use a single neuron sigmoid in case of binary classification problems, while multi-classification uses a final softmax layer whose width is automatically set accordingly w.r.t labels inside the dataset.

Many hyperparameters for both the controller and generated architectures could have been tuned, but we could not do that in depth due to time limitations. We train the LSTM controller for 10 epochs, and for each epoch 10 architectures are generated. Each of the generated architectures are trained for 10 epochs as well. Both controller and child networks trainings are optimized using

Adam optimizer [23] with learning rates fixed at 0.01 and 0.001 for the controller and generated architectures respectively, keeping Keras' default value for β. The controller uses a learning rate decay of 0.1, while child networks use none.

Training time with LSTM's hidden size of 512 and looking for 20-layer-long architectures at maximum, took around 1 hour on a GeForce GTX 1660 Ti GPU on MNIST Digits dataset. The default LSTM's size has been fixed to 128, though, as the data we tested out was quite simple in nature (only numerical or categorical features), thus our MLPNAS model didn't need a bigger hidden space to learn a powerful enough input-output function Φ.

We also made an experiment in which we increased to 15 the number of epochs for the Controller as well as the number of architectures sampled per epoch, that has been brought to 30. With this configuration we encountered many promising architectures, even if it must be pointed out that we are dealing with a toy problem on a very simple dataset, and complex architectures may not be worthy. As we show in Table 4, even if the first architecture is the best we found so far with a 96% validation accuracy, it is too complex for the task at hand. This is clear when we look at the second architecture in Table 4, which reaches a 95% validation accuracy with just 2 layers.

5.2 Obtained MLP architectures

Tables in this section display architectures found by MLPNAS. In particular we show the worst and the best architectures among top 5 layouts discovered by MLPNAS for each task. ***Please notice that validation accuracy refers to validation accuracy obtained after 10 epochs of training during architecture generation, while test accuracy is obtained on a test set after 50 additional epochs of fine-tuning since we did a full train only on the most promising architecture.***

For each result, we also provide the test accuracy obtained from fitting a Scikit-learn's MLP Classifier [24] onto the same dataset.

Table 1: **MNIST Digits** – Without one-shot learning and predictor on 128 hidden size LSTM

Architecture	**Validation Accuracy**	**Test Accuracy**
Scikit-learn MLP test accuracy	–	0.93
(256, 'sigmoid'), (8, 'elu'), (256, 'tanh'), (256, 'relu'), (10, 'softmax')	0.68	–
(32, 'elu'), (512, 'sigmoid'), (32, 'elu'), (512, 'sigmoid'), (10, 'softmax')	**0.72**	**0.72**

Table 2: **MNIST Digits** – Using predictor only on 128 hidden size LSTM

Architecture	**Validation Accuracy**	**Test Accuracy**
Scikit-learn MLP test accuracy	–	0.91
(128, 'sigmoid'), (64, 'sigmoid'), (16, 'relu'), (512, 'elu'), (10, 'softmax')	0.66	–
(32, 'elu'), (32, 'relu'), (64, 'relu'), (64, 'elu'), (10, 'softmax')	**0.74**	**0.86**

Table 3: **MNIST Digits** – Using one-shot learning and predictor on 128 hidden size LSTM, controller batch size set to how many architectures are sampled per epoch.

Architecture	**Validation Accuracy**	**Test Accuracy**
Scikit-learn MLP test accuracy	–	0.92
(64, 'sigmoid'), (512, 'relu'), (10, 'softmax')	0.70	–
(512, 'sigmoid'), (10, 'softmax')	**0.77**	**0.84**

Table 4: **MNIST Digits** – Using one-shot learning and predictor on 128 hidden size LSTM, with extended controller batch size and amount of sampled architectures per epoch.

Architecture	**Validation Accuracy**	**Test Accuracy**
Scikit-learn MLP test accuracy	–	0.92
(128, 'elu'), (64, 'relu'), (64, 'relu'), (32, 'relu'), (512, 'sigmoid'), (64, 'tanh'), (64, 'relu'), (32, 'relu'), (256, 'relu'), (10, 'softmax')	**0.96**	**–**
(256, 'elu'), (10, 'softmax')	0.95	–

Table 5: **10 Speakers** – Using one-shot learning and predictor on 128 hidden size LSTM, controller batch size set to how many architectures are sampled per epoch.

Architecture	**Validation Accuracy**	**Test Accuracy**
Scikit-learn MLP test accuracy	–	0.21
(64, elu), (32, relu), (16, tanh), (512, relu), (32, elu), (16, sigmoid), (64, elu), (16, elu), (512, relu), (10, softmax)	0.42	**–**
(128, elu), (64, elu), (32, relu), (128, sigmoid), (32, relu), (8, elu), (32, relu), (16, elu), (32, tanh), (10, softmax)	**0.43**	**0.47**

Table 6: **Wine Quality** – Using one-shot learning and predictor on 512 hidden size LSTM

Architecture	**Validation Accuracy**	**Test Accuracy**
Scikit-learn MLP test accuracy	–	0.69
(512, sigmoid), (512, relu), dropout, (512, relu), (256, elu), (256, elu), (256, tanh), (512, elu), (256, elu), (3, softmax)	0.64	–
(16, sigmoid), (512, sigmoid), (3, softmax)	**0.66**	**0.64**

6 Conclusions

Although NAS methods are constantly improving, the quality of empirical evaluation in this field is still lagging behind when compared to other areas in machine learning, AI and optimization. Hence, more efficient performance evaluation strategies are to be researched, in order to give NAS systems a true shot at competing against human-made networks. While this branch of AutoML has clear potential in terms of lifting deep learning experts from the task of manually designing architectures, we think that – more importantly – studying in depth the properties of good neural networks could give us a chance of truly understand them, in a way that is not possible to this day.

