Contributions to Neural Architecture Search in Generative and Heterogeneous Multi-task Modeling

UNAI GARCIARENA HUALDE

Supervised by Roberto Santana and Alexander Mendiburu

2021

eman ta zabal zazu

Universidad del País Vasco
Euskal Herriko Unibertsitatea

Konputazio Zientziak eta Adimen Artifizialaren Saila
Departamento de Ciencias de la Computación e Inteligencia Artificial

Contributions to Neural Architecture Search in Generative and Heterogeneous Multi-task Modeling

by

Unai Garciarena Hualde

Supervised by Roberto Santana and Alexander Mendiburu

Dissertation submitted to the Department of Computer Science and Artificial Intelligence of the University of the Basque Country (UPV/EHU) as partial fulfilment of the requirements for the PhD degree in Computer Science

Donostia - San Sebastián, June 2021

Acknowledgements

The amount of pillars in which this dissertation rests is so extensive that just attempting to name them all would probably double the length of the document, so I have had to make a small selection of them. These first lines cannot be directed towards any others than my advisors, Roberto Santana and Alex Mendiburu. Had it not been for their guidance, chances are that this document would have never materialized. I hope that we can keep working together after this milestone. Thank you very much.

I also thank the members of the Intelligent Systems Group, and the colleagues that I have made throughout the four years I have spent here, as they have also played integral roles in this dissertation. I would like to especially thank the people I have spent the most (and best) time with, Borja and Ibai, but also everyone who has step foot in these labs. Thank you all very much. *Si no fuera por estos ratos...*

I also feel the need to thank Professors Penousal Machado and Nuno Lourenço for hosting me at the University of Coimbra, during difficult times due to the global pandemic. That did not, however, prevent them -along with my colleagues, especially Daniel and Pedro- from making my stay there a very enjoyable and productive one.

The part played by John Kennedy, who has thoroughly read and analyzed each phrase written in this dissertation (with the exception of these in this section), and many more left outside it, cannot be overstated. Thank you very much too, Johnny.

While the academic part has been, of course, instrumental to the achievement of this goal, I can't leave my family and friends out from these paragraphs. Thank you very much *Ama*, *Aita*, Ainhoa, Joana, and the rest; and Antón, Haritz and the rest.

As I said, I have had the difficult task of selecting the people who explicitly appear in this section. Nevertheless, if you feel that you should have been here and you are not, I sincerely apologize, and please, take a part of this last Thank You.

I am grateful to the University of the Basque Country for the financial support during the last four years (through PIF16/238 the grant). In addition, this work has been partially supported by the Saiotek (IT-609-13, IT-1244-19, Basque Government), Elkartek (KK-2020/00049, 3KIA project funded by the SPRI-Basque Government) programs, and the TIN2016-78365-R (Spanish Ministry of Economy, Industry and Competitiveness), PID2019-104966GB-I00 (Spanish Ministry of Science and Innovation) projects. We also acknowledge the support of NVIDIA Corporation with the donation of a Titan X Pascal GPU which has been used to conduct part of the experiments reported in this dissertation.

Summary

In the last decade, deep neural networks have risen as powerful computing systems, which, when properly trained, have shown impressive performances in vastly different problems and domains: from classification to optimization, from health to logistics. However, the performance of these models is largely dependent on its structure (the architecture, training hyperparameters, etc.). Currently, in most cases, the process for improving the performance of a neural model in a given task involves the development of another, more complex deep neural network. Multiple repetitions of this process have resulted in the procedure of manually devising the structure of a neural model becoming an increasingly complex assignment.

In this context, researchers and practitioners have recently shown interest in methods which automatically search for structures of neural models. Due to the costly nature of deep neural network training, enormous amounts of computation time are spent on the automatic development of neural structures in order to obtain model configurations which are capable of outperforming the previous best proposal.

In a framework in which the evaluation of every structure counts, developing efficient methods for automated neural structure design could bring two benefits: i) save considerable amounts of computational time and resources (greener computing), and ii) provide better architectures.

That is precisely the final aim of this dissertation. To achieve that goal, we embrace two different problems with very different characteristics. First, we focus on the generative adversarial network, which has produced impressive results in, among others, the area of realistic image generation. Despite this, the automated optimization of its structure has not gathered as much interest as other popular deep neural models, although the results reported in this dissertation suggest that the performance of these models is also strongly dependent on their structure. Secondly, we focus on the heterogeneous multi-task learning problem, a framework with an ample potential, but an equally high complexity level. In this case, not only the efficiency of structural search methods is investigated, but a new model with the capacity to answer to heterogeneous problems and an optimizable structure is also proposed.

In both cases, different types of approaches to the automated search of structures are considered, to cover as many types of optimization algorithms as possi-

ble, always with the same mindset: efficiently finding strong performing structures. Extensive analyses have been carried out in all cases in order to understand the results provided by each approach in each search type. These studies have shown that efficiency gains in structural searches can be achieved by investigating the characteristics of the model and the mechanics of the search algorithms; and that strong relationships between the components of neural structures exist, in the sense that for one component of a network to work properly, it needs to be surrounded by the right components.

Contents

Part II Efficient search of complex DNN structure spaces

Acronyms

ANN Artificial Neural Networks.
ARIMA Autoregressive Integrated Moving Average.

BN Bayesian Network.

CNN Convolutional Neural Network.

DAG Directed Acyclic Graph.
DB Database.
DL Deep Learning.
DNN Deep Neural Network.

FID Fréchet Inception Distance.

GA Genetic Algorithm.
GAN Generative Adversarial Network.
GD Generational Distance.

HC Hill Climbing.
HMTL Heterogeneous Multi-task Learning.

IGD Inverted Generational Distance.

KL Kullback Leibler.

MLP Multi-layer Perceptron.
MMD Maximum Mean Discrepancy.
MOP Multi-objective Problem.
MSE Mean Squared Error.
MTL Multi-task Learning.

NAS Neural Architecture Search.
NASH Neural Architecture Search by Hill Climbing.

NE Neuroevolution.
NM Network Morphism.

PF Pareto Front.
PGM Probabilistic Graphical Model.
PS Pareto Set.

RNN Recurrent Neural Network.

SGD Stochastic Gradient Descent.

VAE Variational Autoencoder.
VALP Vertices, Arcs, Loss functions, and hyperParameters.

I planned each charted course
Each careful step along the byway
Oh, and more, much more than this...

1

Preliminaries

1.1 Introduction

The usage of machine learning (ML) techniques has increased dramatically in the last few decades. While the improvement of the hardware required to run ML models is a fundamental factor, the main facilitator to this change is probably the vicious circle of society discovering new applications of these methods, which leads to further research in the area, ultimately conquering new frontiers and discovering even newer applications [74]. Several iterations of this loop have resulted in an extensive growth of the ML field in both the accuracy of the developed models, and also in the types of tasks that they can cover.

Until relatively recently, improving the performance of ML algorithms consisted of designing increasingly domain-specific methods, which fitted certain problems exceptionally well [74]; e.g., ARIMA models conceived for time series analysis, complex Markov models used for speech recognition, or models that explicitly extracted predefined features from data, used for analyzing images. This trend has, however, experienced a significant shift since the potential of deep neural networks (DNN) came to light.

Discovering the possibilities of DNNs has resulted in a large amount of ML research gradually gravitating towards deep learning (DL), a subfield which is based on DNNs. DNNs are models composed of sequences of basic operations, such as matrix operations (multiplication and addition) and simple non-linear function applications (e.g., sigmoid function), but it has been proven that they possess the necessary modeling power to accurately represent any arbitrary function. This capacity is a consequence of the layer-stacking structure of these models, which, in order to show all their potential, must be carefully designed and tuned. Particularly, the optimization of the parameters of the model (in this work referred to as weights), and finding an adequate neural architecture (the distribution of the basic operations or layers) for the problem at hand. Regarding the first issue, the optimization of the weights, it is usually dealt with by means of the backpropagation technique, which has become the standard way of optimizing the weights of a DNN (although other methods have provided interesting results too). The second one is still a work in progress, as DNNs

of arbitrary complexity can be defined, regarding the number, size, and type of the layers (which ultimately determines the amount of parameters in the model). Adequately defining a complex architecture is not a trivial task, and empirical evidence shows that, in general, more complex architectures have a higher chance of providing a better result than simpler ones.

Over the last decade, increasingly complex structures (which comprise the neural architecture and other required hyperparameter choices) have been proposed in the literature, each outperforming their *simpler* predecessors. The area in which DNNs have arguably had their largest success to date, image classification, serves as a clear indicator of this trend. Taking into account one of the most popular benchmarks of this area, the ImageNet benchmark [28], the top eight performing DNNs have no less than 390M parameters, whereas the rest of DNNs with near state-of-the-art results rarely reach that amount [105]. This trend is not only observable in image-related tasks. The top performing DNN for language modeling reports a large margin over the previous best in the Penn Treebank benchmark [88], both in number of parameters (175,000M over 395M), and perplexity, the standard metric for word prediction (20.5 over 31.3. For context, the 29th model obtained 58.6) [106].

Whereas the application areas in which DNNs found the greatest improvements over other methods were related to target value prediction, generative modeling is a clear example of new horizons recently conquered by neural models. Generative adversarial networks (GAN) [49], and, to a lesser extent, variational autoencoders (VAE) [66], spearhead a large collection of models and methodologies that can be used for generating data from a distribution described by a known set of observations. Boltzmann machines and their variations [117], back-drive (or network inversion) [82], and the recent, prominently emerging, adversarial learning [125] are other examples of neural models which can be used to generate new data. The application domains of these models are multiple and largely different, as they can serve the purpose of, among others, generating realistic images, creating *releasable* data which follows the same distribution as certain confidential items of data, or driving evolutionary searches by providing new individuals for the upcoming generation.

The time consumption of manual structure design increases as so does model complexity, and considering the rate at which DNN complexity is escalating in state-of-the-art neural architectures, this trend could eventually lead to the point in which manually defining DNN models is no longer a feasible task. This fact, when paired with the recent advances in hardware technology which exponentially accelerate the speed of DNN weight optimization procedures, has garnered a lot of attention in the research field of automated search of the DNN structure space. Methods aiming at this goal are commonly referred to as neural architecture search (NAS) algorithms [33].

The problem of automatically devising DNN structures has been tackled from very different perspectives: from those based on evolutionary algorithms (EA) [95], to reinforcement learning [38], differentiable methods [36], or local searches [33]. Due to their capacity to globally explore search spaces, EAs have attracted a larger portion of the researcher attention, hoarding to some extent the research production volume. The synergy level between these two fields led to coining the Neuroevolu-

tion (NE) term, which encompasses all the evolution-inspired algorithms. Recently, however, local-oriented search algorithms have arisen as a competitive alternative [34, 136]. Although they successfully achieve the goal of finding *good* architectures with minimal human intervention, NAS methods are not flawless. One particular aspect that is commonly held against them is the computational effort they require, and thus any efficiency gains that can be made in that field are welcome. This aspect is especially important in DNN models that are inherently complex, of which multi-network models (such as GANs and VAEs) are a particularly clear example.

1.2 Outlook of the dissertation

Despite the elevated computational cost requisites of NAS algorithms, the quality of the results they produce totally justifies the required investment, hence a lot of computational resources are spent on them. While the different NAS methods reach their goal, an abundance of information which describes the progress of the NAS procedure is also generated, and more often than not, not included in any kind of report. In this dissertation, part of the effort is devoted to addressing the efficiency issues of NAS algorithms by exploiting the *residual* information of preceding NAS runs. Due to the elevated number of structures that are commonly evaluated in NE runs, we focus on this particular set of NAS algorithms.

To achieve the goal of making the NAS runs more efficient, we investigate two different approaches with the same goal: smart searches in which selections of the most appropriate structural variation operators or models are made, thus avoiding costly evaluations of potentially underachieving DNN structures. In the first approach, the efforts are directed exclusively towards GAN models, which serve as a useful use case, as these models are complex and highly used by the generative modeling community. In the second approach, the knowledge acquired in the previous one is extrapolated to a more general domain: the VALP. The VALP is a multi-network model, oriented to multiple heterogeneous tasks, which has very few restrictions in terms of the architecture. Because of this characteristic, the search space of VALP structures includes current state-of-the-art DNN models. This implies that any methodological proposal which is able to operate as expected in this *complex* search space (the final contribution of this work) is likely to perform well in any other (limited to a specific problem) search domain.

This dissertation is organized in two parts: Analysis and contributions to NE for GANs, and contributions to the NAS field.

The first part of the dissertation is divided into Chapters 3 and 4. The first one proposes a NE procedure in which GANs are evolved for accurately generating Pareto set approximations. After that, an advanced analysis of the introduced NE procedure is performed, demonstrating the validity and transferability of the approach and showing the potential of the information generated during the procedure. In Chapter 4, two different approaches for exploiting that information are presented. They are used for improving the efficiency of totally different NAS searches for GANs.

The second part of the dissertation also consists of two chapters, Chapter 5 and 6. The first one introduces the VALP, a model without a defined architecture, formed by multiple DNNs which can be interconnected in different ways. These characteristics make them suitable for solving multiple tasks of different types simultaneously, as the sub-DNNs within the VALP can provide outputs of different kinds. In the second one, the contributions to the NAS field are presented. These are achieved by applying the conclusions drawn in the previous part to a more general search space, that of the VALP. The VALP search space is considerably larger than those of other DNN models, as each sub-DNN can be arbitrarily complex.

Finally, Chapter 7 draws the general conclusions of the dissertation and points out potential future research lines, and Chapter 8 compiles the publications and submissions derived from the research carried out in this dissertation.

2

Background

2.1 Deep Neural Networks

Artificial Neural Networks are computing systems whose structure and behavior tries to mimic the brains of animals. It can be seen as a collection of nodes (called neurons), and a set of connections between these nodes. Each node processes a signal (a vector of real numbers) and sends this signal towards other nodes across the connections.

Neural models have seen their popularity rise and fall since they were proposed in the mid-20th century. The perceptron is usually considered as the earliest predecessor of the modern definition of DNNs [112]. As the modeling capacity of a single perceptron did not exceed the limits of a linear estimator, the potential of ANN models remained latent. However, by combining the linear separator properties of several perceptrons, it is possible to model any function by defining one perceptron for each of the bounds of the target function.

The employment of multiple perceptrons to represent a complex function still requires the intervention of a human to combine and interpret the decisions made by the set of perceptrons (from this point on, called layers of neurons). By placing another layer of neurons which receive the outputs of the previous ones, this second layer can automatize this task. This model starts to resemble current DNNs, and it is able to represent any arbitrary function by itself [26, 58]. This potential, however, probably would not have been fulfilled without the definition of an efficient technique for weight optimization, i.e., the backpropagation algorithm [135].

Current DNNs are commonly devised as multiple layers of neurons, each of which can be activated by a non-linear function. In each layer, simple matrix operations (which vary depending on the DNN architecture type) are performed to the incoming features. It is theorized that the deeper the layer is located in a DNN, the higher the abstraction of the representation of the data expressed by the neurons of that layer. By adding the non-linearity to each layer, the DNN is provided with the modeling power necessary to represent any arbitrary function. This posed an additional obstacle to the realization of the potential of DNNs. Because depth is one of the main sources of modeling power of neural models, it was not until the last

decade, in which the evolution of computational capacity allowed increasingly complex models, that the popularity of DNNs reached its current state.

A visual representation of a two-hidden layer classification DNN can be seen in Figure 2.1. The *raw* features (x_1, x_2, x_3, x_4) of an observation are transformed in each layer, becoming more and more abstract and rich. The values in the final layer can be interpreted as the probabilities of the input belonging to each class.