References

[1] Dr. Debadeepta Dey. Advanced machine learning day 3: Neural architecture search. https://www.microsoft.com/en-us/research/video/advanced-machine-learning-day-3-neural-architecture-search/, October 2018.

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. 2015.

[3] Frank Hutter and Feurer M. Hyperparameter optimization, in automated machine learning. *Automated Machine Learning*, 2019.

[4] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. Automated machine learning: Methods, systems, challenges. *Automated Machine Learning*, 2019.

[5] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. *CoRR*, abs/1611.01578, 2016.

[6] Barret Zoph et al. https://ai.googleblog.com/2017/11/automl-for-large-scale-image.html.

[7] https://analyticsindiamag.com/why-tesla-invented-a-new-neural-network/.

[8] https://nni.readthedocs.io/en/latest/NAS/one_shot_nas.html.

[9] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *ArXiv*, abs/1808.05377, 2019.

[10] Hanzhang Hu. Macro neural architecture search revisited. 2019.

[11] Karim Ahmed and Lorenzo Torresani. Connectivity learning in multi-branch networks. *ArXiv*, abs/1709.09582, 2017.

[12] Shreyas Saxena and Jakob Verbeek. Convolutional neural fabrics. *ArXiv*, abs/1606.02492, 2016.

[13] Richard Shin, Charles Packer, and Dawn Xiaodong Song. Differentiable neural network architecture search. In *ICLR*, 2018.

[14] Tom Veniat and Ludovic Denoyer. Learning time-efficient deep architectures with budgeted super networks. *ArXiv*, abs/1706.00046, 2017.

[15] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *ArXiv*, abs/1806.09055, 2019.

[16] https://ai.googleblog.com/2019/04/morphnet-towards-faster-and-smaller.html.

[17] George Kyriakides and Konstantinos G. Margaritis. The effect of reduced training in neural architecture search. *Neural Computing and Applications*, 04 2020.

[18] Jeovane Honorio Alves and Lucas Ferrari de Oliveira. Optimizing neural architecture search using limited gpu time in a dynamic search space: A gene expression programming approach, 2020.

[19] Barret Zoph, V. Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8697–8710, 2018.

[20] 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.

[21] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. *ArXiv*, abs/1611.01578, 2017.

[22] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[23] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014. cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

[24] Scikit-learn mlp classifier. `https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html`.

[25] Kirthevasan Kandasamy, Willie Neiswanger, Jeff Schneider, Barnabás Póczos, and Eric P. Xing. Neural architecture search with bayesian optimisation and optimal transport. *CoRR*, abs/1802.07191, 2018.

[26] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, USA, 2004.

[27] E. L. Lawler and D. E. Wood. Branch-and-bound methods: A survey. *Operations Research*, opre/14.4.699, 1966.

[28]

Appendices

A REINFORCE algorithm

A loss often implied to maximize the policy gradient is given by the REINFORCE algorithm, which is a typical algorithm in RL that represents a Monte-Carlo variant of a stochastic policy gradient. Its objective is then to learn a policy that maximizes the cumulative future reward score G by means of a combination of the rewards produced by a reward function R, computed on all the (a_i, s_i) pairs, where the policy π is defined as a probability distribution of actions in which a higher probability value corresponds to a higher expected reward for an action from a given observed state.

The REINFORCE algorithm tries to maximize an objective J defined as the product of the cumulative future discounted reward $G(t)$ w.r.t. a baseline (0.5, for example, that basically will make it discriminate about half of the actions as good and half as bad) with log-probabilities of actions w.r.t. the policy as $\ln \pi(A|S, \theta)$, where $A = [a_1, ..., a_T]$. Formally, it can be written as:

$$J(\theta) = \alpha\gamma^t G[\nabla_\theta \ln \pi(a_t|s_t, \theta)]$$

Algorithm 2: REINFORCE

N number of epochs;
T number of steps;
;
θ random;
;
for *n = 1, ..., N* **do**
    Generate an episode, (s_1, a_1, r_1), ..., (s_T, a_T, r_T), following the current policy $\pi(.|., \theta)$;
    for *t = 1, ..., T* **do**
        Compute the discounted cumulative reward $G(t)$ [6];
        $\theta \leftarrow \theta + \alpha\gamma^t G[\nabla_\theta \ln \pi(a_t|s_t, \theta)]$;
    end
end

where γ is the discount factor and $\pi(a_t|s_t, \theta)$ the probability of the occurrence of (s_t, a_t) given the current trajectory τ followed by the agent, regulated by the learning rate, α. Indeed, the learning rate is a key factor in traversing appropriate regions of the search space, because a NAS model could very easily get stuck in regions where no good architectures can be found for the problem at hand.