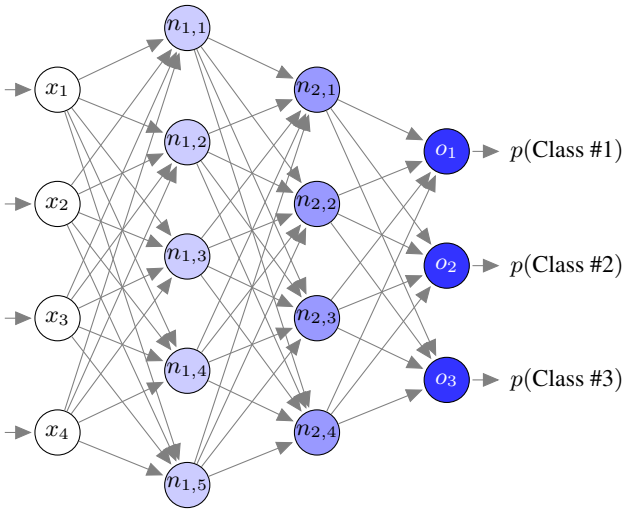


Fig. 2.1: Schematic representation of a DNN for a three-class classification problem. An observation (represented by its features) is placed in the input of the network and is subjected to a list of matrix transformations as well as a possible application of a non-linear activation function. Each arrow represents one scalar multiplication. The first index in the neurons is the layer they belong to. Each neuron n performs a summation of all the incoming values. The output layer (in the general case) consists of one neuron per possible class, each being interpreted as the probability of the observation placed in the input belonging to a certain class (if activated with a softmax function).

2.1.1 Architecture types

The rise to prominence of DNNs came when it was discovered that the performance of neural models was heavily dependent on the architecture of the model. If the adequate neural architecture is chosen to deal with a given problem, the capabilities of the DNN to cope with that challenge have proven to be competitive with different models in different areas [124, 19].

Broadly speaking, DNNs can be categorized in one of three classes according to their architecture. Firstly, the *simplest* layout for a DNN, the multi-layer perceptron

(MLP) [26]. These DNNs make no assumption about the data they model. They are commonly organized in sequential dense layers of neurons, which means that every neuron in layer l receives a value from every neuron in layer $l-1$, and the value they compute is provided to every neuron in layer $l+1$. This sequential arrangement is known as feedforward DNN, as information only flows in a forward direction. The DNN displayed in Figure 2.1 corresponds with an MLP. The transformations performed in these DNNs correspond with straightforward matrix multiplication and addition, and the application of a simple, non linear function. In mathematical notation, an MLP layer can be expressed as $l_{i+1} = f_i(l_i w^i + b^i)$. l_i represents the i-th layer, w^i the i-th matrix of weights, and b^i the vector of biases. Finally, f_i stands for the activation function of the i-th layer. w^i and b^i are the optimizable parameters of the model (commonly optimized using backpropagation, and in this work referred to as weights), and f_i are hyperparameters of the model. The layers l_i are composed of the neurons in that layer, $l_i =< n_{i,1}, n_{i,2}, ...n_{i,h} >$, where h is the number of neurons in the corresponding layer. As an example, following the architecture described in Figure 2.1, $l_2 =< n_{2,1}, n_{2,2}, n_{2,3}, n_{2,4} >$, and, for example, $n_{2,1} = f(\sum n_{1,i} \times w^2_{i,1} + b^2_i)$. f could be any simple, non-linear activation function, $ReLU = max(0, x)$ being a popular choice. The design of an MLP architecture can be commonly viewed as a hyperparameter setting scenario, since it *simply* consists of defining the number of layers and neurons within them, and the activation functions.

Although not making any assumption about the problem structure makes them suitable for a wide range of applications, their structure must be carefully designed for each problem if competitive results are expected. The other two architecture types, specially defined to fit data with certain characteristics, have been the subject of extensive research, as they are able to provide top end results when applied in the right circumstances.

Convolutional DNNs (CNN) [69], the type that, to a great extent, brought DNNs back into prominence, are based on layers specialized in extracting abstract features from spatial data, e.g., images. This special type of DNNs, instead of dense layers as in the MLP, are composed of convolutional and pooling layers, which consist of kernels that analyze sets of contiguous data positions. These kernels act as sliding windows over the data, and, by producing similar results when analyzing similar contiguous positions, are able to detect patterns described by the pixels in the input (or neurons in the previous layer, in the case of the hidden layers). The different transformations of the data flowing through a sequence of convolutional layers is commonly referred to as *blob*. Multiple types of kernels exist, each of which is defined by a different operation over the blob in the previous layer. A standard convolutional layer consists of several filters which perform a scalar multiplication of the matrices and also have a bias term, similar to the MLP. The pooling layer consists of simpler kernels, which just select the maximum (or mean, depending on the pooling type) value of the values covered by the kernel. This way, what in the first layer was a set of pixels showing a sudden color change, can be represented as the direction of an edge in the following one. A representation of how a convolutional filter works can be seen in Figure 2.2.

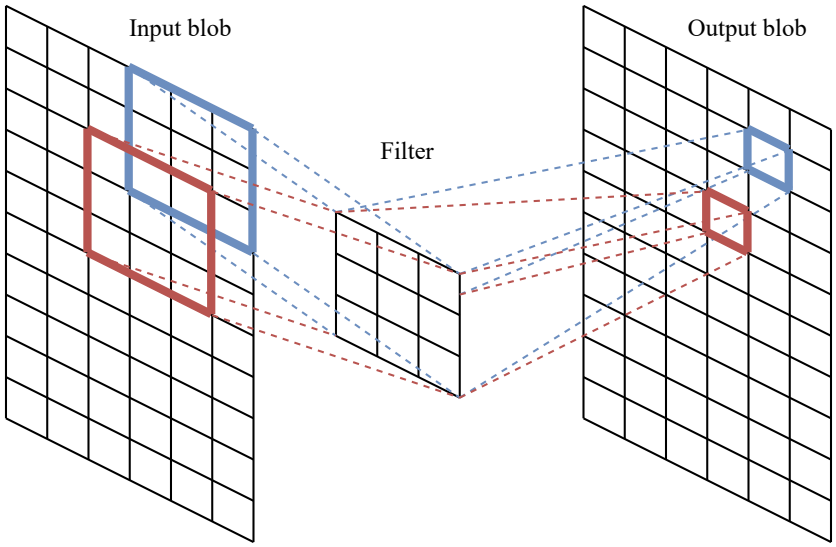


Fig. 2.2: Example of a single filter performing over a 2D blob. In this figure, we can see a visual representation of one convolutional filter transforming the input blob of data into the output blob.

Creating specialized kernels, on top of simplifying the decision process when it comes to performing the final task (e.g., classification), results in a reduced number of parameters when compared to MLPs, which would need enormous amounts of parameters to address the spatial relation between pixels in a similar manner to CNNs. Technically, CNNs can also be seen as regularized MLPs. The architectures of state-of-the-art CNNs have experienced a recent spike in terms of complexity, when compared to the traditional sequential design. Components such as skip connections [52] or performing the same prediction multiple times with the same model [124], have made the architecture design a non-trivial matter.

Recurrent DNNs (RNN) [57] are specially designed to deal with sequential data. They can be composed of layers similar to those in an MLP, only that, in this case, they are not arranged in a sequential manner. The main characteristic of these DNNs is that they include recurrent connections, this is, the information does not only go in the forward direction, as in an MLP or a common CNN. The information in certain layers is fed to a previous state, so that the final prediction made by the network is done within the context of a sequence of observations it has been fed. A schematic representation of an RNN is shown in Figure 2.3.

Similarly, the complexity of effective recurrent architectures [57] makes their manual design a complex job to carry out.

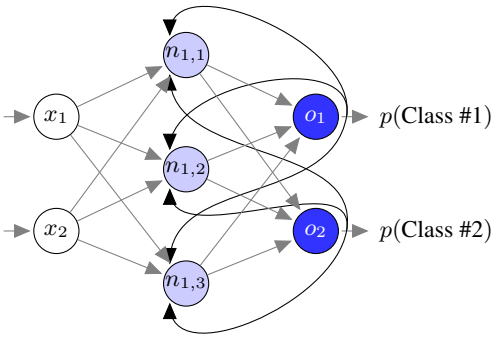


Fig. 2.3: Schematic representation of a minimalist RNN of a single layer, in which the output of the network is redirected to the neurons in the hidden layer.

2.1.2 DNN weight optimization procedure

The primary source of the modeling power of DNNs, their depth, makes the application of *traditional* weight optimization techniques, such as linear least squares, unfeasible. Because of this, and despite the development of other strategies with the same goal providing competitive results, DNN weight optimization is commonly achieved using backpropagation [75]. As can be seen in the explanation of the MLP in the previous section, the operations performed by each layer of the MLP can be formulated as a function, and all of them are derivable. Measuring the quality of a DNN at a given task is commonly achieved by employing a loss function over the final product of the DNN (the sequential application of the functions described by the layers in the network), which is also derivable.