Since in REINFORCE algorithm the sample gradient expectation is equal to the actual gradient, expressed in terms of log-probabilites on π, it reflects a good theoretical convergence, albeit being a Monte-Carlo-based method it may suffer from high variance.

B Model selection and Cross validation

Model selection has always been one of the most researched areas in machine learning, as the ability to assess the goodness of a model while being able to identify the most suitable configuration set of parameters is key behind most learning algorithms. In this appendix we will try to give both a practical intuition and a theoretical proof built on sound mathematical foundations that sit behind the convergence of an algorithm from statistical learning theory, a well-developed field with many interactions with mathematics, computer science and probability theory.

As we know, the idea underlying supervised statistical learning is that, given a $X \times \mathcal{R}$ space, with distribution measure P, and a measurable loss function l that maps $Y \times Y \rightarrow [0, \infty)$, we have a knowledge of a subset $\hat{S} = \hat{S}_n = (x_1, y_1), ..., (x_n, y_n) \sim P^n$ of n i.i.d. samples w.r.t. P fixed, but unknown, with the aim of minimizing the expected risk $L(f)$ defined as:

$$L(f) = \mathbb{E}_{(x,y)\sim P}[l(y, f(x))] \tag{5}$$

which can be interpreted as the capability of the model in predicting unseen data.

With that in mind, we would like to address two simple questions:

- *Model assessment*: how can we measure future performance of a model?
- *Model tuning*: how can we choose the best hyper-parameters?

First off, we define a learning algorithm as a map that given a dataset $\hat{S}$ sends it in some function $A(\hat{S}) = \hat{f}$, where $\hat{f} \in \mathcal{F}$, that is, the space of all possible existing functions. A common practice is to imply empirical risk minimization (ERM), where the true expected risk $L(f)$ is substituted by a proxy evaluated on the available data, that is, the empirical risk $\hat{L}(f)$, which turns the objective of learning algorithms into its minimization instead:

$$\hat{f} = \min_{f\in\mathcal{H}} \hat{L}(f), \quad \hat{L}(f) = \frac{1}{n}\sum_{i=1}^{n} l(y_i, f(x_i)) \tag{6}$$

which can also be seen as drawing all the data samples in $\hat{s}$ from a uniform distribution. $\mathcal{H}$ in this case represents a subset of $\mathcal{F}$ and has to be intended as a space of suitable functions for the problem at hand, as we will discuss later in Section B.1.2.

[6] Please, refer to *mlpnas.py* in the code for details on $G(t)$ computation.

B.1 Model assessment

The first quantity that we want to introduce to assess how well a model can generalize to future data is known as the *excess risk*, defined as:

$$L(\hat{f}) - \min_{f \in \mathcal{F}} L(f) \tag{7}$$

which, given an empirical solution $\hat{f}$, compares its expected risk against what can be seen as the **very best model** (on unseen data) extracted from the global functional space $\mathcal{F}$, in order to measure if their orders of magnitude are close enough. One thing to note is that the excess risk is a random variable itself, as it depends on the empirical solution $\hat{f}$, which in turn is related to a random set $\hat{S}$: by changing the dataset we train our model on, we can get randomly different solutions that may achieve different levels of performance.

B.1.1 Mathematical foundations

In order to evaluate the excess risk we have to introduce an other quantity:

$$\mathcal{P}_{\hat{S} \sim P^n}(L(\hat{f}) - \min_{f \in \mathcal{F}} L(f) > \epsilon) = \Delta_n, \quad \forall \epsilon > 0 \tag{8}$$

which formalizes the cumulative distribution function over the training set $\hat{S}$ of the excess risk (being a random variable) bigger than a given non-zero threshold ϵ, that is, of the expected risk of the empirical solution exceeding the very best expected risk by more than ϵ. This is a strong requirement since we would like to understand the behaviour of this distribution. An even stronger requirement would be to ask for this quantity to converge when we are given infinitely many samples in the dataset $\hat{S}$, which is known as (sample) *consistency*:

$$\lim_{n \to \infty} \Delta_n = 0 \tag{9}$$

which has even more stringent variants known as consistency *in expectation* and consistency *almost surely* that tackle the two namesake notions of convergence that a random variable might have.

An other requirement that we might add would be to ask not only for this quantity to converge, but also to know how fast it would do so, which is known as the *non-asymptotic bounds* property:

$$\Delta_n \leq \delta(n, \epsilon, \mathcal{F}) \quad \text{for some function } \delta \tag{10}$$

What we have defined so far basically tells us that given any algorithm and any problem within $\mathcal{F}$, we are able to identify how fast it will converge, as it will have a small (finite) sample complexity for many different distributions P; but is this actually true? **Not really**, and this is formalized by the *no free-lunch theorem*, which states as follows:

"Given an algorithm there exists a problem for which the learning performance is arbitrarily bad", or in other words, there will be at least a problem for which the excess risk Δ_n is unbounded since every algorithm needs a so-called *inductive bias*, i.e. an assumption. This can be somewhat trivially proved with a counter-example, but still, it is a fundamental result for algorithm design.