Basically, the backpropagation algorithm consists of applying the derivative chain rule multiple times, once per layer in a DNN. The derivative of a loss function desired to be optimized can be computed with respect to input of the DNN by starting from the last layer, and working backwards, one layer at a time. By performing this procedure, the gradients of all the parameters in the DNN are *propagated* backwards, which allows the application of gradient descent (GD) algorithms (and variations of it) aiming at reducing the value of the loss function. Intuitively, GD algorithms use the gradients of a function, the value of which function is desired to be minimized. They perform *subtle* changes in the parameters of said function in the opposite direction of the gradients, resulting in a (commonly) reduced decrease in the function value. By applying this process iteratively, GD algorithms can reach a local minimum.

The case for CNNs and RNNs is similar to that of the MLP. The kernels in a CNN are also basic derivable operations, and therefore, capable of being trained (i.e., getting its weights optimized) using the backpropagation technique. RNNs can also

be trained via backpropagation, although the methodology varies slightly. The RNN has to be *unfolded* several times, simulating a set of identical DNNs, where the previously recurrent connection now connects two adjacent DNNs. This way, a RNN can be interpreted as a feedforward DNN, a requirement for applying backpropagation.

Taking into account the large number of parameters that DNNs usually have, the more examples (in the data) available to train them, the better their performance is. This, in turn, poses the problem of a high computational demand when computing the gradients for the predictions of a DNN to the examples of a database (DB). Because of this, instead of using a standard version of the GD algorithm, stochastic GD (SGD) is commonly used. This technique consists of sampling subsets of examples from the DB, and performing a GD step exclusively taking into account the gradients computed using those examples. This process is repeated, iterating over the whole dataset so that all examples are shown to the model at some point. Several variants of SGD have been developed with multiple inspirations, such as adaptive estimation of first-order and second-order moments (Adam) [65], or weight-exclusive regularization (Adagrad) [29].

2.1.3 Muti-network models and areas of application

Although the main source of popularity for DNNs comes from the usage of CNNs for image-related prediction tasks -primarily classification and segmentation-, they also have excelled in other areas. A great example of a research field recently conquered by DNNs is that of generative modeling. Although restricted Boltzmann machines were invented back in the last century, it was not until recently that DNN-based models, VAEs [66] and especially GANs [49], proved to be a significant improvement over previous generative models. Both of these models are composed of two sub-DNNs which fulfill different complementary roles in their corresponding frameworks. Not only have DNNs been able to provide excellent results when it comes to data generation, but they can also serve as a basis for a state-of-the-art evaluation metric for generative models: the Fréchet inception distance (FID) [54].

Another area to which DNNs have been applied is that of multi-task learning (MTL) [13]. Taking advantage of the capacity of DNNs to work with large pieces of data, several approaches that attempt to use the same DNN to solve multiple separate problems have been presented recently [141]. When these problems are related in some way, e.g., classifying images of characters from two separate alphabets, it is argued that the initial layers of the DNN can perform the same task of extracting the *low-level* features (i.e., edges) indistinctively for both tasks. Next, the inner layers are assigned the task of *recognizing* the characters starting from the prebuilt features. The advantage of using a single DNN to model different DBs is twofold. Firstly, the reduction of computational effort, as a single model is faster to train than multiple models. Secondly, it has been argued that the tasks regularize each other, as a model cannot overfit a single of the multiple problems it is modeling, thus making it stronger in terms of generalization capabilities.

Moreover, the moldability of DNNs allows them to create different *branches* of layers, where each branch can be seen as a subnetwork fitted within a larger model,

giving each of them the capability to make a prediction for separate tasks, simultaneously. This way, not only can a DNN perform similar tasks (as in the previous example, classifying images from DBs with similar characteristics), but it also can perform different types of predictions about them (e.g., classifying images of cars according to their brand and, at the same time, predicting their horsepower).

In the case of these more complex models, apart from the architecture itself, there are more components (which will be covered in due course throughout the dissertation) that need to be fixed before starting the learning phase. We refer to all the necessary model design choices as the structure of the model.

2.1.4 Neural architecture search

Most of the extensions to DNNs that make them so powerful in any domain they perform come at the cost of increasingly complex architectures. *Simple* models designed for performing a single prediction task are becoming increasingly complex. This effect is more sharply suffered by models which are inherently complex, such as MTL or GANs. While automatically devising DNN architectures has been an interest of the DNN community for a very long time [120], it has currently become a norm when it comes to achieving improvements over the best known architectures. The fact that in the last decade significant advances have been made on computer hardware, particularly in graphic processing units, has meant that the interest in these neural architecture search (NAS) algorithms has increased. Despite these improvements, computational complexity is still one of the main concerns of practitioners, as the improvements in hardware are usually followed by increases in the complexity of the newly proposed models.

The approaches at the NAS problem have been multiple and diverse. Local searches, methods which can keep the number of evaluations within reasonable margins, have served as a basis for different techniques. Differentiable approaches, such as reinforcement learning-based [146], or straight-up gradient descent [36] have also been successful. Recently, other kinds of local searches have also reported competitive results. Hill climbing (HC) approaches, which rely on the application of operators that modify the architecture, have also resulted in state-of-the-art outcomes in image classification tasks [136]. By adequately defining the search space, and because of the existence of multiple architectures which can provide top results, and how these are distributed within the search space, these methods can be effective while keeping the number of evaluations low.

However, it has been another operator-based search type which has attracted the most research volume, evolutionary algorithms (EA).

2.1.5 Optimization and evolutionary algorithms

EAs [32] are based on the natural evolution of species. The first phase of a common EA consists of a set of random individuals, called a population, being evaluated. Each individual, $\mathbf{x}$, represents a solution to a given problem, f, and its *fitness* is estimated according to how good that solution is to the problem, $f(\mathbf{x})$. As in natural

evolution, the fittest ones survive, and are eligible for the next population. Mimicking the mechanisms of nature to advance the evolution of species, an EA makes use of different kinds of methods to alter the individuals (or generate new ones altogether) strong enough for advancing to the following population. These modifications, of which a popular example are mutation operators in genetic algorithms, affect the fitness of the individual towards the problem. A positively modified individual is likely to continue to the next population, while one that received a negative modification is not. This way, the population advances by having increasingly stronger individuals, the best of which is considered the outcome of the whole process.

2.1.6 Multi-objective optimization

The difficulty of optimization problems, as the one defined above using $\mathbf{x}$ and f, can vary in different aspects. One such aspect if the composition of the problem function f, depending on which, the optimization problem can have multiple (potentially conflicting) objectives to be optimized simultaneously.

A multi-objective problem can be defined as the optimization of a vector of functions $f(\mathbf{x}) = (f_1(\mathbf{x}), ..., f_m(\mathbf{x}))$, where $\mathbf{x} = (x_1, ..., x_n)$, m is the number of objective functions and n is the number of decision variables. Commonly, it is not possible to find a single $\mathbf{x}^*$ solution which is optimal regarding all f_m functions, as each function evaluates the solutions differently. Therefore, algorithms which deal with this kind of problems usually provide a set of solutions. This set is composed by *non-dominated* solutions [90].

Assuming a minimization problem, an objective vector $f(\mathbf{x})$ dominates a vector $f(\mathbf{y})$, if and only if $\forall i \in \{1, ..., m\}, f_i(\mathbf{x}) \leq f_i(\mathbf{y}) \wedge \exists i \in \{1, ..., m\} : f_i(\mathbf{x}) < f_i(\mathbf{y})$. A vector $f(\mathbf{x})$ is non-dominated if there is no $f(\mathbf{y})$ that dominates $f(\mathbf{x})$. If $f(\mathbf{x})$ is non-dominated, $\mathbf{x}$ is Pareto optimal. The set of Pareto optimal solutions is the Pareto set (PS), and the projection of these solutions in the objective space is the Pareto front (PF) [22].

2.1.7 Neuroevolution

The search space for DNN architectures has been explored by different variations of EAs a significant amount of times, to the point that these approaches have been categorized in a research subfield of their own, neuroevolution (NE) [95]. Moreover, EAs have been automating the structural search of neural networks for over two decades, which has produced a wide range of approaches, each with their own particularities.

One key aspect of NE algorithms is the codification of the individuals. In neuroevolution, codification can be one of direct or indirect [136]. A direct codification implies that the individual in the EA contains the DNN itself, that it needs no process before being evaluated. Indirectly encoding DNNs in an individual consists of evolving a more *abstract* representation of the DNN, a representation which needs some kind of preprocess before it can be evaluated. For example, an individual limited to specifying the weight initialization function would need a decoding phase before

being evaluated. In this example, this phase would consist of using the function to initialize the weights and training them (e.g., by using backpropagation).

Initially, with the more *simple* neural networks, direct encoding was the easier and more effective method to design individuals. However, as complexity increases, this strategy has become more and more difficult to carry out, and indirect encoding has become the preferred choice. One key aspect that is commonly left out when indirectly encoding individuals are the weights of the network, which are required to be computed before evaluating the DNN. One way to do so, is to randomly initialize and train them using backpropagation, which is the main reason for the costly nature of NE algorithms.

The traditional approach to NE commonly considers relatively low-parametrized networks both regarding the number of layers and the amount of neurons in them. However, as the hardware supporting DNNs has advanced, these methods have shifted from performing low level modifications, e.g., the addition of one neuron or connection [121], to more complex operations, such as the concatenation of full neural cells to the DNN [87]. This second kind of evolution has proven competitive against hand-crafted structures, and is the most popular approach considering the amount of recent work devoted to it. Currently, these two scopes of variation operators are known as macro (altering the general structure of the neural model) and micro (modifications limited to the small cells) searches [136].

Part I

Analysis of neuroevolutionary techniques for DNN generative models

3

Analysis of the transferability and robustness of evolved GANs

3.1 Introduction

Learning how to generate data samples that resemble real ones can be convenient for many real-world scenarios; for example, to avoid violating privacy and confidentiality of certain data when this is released [123], or when there are not enough training instances for a ML procedure [5]. In this context, DNN based generative models [48, 74] have shown good results, and proved their usefulness. As it has been stated in the previous chapter, among other aspects, the number of hidden layers (or depth of the DNNs) has proved to be key in enhancing the performance of DNN models in different domains, such as complex image analysis problems [52, 69], or generative modeling [49].

In the last few years, several efforts for automatically designing the structure of DNNs have been devoted, particularly using EAs [80, 85, 95, 122]. Most of these works focused on evolving CNNs [69]. In most of these approaches, the object of evolution is the set of hyperparameters of the network, e.g., the activation functions, or the layer/kernel size. Sometimes, the weights themselves are included in the evolutionary procedure, whereas the loss functions, generally used to train the model, are hardly ever included. Despite the success EAs have had in this area, they are a costly approach, as they manage populations of DNNs. Each of these DNNs has to be evaluated, and this commonly involves training networks. Therefore, agility and transferability are two key aspects to take into account when designing NE algorithms. This is especially true for multi-network models, as the size of the search space increases with each DNN added to the model. GANs [49] are one popular instance of multi-network models which have arisen as one of the top performing DNN-based generative models. Their impressive results have garnered them great popularity, mainly in the area of realistic image generation [110]. Despite being highly regarded because of the quality of their generations, GANs are not flawless. The first issue to address when using GANs is to design an adequate structure for the model. This design implies making a number of architectural choices similar to those present in other DNNs (e.g., the number of hidden layers for the networks composing a GAN: the discriminator (D) and the generator (G), the activation functions, etc.).

Other aspects of the model structure, such as the training methodology, also need to be properly designed. For a training procedure that suits a given model structure, D and G may need having their respective parameters updated with a different frequency [131]. This synchronization parameter must also be optimized. Furthermore, as it has been shown in recent works [6, 103], loss function alternatives that optimize different criteria can be more effective than the traditional GAN training loss function. In domains where this choice is not clear, the selection of the loss function is another aspect to be optimized.

An inharmonious relation between the model structure, the synchronization policy, and the loss function could lead to poor results, such as a non-converging scenario, or a generator with some sort of malfunction (e.g., being unable to reproduce the whole original distribution; an effect known as mode collapsing [15]). We comprise all the mentioned components which affect the performance of a GAN (from the architecture of the network to the loss function and the update frequency) into the *GAN structure* term.

Designing GAN structures is not a trivial task because of the many choices to be made. In consequence, we resort to a NE algorithm to avoid the necessity of manually designing GAN structures capable of generating realistic samples while evading the mode collapsing problem. As it has been stated before, the success achieved by NE algorithms comes at the expense of expensive hardware being busy for long periods of time, due to the high computational demands of this kind of algorithms. In order to palliate this drawback, we propose a multi-objective approach, aiming at the optimization of both performance and complexity. Additionally, evolving GANs which can be trained and tested faster also results in a more agile NE process.

We recognize that the problem of designing an adequate structure for generating data with certain characteristics can be considerably eased by reducing the number of choices to be made; those being any of the activation or initialization functions, number of neurons in a hidden layer, etc. We theorize that the decisions made for GAN design in this problem can be extrapolated to others. This would ease future GAN designs in any problem domain.

With this premise in mind, we perform an a posteriori analysis of the GANs evolved by the NE algorithm with two main goals. First, to test to what extent the results obtained are transferable between problems. Secondly, assuming that the first hypothesis is fulfilled, to detect characteristics frequently found in the best performing GANs. This would reduce the difficulty of the GAN design problem, regardless of whether it is manual or automatic, by reducing the number of choices to those *chosen* by the NE algorithm.

The NE approach presented in this chapter is based on a genetic algorithm (GA), an EA type which main characteristic is the employment of mutation, crossover, and selection mechanisms, mimicking the process of natural selection [97].

In our GA, GANs are indirectly encoded by means of lists that describe the parameters that define the generator, the discriminator, and the training procedure. Indirect encoding allows an easier application of the genetic operators, at the expense of not taking advantage of previously learned weights. Most importantly, this approach does not include the number of inputs and outputs of the GAN, which depends on

the data. Therefore, a network structure that has been evolved for a specific data distribution can be tested on a different problem. This allows the evaluation of the transferability of the evolved structures across problems and dimensions, determining the level of generalization of the described methodology.

We use the problem of approximating the Pareto set (PS) of bi-objective functions as a testbed to investigate a number of issues relevant to GANs. First, this enables us to evaluate the quality of a GAN, unbiased and automatically, at generating data points. This is achieved by computing a distance in the objective space between the approximated and the original points known to be in the PS, i.e., a distance between the Pareto front (PF) [22] and the approximation made by the GAN. This results in a computing resource-saving move, as usually the number of objectives is significantly lower than the number of decision variables. Secondly, we link the GAN problem of mode collapsing with the question of producing a homogeneous covering of the PF. In this chapter, we initially make the assumption that a high quality covering of a PF implies a GAN generator capable of generating an accurate and well-distributed PS approximation. We test the veracity of this assumption later in the experimental section.

In summary, it can be said that, in this chapter, a two-level PS approximation is made. First, in order to assess the quality of a GAN, the PS approximation provided by the model is compared to the real PS. Secondly, the EA which optimizes the GANs also takes advantage of PS approximations balancing complexity and accuracy of the GANs.

Finally, the level of transferability of the evolved GANs is tested in three different levels, the last of which consists of a completely different problem domain to the first two. For the first level, GANs evolved for certain PS approximation problems are tested with different problems from the same suite, whereas the second level does so for problems with a larger number of decision variables. For the third level, the generalization capabilities of the structures are tested in another problem domain, the Gaussian mixture approximation problem [93, 130].

This chapter presents the following contributions: 1) We introduce a NE algorithm for the fast evolution of highly-flexible GAN structures. 2) We propose the usage of the PS approximation problem as a testbed for evaluating the approximation capacity of GAN structures. To that end, a metric applied to the projection in the objective space of the approximation made by the GANs is employed. 3) We show that the proposed NE method is scalable, and leads to the fast evolution of robust and accurate generators. 4) We show that the evolved structures exhibit some level of transferability between both PS approximation problem scales and domains, and to a different domain; in this case, the Gaussian mixture approximation problem. 5) We analyze the results of the NE algorithm and assemble a collection of good practices to be followed when designing GANs.