B.1.2 Algorithm design

The most standard procedure to design algorithms takes place in two stages:

1. Pick a (possibly large) class of functions $\mathcal{H}$, known as the *hypothesis space*, which should be rich enough to comprise a plethora of types of functions, s.t. ideally:

$$\min_{f \in \mathcal{H}} L(f) = \min_{f \in \mathcal{F}} L(f)$$

2. Define a map $A_\gamma(S_n) = \hat{f}_\gamma \in \mathcal{H}$ to explore the space $\mathcal{H}$, where $\hat{f}_\gamma$ represents a one-parameter (one- just for simplicity) family of solutions on γ that controls the regularity of the solution and the performance of the algorithm (i.e., kernel size, reg parameter, etc.).

where the size of the exploration within $\mathcal{H}$ itself is what defines explicitly or implicitly the inductive bias of the learning algorithm. For example, in the case of ERM a classical definition of $\mathcal{H} = \mathcal{H}_\gamma$ would be to take a sequence of increasingly bigger spaces, as follows:

$$\mathcal{H}_1 \in \mathcal{H}_2 \in ... \in \mathcal{H}_\gamma \in ... \in ...\mathcal{H}$$

where the spaces $\mathcal{H}_\gamma$ might be taken as the spaces of linear models $f(x) = w^T x$ (or $f(x) = w^T \Phi(x)$ with feature maps), s.t. every weights vector w has l_2-norm smaller than γ, i.e. $||w|| < \gamma$, or maybe the l_0-norm $||w_0|| < \gamma$, etc. This is a simple example, as an endless amount of more complex choices beyond constrained ERM could be made, but it gives an idea on how crucial the choice of the hypothesis space is: with this $\mathcal{H}$ low-norm functions would be learnt well by the model, whereas it would struggle with high-norm ones since they wouldn't be implied by its inductive bias.

B.2 Model tuning

What changes when we have to deal with a family of solutions is that not only we'd like to assess the model's performance on unseen data, but also we'd like to tune it through the hyper-parameter space in order to identify the parameter that makes it generalize the best. The best theoretical parameter tuning would be given by the γ that minimizes the excess risk:

$$\gamma^* = \inf_\gamma L(\hat{f}_\gamma) - \min_{f \in \mathcal{H}} L(f)$$

which can be further decomposed into:

$$L(\hat{f}_\gamma) - \min_{f \in \mathcal{H}} L(f) = \underbrace{(L(\hat{f}_\gamma) - L(f_\gamma))}_{Variance/Estimation} + \underbrace{(L(f_\gamma) - \min_{f \in \mathcal{H}} L(f))}_{Bias/Approximation}$$

where the first term in brackets quantifies the sample error due to finite sampling (i.e., variance) and the second one quantifies the approximation error due to the chosen regularization scheme (i.e., bias) – recall that the $\hat{f}$ means the function was learnt from the training set $\hat{S}$. This leads to the well-known result of the best γ, i.e. γ^*, being the one which solves the *bias-variance trade-off*.

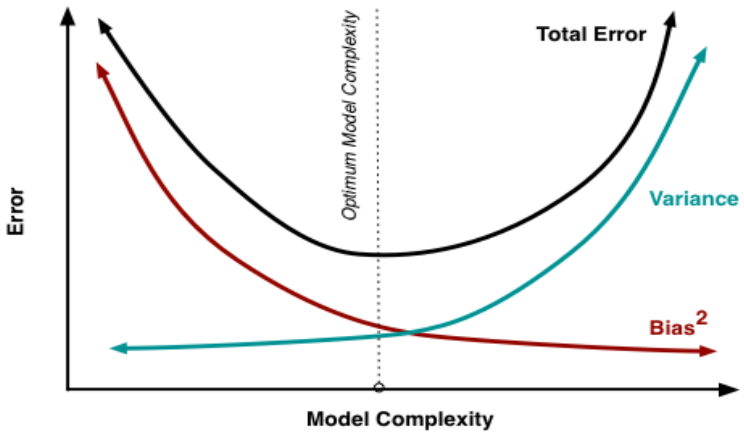

Figure 8: Bias-variance trade-off with best possible parameter γ^*

Despite all of the above being correct from a theoretical standpoint, both the expected risk and excess risk cannot be computed in practice, since they require knowledge of the probability distribution P which we don't have access to. So, given a family of functions $\hat{f}_\gamma$, how can we actually assess their quality $L(\hat{f}_\gamma)$ and select the best one, $\hat{f}_{\gamma^*}$, in practice?

B.3 Cross-validation

The usual answer to the above question is cross-validation, which can be summarized as follows (regardless of the splitting strategy in use):

1. Split the dataset $\hat{S}$ into training and validation set, $\hat{S}_{Train}$ and $\hat{S}_{Val}$;
2. Train the model $\hat{f}_\gamma$ on the training set $\hat{S}_{Train}$ only;
3. Assess it by ERM on the validation set $\hat{S}_{Val}$;
4. Repeat 2. and 3. for each γ available and select the best one.

This procedure has been proven empirically (i.e., it's been applied to learning algorithms) to be successful multiple times, but what we would like to do next is to give a theoretical proof of why that's the case, by making use of the mathematical tools that we've defined in Section B.1.

For simplicity, we will assume at first to have only one parameter γ and, just for reference, we will consider the simplest form of cross-validation, that is hold-out cross validation (HO, for short), which follows the few points listed above word for word:

1. Split the dataset $\hat{S} = \hat{S}_{Train} \cup \hat{S}_{HO}$ into a training set and a hold-out set of m samples which will be kept fixed, where $\hat{S}_{HO} = (\tilde{x}_1, \tilde{y}_1), ..., (\tilde{x}_m, \tilde{y}_m)$;
2. Use only $\hat{S}_{Train}$ to learn $\hat{f}_\gamma$;
3. Assess it by $\hat{L}_{S_{HO}}(\hat{f}_\gamma) = \frac{1}{m}\sum_{j=1}^{m} l(\tilde{y}_j, \hat{f}_\gamma(\tilde{x}_j))$.

What we would like to prove is that the expectation of the empirical risk on the validation set $\hat{S}_{HO}$ is a good estimate of the expected risk w.r.t. $\hat{f}_\gamma$, whereas the same doesn't hold on the training set, since this would justify the need to separate a validation set $\hat{S}_{HO}$ in the first place:

$$\mathbb{E}[\hat{L}_{S_{HO}}(\hat{f}_\gamma)] = L(\hat{f}_\gamma)$$
$$\mathbb{E}[\hat{L}(\hat{f}_\gamma)] \neq L(\hat{f}_\gamma)$$

In order to to do this, we just have to recall a basic property of i.i.d. random variables – for any sequence of n i.i.d. random variables $Z_1, ..., Z_n$ drawn from the same distribution Z, we can write:

$$\mathbb{E}_{Z_1,...,Z_n}[\frac{1}{n}\sum_{j=1}^{n} Z_j] = \mathbb{E}[Z] \tag{11}$$

since by linearity of the expectation operator (it's just an integral), we can derive:

$$\mathbb{E}_{Z_1,...,Z_n}[\frac{1}{n}\sum_{j=1}^{n} Z_j] = \frac{1}{n}\sum_{j=1}^{n}\mathbb{E}_{Z_1,...,Z_n}[Z_j] \tag{12}$$
$$= \frac{1}{n}\sum_{j=1}^{n}\mathbb{E}_{Z_j}[Z_j] \tag{13}$$
$$= \mathbb{E}[Z] \tag{14}$$

where the second equality is crucial, as the expectation over n random variables $\mathbb{E}_{Z_1,...,Z_n}$ is simply a n-ple integral that, if taken w.r.t. only to Z_j collapses to a single integral, $\mathbb{E}_{Z_j}[Z_j]$. This equality not only holds for straight random variables, but also for a sequence of i.i.d. random variables $Z_1, ..., Z_n$ passed through a given function h, since a function applied to a random variable is itself a random variable that belongs to the transformed probability distribution $h(Z)$, s.t. we can write:

$$\mathbb{E}_{Z_1,...,Z_n}[\frac{1}{n}\sum_{j=1}^{n} h(Z_j)] = \mathbb{E}[h(Z)] \tag{15}$$

Now, to port these results to the expectation over the held-out set $\hat{S}_{HO}$, we just have to fix $Z_j = (\tilde{x}_j, \tilde{y}_j)$ and h to the measurable loss function of choice s.t. $h(Z_j) = l(\tilde{y}_j, \hat{f}_\gamma(\tilde{x}_j))$, as such:

$$\mathbb{E}_{S_{HO}\sim P^m}[\hat{L}_{S_{HO}}(\hat{f}_\gamma)] = \frac{1}{m}\sum_{j=1}^{m}\mathbb{E}_{S_{HO}\sim P^m}[l(\tilde{y}_j, \hat{f}_\gamma(\tilde{x}_j))] \tag{16}$$

$$= \frac{1}{m}\sum_{j=1}^{m}\mathbb{E}_{\{(\tilde{x}_1,\tilde{y}_1),\ldots,(\tilde{x}_m,\tilde{y}_m)\}\sim P^m}[l(\tilde{y}_j, \hat{f}_\gamma(\tilde{x}_j))] \tag{17}$$

$$= \frac{1}{m}\sum_{j=1}^{m}\mathbb{E}_{(\tilde{x}_j,\tilde{y}_j)}[l(\tilde{y}_j, \hat{f}_\gamma(\tilde{x}_j))] \tag{18}$$