The chapter is organized as follows: We introduce the GAN model and its main components in Section 3.2. The elements of a GAN that can be subject to modification are detailed in Section 3.3, where the representation and the evolutionary operators used to evolve GANs are also introduced. Pareto front approximations and the metrics used to evaluate GANs are presented in Section 3.4. Section 3.5 describes the

Gaussian mixture approximation problem. Work related to ours is described in Section 3.6. The experiments and analysis that validate the NE approach are described in Section 3.7. The conclusions extracted from the experiments, and some topics for future research are presented in Section 3.8.

3.2 Generative adversarial networks

A GAN [49] model, as following its original description, is composed of two networks; a generator G and a discriminator D. The goal of the generator is to create samples that look as similar as possible to examples from the available data, in an effort to fool the discriminator. At the same time, the discriminator tries to correctly discern the examples found in the original data from the samples produced by the generator. A schematic representation of the GAN structure is shown in Figure 3.1.

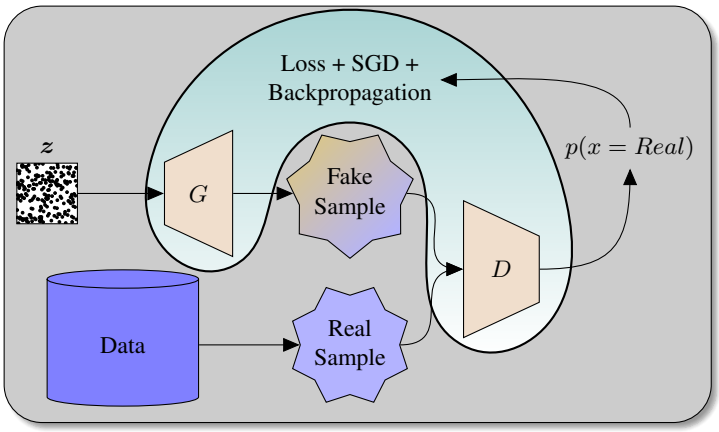


Fig. 3.1: Schematic representation of GAN. The generator network G receives a random noise input for transforming it into a sample as similar as possible to the real ones. The discriminator receives samples both from the original dataset and the generator and attempts to distinguish them apart. Finally, the weights are updated in both networks depending on the ability of the discriminator. Examples correctly classified affect the weights in the generator, whereas the misclassified ones serve for improving the discriminator.

The domain in which the GAN is going to be applied can determine the choice of the DNN architecture type. For example, when used for generating realistic images, convolutional and transposed convolutional layers are applied in the discriminator and the generator respectively. For general domains, MLP architectures can be used.

The weights of the generator are denoted by θ_g, and the generator's distribution over data $\mathbf{X}$ as p_g. $\mathbf{z}$ stands for the input random noise sampled from a latent variable that G receives. This leads to the generated samples, $\hat{\mathbf{x}}$. Formally, $G(\mathbf{z}, \theta_g) \rightarrow \hat{\mathbf{x}}$.

Similarly, the discriminator parameters are represented with θ_d. The discriminator can receive as input true $\mathbf{x}$ or fake samples $\hat{\mathbf{x}}$ and it outputs probability values indicating how likely the samples are to be observations of the original data.

Equation 3.1 captures these two opposing goals:

$$\begin{aligned} \arg\min_{\theta^G} \max_{\theta^D} V(D,G) = \mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})}[log(D(\mathbf{x}))] + \\ \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}(\mathbf{z})}[log(1 - D(G(\mathbf{z})))] \end{aligned} \tag{3.1}$$

where $\mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})}$ and $\mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}(\mathbf{z})}$ represent, respectively, the expectations with respect to the original data distribution p_{data} and the latent variable distribution $p_{\mathbf{z}}(\mathbf{z})$ [49].

A successful training of the model would result in the *fake* samples created by the generator resembling real ones.

This generative adversarial approach is just one example of how a GAN can be trained, but it could be a sub-optimal approach. The authors of [103] show that the principle on which GANs are based can be generalized to arbitrary f-divergences [100]. Statistical divergences are functions used to estimate how dissimilar two distributions are, of which one of the most widely known instances is the Kullback-Leibler divergence measure (KL) [71].

The original GAN loss function, presented in Equation 3.1, corresponds, in fact, to the approximation of the Jensen-Shannon divergence [49]. However, it is possible to make other choices in terms of divergence functions within the general GAN scheme, each one determining one particular way to estimate the dissimilarity between fake and true samples. In [103], the authors derive different divergence metrics for GANs, each having its own properties, which could make them more appropriate for different domains. The authors show that, when the generative scheme is misspecified and does not contain the true distribution, the divergence function used for estimation has a strong influence on the model being learned. Therefore, we consider the divergence metric itself to be an optimizable element of the NE algorithm.

3.2.1 GAN drawbacks

The distribution of real-world data is usually within a multimodal space, where regions with a higher probability than others exist. An ideal generator should perfectly reproduce this distribution, avoiding the omission of areas with lower probability. However, GANs tend to be unsuccessful in this matter [113]. One of the main goals of our method for GAN evolution is to guide the NE search towards models which evade this issue, known as mode-collapsing.

A simplistic example of mode collapsing can be visualized in Figure 3.2. In it, we can observe a true distribution with two prominent modes, represented with a continuous blue line, and two approximations. The first approximation, the one represented with an dashed orange line, would represent an approximation collapsed

to a mode. Mode collapsing affects particularly to GANs because the discriminator cannot afford to classify as *fake* all observations in the primary mode ($x = 0$), as it would discard many original observations. Generators *realize* this circumstance, and take the *shortcut* of learning the most prominent mode of the whole distribution. The dotted green line represents a more accurate approximation compared to the orange one, as it captures both modes of the data.

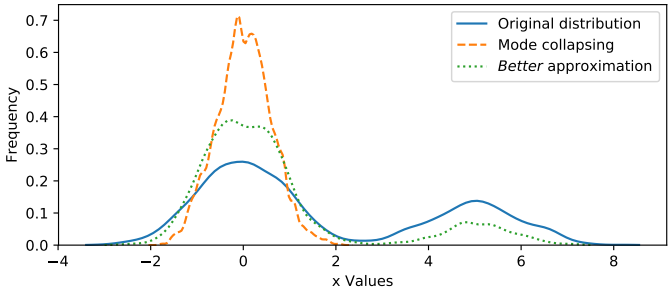


Fig. 3.2: Mode collapsing example.

Depending on several factors (e.g., network structures, problem domain, complexity of the variable distribution to be captured, etc.), the GAN could benefit from training one of the two networks more often than the other. However, setting the frequency in which θ^D and θ^G should be updated in advance is not trivial. In this chapter, we add one *loop* parameter to G and D which indicates the number of times that θ^D and θ^G are updated in each training iteration. If the loop parameter is 2 for D and 3 for G, θ^D will be updated twice in each iteration, whereas θ^G will be updated three times. This extra parameterization allows the exploration of more flexible training schemes.

In this chapter, both networks in the GAN are based on the MLP architecture, because of its great flexibility in terms of modeling capabilities. In contrast to other types of neural networks (e.g., CNNs), the common MLP has a single type of layer (fully connected layers), and the model does not make assumptions about particular dependencies between the variables. In other networks like CNNs, convolutional operations are commonly applied under the assumption of spatial dependency between variables. The employment of the MLP architecture results in two main advantages. First, it is able to deal with the PF and Gaussian mixture approximation problems without constraints or assumptions about the dependencies between their variables. Secondly, it results an easy GAN codification scheme, which helps the NE algorithm when exploring a large variety of GAN structures. Nevertheless, the methodology developed in this chapter can also be adapted and applied to other architectures, such as CNNs, when the problem so requires.

GAN training is performed with the common approach of using a combination of backpropagation and stochastic gradient descent, which makes use of a loss function.

All the experimental parts of this dissertation make use of the Adam gradient descent algorithm [65] with an initial step size of 10^{-4}.