$$= L(\hat{f}_\gamma) \tag{19}$$

When we take into account the training set $\hat{S}_{Train} = (x_1, y_1), ..., (x_n, y_n)$ instead, the transformed random variables $l(y_1, \hat{f}_\gamma(x_1)), ..., l(y_n, \hat{f}_\gamma(x_n))$ are **not** independent anymore, since $\hat{f}_\gamma$ was computed on $\hat{S}_{Train}$ itself. The derivation shown above in Eq. (13-14) cannot be replicated here since the expectations over all the random variables in $\hat{S}_{Train}$ doesn't collapse to single integrals, hence:

$$\mathbb{E}_{S_{Train}}[\hat{L}(\hat{f}_\gamma)] \neq L(\hat{f}_\gamma)$$

which concludes our proof. We've proven theoretically that the empirical risk on the validation set $\hat{S}_{HO}$ is a good estimate in expectation of the true expected risk on $\hat{f}_\gamma$ since it is unbiased, unlike the one on the training set $\hat{S}_{Train}$. But we can say something more, actually, regarding the speed of convergence by simply expressing in a quantitative form the well-known *law of large numbers*:

$$\lim_{n\to\infty}\mathbb{P}\left(|\frac{1}{n}\sum_{i=1}^{n}Z_i - \mathbb{E}[Z]| \geq \epsilon\right) = 0 \tag{20}$$

for any sequence of i.i.d. random variables $Z_1, ..., Z_n$ and $\epsilon > 0$.

To actually say something about the speed of convergence we have to introduce another set of quantities known as *concentration inequalities*, which represent a key building block within the realm of all learning theory, since they relax the need of the law of large numbers for infinitely many samples within $\hat{S}$ and quantify a notion of the upper bound. As an example, we will express one of the simplest concentration inequalities, known as *Hoeffding inequality*, which provides an upper bound to the probability that the empirical risk of bounded i.i.d. random variables (i.e., $|Z_i| <= C$, for a given C) deviates from its expected risk by more than a certain amount ϵ, defined as:

$$\mathbb{P}\left(|\frac{1}{n}\sum_{i=1}^{n}Z_i - \mathbb{E}[Z]| \geq \epsilon\right) \leq 2e^{-C\epsilon^2 n} \tag{21}$$

which not only tells us that the empirical risk is going to converge to the expected risk, but also that it's going to converge exponentially fast, depending on the bound C. This result strongly resembles the non-asymptotic bounds property that we defined in Section B.1, and indeed if we just call $\delta = 2e^{-C\epsilon^2 n} \in [0, 1]$ and revert the inequality sign, we can state that with probability at least $1 - \delta$, or in other words with *high probability*:

$$|\frac{1}{n}\sum_{i=1}^{n}Z_i - \mathbb{E}[Z]| \leq \frac{1}{\sqrt{Cn}}\sqrt{\log\left(\frac{2}{\delta}\right)}^{7} \tag{22}$$

Now, if we go back to our empirical and expected risk over $\hat{S}$ instead of generic i.i.d. random variables Z_i and we consider a bounded loss function $l(y, f(x))$ for any f (and in particular the $\hat{f}_\gamma$

[7] Substitute $\epsilon = \frac{1}{\sqrt{Cn}}\sqrt{\log\left(\frac{2}{\delta}\right)}$ within the brackets.

learnt from $\hat{S}_{Train}$) and (x, y), we can write from (22) with probability at least $1 - \delta$:

$$|\hat{L}_{S_{HO}}(\hat{f}_\gamma) - L(\hat{f}_\gamma)| \leq \frac{1}{\sqrt{Cn}}\sqrt{\log\left(\frac{2}{\delta}\right)}$$

$$\vdots$$

$$L(\hat{f}_\gamma) \leq \hat{L}_{S_{HO}}(\hat{f}_\gamma) + \frac{1}{\sqrt{Cn}}\sqrt{\log\left(\frac{2}{\delta}\right)}^{8}$$

which confirms once more that the empirical risk on the held-out set S_{HO} is a good estimate to the expected risk minus a deviation which depends on the inverse of $\sqrt{n}$.

What about model tuning? What if instead of considering a single γ, we assume to have a distribution, or hyper-parameter space from which to extract the best γ on the expected risk?

$$\gamma_* = \arg\min_\gamma L(\hat{f}_\gamma)$$

What happens in practice is that we compute the above procedure on the held-out set S_{HO}

$$\hat{\gamma}_{S_{HO}} = \arg\min_\gamma \hat{L}_{S_{HO}}(\hat{f}_\gamma)$$

hoping that $L(\hat{f}_{\hat{\gamma}_{S_{HO}}}) \approx L(\hat{f}_{\gamma^*})$, or, $L(\hat{f}_{\hat{\gamma}_{S_{HO}}}) - L(\hat{f}_{\gamma^*}) \approx 0$, that is, that the best parameter computed on the S_{HO} is close to the true best one w.r.t. the empirical function $\hat{f}$. Can we have any guarantees on this by using the tools defined above for a single γ? **Yes**, we can!

The first thing that we can do is to assume to have N possible values $\gamma_1, ..., \gamma_N$, since in practice we cannot hope to explore a possibly infinitely large hyper-parameter space. With a moderately trivial derivation through a decomposition, the union bound property and Hoeffding's inequality applied to $L(\hat{f}_{\hat{\gamma}_{S_{HO}}}) - L(\hat{f}_{\gamma^*})$, and given a $\delta \in [0, 1]$, we can prove with a probability of at least $1 - \delta$:

$$|\hat{L}(\hat{f}_{\hat{\gamma}_{S_{HO}}}) - L(\hat{f}_{\gamma^*})| \leq \frac{2}{\sqrt{Cm}}\sqrt{\log\left(\frac{2N}{\delta}\right)}^{9}$$

$$\vdots$$

$$L(\hat{f}_{\gamma^*}) \leq \hat{L}(\hat{f}_{\hat{\gamma}_{S_{HO}}}) + \frac{2}{\sqrt{Cm}}\sqrt{\log\left(\frac{2N}{\delta}\right)}^{8}$$

for which we can make a similar reasoning to the single γ case, albeit here the number of different parameters that we want to try, N, appears in the deviation's numerator. Ideally, we would like to try as much parameters as we could and thus N to be possibly very large; but, as shown here, this would increase the deviation, that is, the precision at which we could make our estimate would decrease.

Of course, in this discussion we are assuming the N parameters to be sufficiently dense w.r.t. how well they are spaced within the hyper-parameter space; further proof would indeed be needed to show that the estimating γ from a finite set of N possibilities is a good enough approximation to estimating it from the infinite hyper-parameter space. Where does assessment fit in all of this? Or, in other words, after tuning the model to identify the best γ, is it reasonable to assess it on S_{HO}? **No!** Indeed by following a similar reasoning to before in (11-14), the $(x_1, y_1), ..., (x_m, y_m)$ pairs within S_{HO} wouldn't be independent from the candidate $\gamma_{S_{HO}}$, hence:

$$\mathbb{E}_{S_{HO} \sim P^m}[\hat{L}_{S_{HO}}(\hat{f}_{\hat{\gamma}_{S_{HO}}})] \neq L(\hat{f}_{\hat{\gamma}_{S_{HO}}}) \tag{23}$$

So, the only way to assess the model after computing $\hat{f}_\gamma$ on the training set S_{Train} and tuning the best γ on the validation set $\gamma_{S_{HO}}$ is to split again into a test set S_{Test} of fresh data, forming the common three-way split used in practice $\hat{S} = \hat{S}_{Train} \cup \hat{S}_{Val} \cup \hat{S}_{Test}$, from which:

$$\mathbb{E}_{S_{Test}} \sim P^m[\hat{L}_{S_{Val}}(\hat{f}_{\hat{\gamma}_{S_{Val}}})] = L(\hat{f}_{\hat{\gamma}_{S_{Val}}})$$

[8] Only the right half of the absolute value is shown

[9] Define δ here as $2Ne^{\frac{-C\epsilon^2}{4n}}$

We've shown that cross-validation is indeed a valuable procedure since it's backed up by sound mathematical proofs that show that the expected risk on γ^* (fixed or not) is squeezed by the empirical risk up to a certain degree. We've also proved why in practice it is recommended to split the data in three chunks, while keeping one always fresh. Despite focusing on hold-out cross-validation these results can actually be extended to all other kinds of cross-validation (i.e. multi-pass, sequential, w/ or w/o replacement, unbalanced classes, etc.) in a similar manner.

B.4 Model selection in Neural Architecture Search

An interesting approach to neural architectures' tuning was the previously mentioned NASBOT [25], a Gaussian-process-based Bayesian optimization framework that is able to evaluate the similarity between architectures by means of a distance metric in the search space of neural architectures, which can be computed efficiently via an optimal transport program. This distance might be of independent interest to the deep learning community as it may find applications outside of Bayesian optimization since it proved to outperform other alternatives for architecture search in several cross-validation-based model selection tasks on convolutional networks.

C Convex vs. non-convex problems

Whenever we try to find the best approximate function capable of solving a given learning problem, it is vital to assess if the task we are trying to learn is convex or non-convex. In other words, we'd want to reason on the convexity of the target objective function that we want to minimize w.r.t. the parameters that characterize its domain before even designing a learning algorithm.

Dealing with a convex function benefits from a series of important advantages:

- Every local minimum is also a global minimum.
- Strong theoretical guarantees regarding convergence of the optimization process [26].
- In the case of strong-convexity, there is a unique global minimum.

On the other hand, non-convex functions are much harder to optimize, as they have potentially many local minima, saddle points and flat regions. Consequently, there is a lack of convergence guarantees, and the risk of reaching sub-optimal solutions is relatively high depending on the specific problem.

We formally say that a function $f : \mathcal{X} \subseteq \mathbb{R}^n \rightarrow \mathbb{R}$, where $\mathcal{X}$ is a convex set, is convex iff:

$$\forall\, \mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_k \in \mathcal{X}, \forall\, \{\theta_1, \theta_2, ..., \theta_k \mid \sum_{i=1}^{k} \theta_i = 1\} :$$

$$f(\theta_1 \mathbf{x}_1 + ... + \theta_k \mathbf{x}_k) \leq \theta_1 f(\mathbf{x}_1) + ... + \theta_k f(\mathbf{x}_k) \qquad \textit{(Jensen's inequality)}$$

from which the definition of strong convexity is given by substituting in the inequality $\leq$ with $<$ and by adding the constraint $\mathbf{x}_1 \neq \mathbf{x}_2, ..., \neq \mathbf{x}_k$.