3.3 Evolving GANs

In this section we introduce the approach designed for evolving GANs and provide details on its implementation. Among the GAN components, we consider:

1. Structures of the generator (G) and the discriminator (D). This includes, for each of them:
   a) Number and size (amount of neurons) of the hidden layers.
   b) Activation function to be applied after each layer.
   c) Weight initialization function for each layer.
2. The probability distribution the latent variable follows.
3. G and D update frequencies.
4. Divergence measures used to train the model.

The NE approach is able to vary these four components. The gradient optimization scheme and the global parameters (e.g., batch size, or learning rate) of the learning process were not included in the evolutionary procedure to maintain fairness across the different combinations. For example, given the relatively reduced training ($1,000$ epochs) the models receive in our method, a model trained with a smaller initial step size (or batch size, for that matter) would be at a clear disadvantage.

The maximum values for layer size, number of hidden layers and loop are parameters of the algorithm (in this case, 50 and 10, and 5, respectively). We allow layers of the same network to be activated by different functions (7 possibilities in total), as well as to be randomly initialized by three different distributions. Eight different divergence measures [7, 49, 103] were used as possible loss functions. Table 3.1 shows the assignment possibilities for the evolvable components of GANs.

Latent variable	**Activation function**	**Loss function**	**Weight initialization**
`Uniform`	`Identity`	`Standard Divergence`	`Xavier`
`Normal`	`ReLU`	`Forward KL`	`Normal`
	`Elu`	`Reverse KL`	`Uniform`
	`Softplus`	`Pearson` χ^2	
	`Softsign`	`Squared Hellinger`	
	`Sigmoid`	`Least squares`	
	`Tanh`	`Modified standard`	
		`Wasserstein`	

Table 3.1: Assignments options for the GAN components modifiable by the NE algorithm.

As we can see, there are several decisions to be made before training the model, and for each of these decisions, several options are available. An exhaustive search is

not feasible for this problem, given its large dimensionality. Even though intelligent searches pose themselves as a great candidate to deal with this issue, the problem would be alleviated by bounding the search to certain promising areas known to contain *good* GAN configurations.

The GA proposed in this chapter for evolving GANs uses a list-based encoding and employs genetic operators that operate over these lists. Each network included in a GAN is encoded by one list of parameters, which contains the description of the layers and loop parameter. Each GAN requires, in addition, a set of model parameters M_{GAN}, which comprises the loss function and the prior distribution. When an individual is evaluated, the GAN is constructed from scratch according to the description, and trained from weights randomly initialized depending on the initialization function of each layer as the starting point. Finally, using a fitness function which will be introduced later, the quality of the model is estimated.

3.3.1 Operators and algorithm design

The GA designed for evolving GAN structures uses a crossover and a mutation operator. Given two parents $P^1 = (G^1, D^1, M^1_{GAN})$ and $P^2 = (G^2, D^2, M^2_{GAN})$, the crossover operator creates two offspring $O^1 = (G^1, D^2, M^1_{GAN})$, and $O^2 = (G^2, D^1, M^2_{GAN})$. This operator preserves the integrity of M^1_{GAN}, M^2_{GAN}, G^1, G^2, D^1, and D^2.

The mutation operator consists of the application of one randomly chosen function among these eight possibilities:

- layer_change: randomly change the amount of neurons in a random layer of one random network (G or D) of a GAN.
- del_layer: delete a random hidden layer from a random network of a GAN (in case it has more than one).
- activ_change: reassign the activation function applied after a random layer from a random network of a GAN.
- weight_change: reassign the function used to initialize the weights in a random layer of a random network of a GAN.
- add_layer: add a layer (with random size, activation and initialization function) in a random location of a random network of a GAN.
- latent_change: change the distribution from which the noise (represented by z in the illustration shown in Figure 3.1) for G is sampled.
- loss_change: randomly reassign the divergence measure used as the loss function for training the GAN.
- D-G_loops: randomly reassign the amount of updates a random network of a GAN receives in one training iteration.

Because the approach has to deal with two objectives, the Pareto-dominance based selection scheme of NSGA-II [27] was used to choose the individuals advancing to subsequent generations.

Algorithm 3.1 displays the NE method employed in pseudocode form.

1 Set $t \Leftarrow 0$. Create a population D_0 by generating N random GAN configurations.
2 **do**
3 Evaluate D_t using the fitness function.
4 From D_t, select a population D_t^S of $K \leq N$ solutions according to a selection method.
5 Create the offspring set O_t by applying genetic crossover to D_t^S with probability p_x or apply mutation to O_t with probability $p_m = 1 - p_x$.
6 Create D_{t+1} by selecting the best N solutions in $\{D_t, O_t\}$.
7 $t \Leftarrow t + 1$
8 **until** Termination criteria are met.

Alg. 3.1: GAN evolving GA

The network structures were constructed using the `tensorflow` library [1]. The DEAP library [37] was used to develop the NE strategy that operates over the developed GAN code.

3.4 Pareto Set approximation

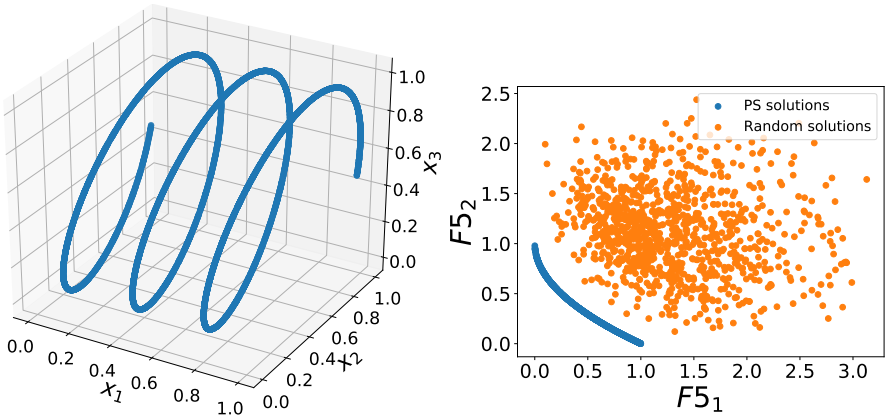
In addition to the issues a designer has to face when composing a GAN, we have to add the often difficult task of evaluating the data generated by these models. As an example, in the image generation domain (an area in which GANs thrive), deciding how *realistic* and *diverse* the generated images are is not straightforward. Thus, we propose a benchmark problem for which we can automatically and unbiasedly evaluate whether generated samples are realistic and diverse: multi-objective problems.

3.4.1 Multi-objective approach

The benchmark of multi-objective problems introduced in [78] consists of a set of continuous optimization test problems with prescribed PSs (nine total problems, F1-F9). Each problem is composed of a set of objective functions which compose the optimization problem, and another function. This last function is able to generate variable values known to be in the PS defined by their corresponding optimization problem.

In Figure 3.3, an example of the main attributes of a given problem (in this case $F5$) is shown. Figure 3.3a displays values of the three first decision variables (x_1, x_2, x_3) of points known to be in the PS, whereas Figure 3.3b shows the projection of these points in the objective space (in blue), along with other randomly generated points (in orange).

From the 9 multi-objective functions introduced in [78], 8 were used for our experiments. We excluded $F6$ to avoid introducing noise into our analysis, as it has three objectives, while the rest of the functions have only two.



(a) (b)

Fig. 3.3: First three decision variables of points known to be in the PS of $F5$ (left). Comparison of these variables in the objective space, i.e., the PF, to random generated decision variables (right).

GANs are trained to sample PS approximations using data points uniformly sampled from the PS functions. After training has been completed, the PS samples produced by the GAN generator are evaluated using the bi-objective function. These evaluations, which are the projections of the decision vectors to the objective space, are considered a PF approximation. The proximity of the real PF and the approximations provided by the GANs is used as a metric for measuring the quality of a given GAN.

3.4.2 Quality measure

The inverted generational distance (IGD) [23], which measures the distance between each point r in the reference PF (R) and the closest point a in the PF approximation (A), is the chosen metric. IGD measures how close A is from R, while it also measures how dispersedly distributed A is. Its formal definition follows:

$$IGD = \frac{1}{|R|}\left(\sum_{r\in R} min_{a\in A} d(r,a)^p\right)^{\frac{1}{p}} \tag{3.2}$$

As can be observed, for a PF approximation A to obtain a low IGD value when contrasted with a reference PF R, each point in R should have a *similar* point in A. Thus, using the IGD as a metric for the EA makes the algorithm penalize GANs which fall into the two categories that we wish to avoid, that is, those models whose approximations are not similar enough to real data, and those which produce collapsed approximations.