Typical examples of convex problems are least squares regression, as well as its regularized versions (e.g. ridge or lasso regression), and logistic regression. All those problems that instead do not rely on convex functions optimization are said to be **non-convex problems**, and among them we can find common algorithms like PCA and K-means.

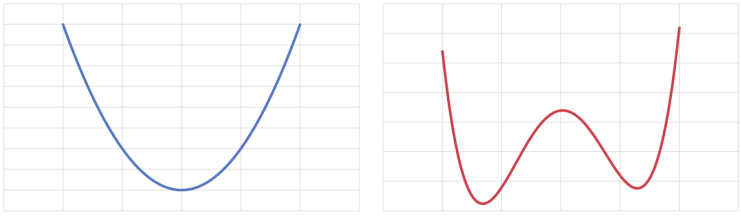

Figure 9: Left: a convex function; Right: a non-convex function

Even when dealing with non-convex problems we are still able to solve them by recurring to the same strategies implied for convex ones, such as SGD, or to specialized methods that use alternating minimization or *branch-and-bound* methods [27], albeit convergence is not guaranteed.

C.1 Neural networks are (in general) non-convex

A well-known example among non-convex problems is the minimization of the loss function of deep neural networks, as it is in general neither convex nor concave. This means that the matrix of all second partial derivatives (the Hessian) is neither positive, nor negative, semi-definite.

The non-convexity can also be seen as a consequence of the objective function $J(W, b)$ of a neural network being the result of many function compositions: even if each single function was convex taken individually, in general the composition of convex functions is not convex, unless additional assumptions are taken into account, but they do not usually apply to neural networks.

As being non-convex, the cost function $J(W, b)$ has also a number of local maxima and minima that are not necessarily global, saddle points and flat regions. This results in having possibly many symmetric configurations, reachable by simply permuting intermediate neurons positions and weights, with the very same approximation capabilities. A plausible workaround to overcome these issues would be to imply only linear layers, but as we know, many of the most common activation functions in use are non-linear (i.e., sigmoid, ReLU, etc.), as they satisfy the *Universal approximation theorem*. Even then, this would only work for small neural networks, as by increasing their depth the non-convexity would be enhanced by the function compositions problem stated above. One thing worth mentioning though, is that deeper networks tend to suffer less from bad local minima (that is, almost all of them coincide to a global minimum), at the expenses of bad saddle points, where the Hessian becomes really unstable and oscillates between non-convex to non-concave.

We still optimize the loss of neural networks using SGD, whose update rule in (24) for the convex setting still holds for the non-convex one. The key problem is that SGD doesn't necessarily go towards the optimum, even in expectation, thus making the parameters' initialization a delicate step that has to be taken with extra-care to reach a reasonable convergence (see Figure 10).

$$w_{t+1} = w_t - \alpha_t \nabla \tilde{f}_t(w_t) \tag{24}$$

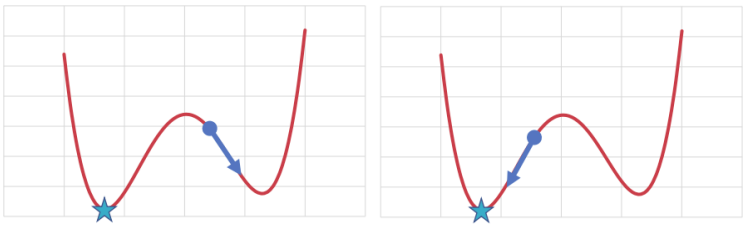

Figure 10: Effect of initialization on convergence towards a global minimum [28].

C.2 Convexity in Neural Architecture Search

As we've seen in Section 2.1, the task of searching for the best candidate neural network to solve a given problem is tightly constrained by the definition of a search space. If we look at the search space $\mathcal{D}$ of all valid neural networks, it is reasonable to assume that it is a non-continuous, categorical and unbounded space, as it suffers even more from the issues stated above. Moreover, we do not even have an explicit ordering rule for it, and even if extensive study on this precise matter is hard to find, one could claim that without further constraints it is not possible to define $\mathcal{D}$ as a convex set.

We also know that a convex function must be defined on a convex set, therefore learning to design good architectures for a given task, NAS, is in general a *non-convex* problem.

If we are interested in the convergence of NAS methods and dive in closer to the two algorithms discussed in detail, ENAS and DARTS, we can say that in both cases the authors do not provide theoretical proofs about the convergence of their method.

DARTS authors claim that in practice the algorithm is capable of reaching a fixed point, given that the learning rate of the inner optimization is suitably initialized [15].

ENAS authors make an even a stronger claim, stating that their method empirically shows that it will always reach the local minimum closest to the starting point given its search configuration, since by maneuvering the weights of the proposed architecture the performance would always degrade [20].