The ability to objectively and automatically measure the dispersity of the samples generated by a GAN is one of the keys of this approach. To implement a similar approach in, say, the picture generation field, one would need to rely on indirect metrics which do not necessarily reflect reality, such as the Inception Distance [113] or Fréchet Inception Distance [54].

The IGD is a variation of another metric, the generational distance (GD). GD computes the summation of the distances between every point in the approximation and the closest point in the real PF. It is formally defined as:

$$GD = \frac{1}{|A|}\left(\sum_{a\in A} min_{r\in R} d(a,r)^p\right)^{\frac{1}{p}} \tag{3.3}$$

For both cases, we use $p = 2$ and

$$d(x,y) = \left(\sum_{k=1}^{m}(x_k - y_k)^2\right)^{\frac{1}{2}} \tag{3.4}$$

While the IGD can be exploited to determine how well distributed the PF approximation is, it does not take into account the generated points that lie far from the PF, as it only considers the points closest to the real PF. GD covers this deficiency, as it takes into account each and every generated point, although it does not consider how distributed the generation is. This makes the two distances complementary.

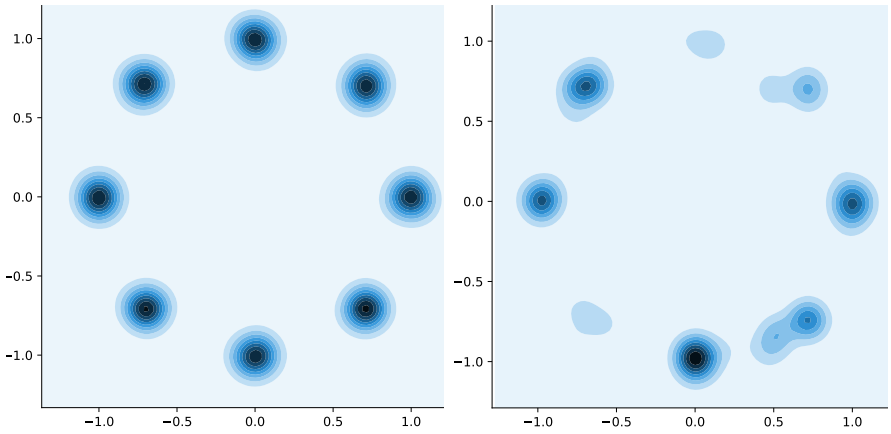
Other metrics, such as the hypervolume [144], could also be used in the approach described in this chapter, both in place of the IGD and GD, or as a complementary metric.

GANs are trained to generate points in the decision space, while the models are evaluated by the projection of their samples in the objective space. On the one hand, this simplifies the computation of the fitness function for a given GAN, as the number of decision variables is commonly much larger than the number of objectives in a multi-objective problem (MOP). On the other hand, we have to assume that a good covering of the reference PF by the projection on the objective space of the points generated by a GAN implies high quality and specially diversity in the generated points. This is not, however, guaranteed, as several points close to each other in the decision space could potentially produce distributed PFs. This issue is analyzed in the experiments section.

3.5 Gaussian Mixture approximation

The Gaussian mixture approximation problem [93] consists of generating a set of points x from a mixture of Gaussian distributions $x \sim \mathcal{N}(\boldsymbol{\mu}, \sigma)$, where $\boldsymbol{\mu} = \{\mu_0, \mu_1, ...\mu_{m-1}\}$ is a vector of means and σ is the unique variance parameter. This problem is especially designed to detect mode collapsing GANs, as models suffering from this deficiency easily learn to generate points from a subset of the m modes. In this case, the chosen specification is the 2D 8-mode variant; in

which $m = 8$, and $\mu_i = (\mu_i^0, \mu_i^1)$. The two components of each μ_n can be interpreted as coordinates in a 2D grid. A visualization of the distribution of x is found in Figure 3.4a. It shows the 8 distribution centers (with $\sigma = 0.05$) describing a circle: $\mu_0 = (0, -1)$, $\mu_1 = (-\sqrt{2}, -\sqrt{2})$, $\mu_2 = (-1, 0)$, $\mu_3 = (-\sqrt{2}, \sqrt{2})$, $\mu_4 = (\sqrt{2}, -\sqrt{2})$, $\mu_5 = (1, 0)$, $\mu_6 = (\sqrt{2}, \sqrt{2})$, $\mu_7 = (0, 1)$.



(a) Target distribution

(b) Undesired effect.

Fig. 3.4: 8-mode, 2D Gaussian mixture problem target and approximation examples.

After being trained, a generative model is sampled aiming to reproduce the target distribution. However, not all models are able to do so, producing results as shown in Figure 3.4b. In this example, a very reduced number of points have been generated where two distribution modes were placed, and some other points far from the original means have also been sampled.

In order to numerically test the quality of a set of samples (and thus, the quality of the generative model), the Maximum Mean Discrepancy[1] [50] (MMD) has been chosen. This metric gives a numeric value representing the difference between two distributions.

3.6 Related work

Even though the application of NE algorithms in multiple areas has already been extensively researched [9, 81, 91, 95, 96, 121, 122], GANs have not particularly benefited from these methods until very recently. Additionally, these techniques have

[1] Available in `https://github.com/tensorflow/models/blob/master/research/domain_adaptation/domain_separation/losses.py`

been near-exclusive to CNNs. We cover this recent literature (along with some other work related to ours) of approaches which could benefit from the study performed in this chapter. Some work carried out on knowledge transferability are also included.

3.6.1 GAN development

In [130], an evolutionary approach, E-GAN, is proposed, in which the antagonistic nature of the GANs is exploited in a different and interesting way. It does not consider the discriminator network as part of the evolutionary process (e.g., an evolvable component within each individual), but it contemplates a single discriminator during the whole evolutionary procedure (against which all generators are tested) as the environment the generators need to adapt to.

The mentioned work addresses two relevant questions. The first one is the consideration of different loss functions for training the networks as a part of the evolutionary algorithm. The second one is related to the mode collapsing issue; the authors add a regularization term, proportional to the confidence with which the discriminator rejects fake samples. This term is reflected in the fitness value of the generators, which are penalized when the samples they generate are limited to a small region of the data.

Even though both the work presented in [130] and ours pursue similar objectives (obtaining generators that avoid the mode collapsing issue), the two approaches differ from the very core of the methodology. Developing a single discriminator over the whole evolutionary process allows all efforts (genetic operators and selection) to be focused on generators, which produces a more in-depth search in this aspect. However, this approach lacks exploitation of the mutual dependency of the generator and the discriminator, which is key in the GAN framework. Additionally, the structures of the networks are fixed beforehand. Genetic mutation consists of weight training iterations applied according to one of three different loss functions. In contrast, our approach evolves GAN networks by pairs, boosting different learning dynamics. Furthermore, our proposal integrates a second objective during evolution, which benefits shorter elapsed times of training and inference, indirectly penalizing structural complexity.

In [2], the authors propose Lipizzaner, an approach based on the diversity enhancing method of spatial evolution [98]. In this case, two separate populations of generators and discriminators are evolved on a grid structure. For evaluation, each element is tested *against* its counterpart in the other grid, as well as its neighbors. This results in an efficient manner of implementing a co-evolutionary approach. This particular method relies exclusively on the Wasserstein loss function, and describes two different mutation operators; a gradient based one (applying an iteration of learning using a minibatch), and a gradient-free option (modifying the learning rate). The fact that the genetic operators cannot modify the structure of the networks prevents the algorithm from exploiting this key feature of the GANs.

The two mentioned works, E-GAN and Lipizzaner, are combined in [128] to form Mustangs, adding the three different mutation operators used in E-GAN to Lipizzaner. In this case, MLP networks are also considered (for evolving GANs that deal

with the MNIST [77] problem). This enhances the latter by adding the loss function component to the evolutionary process. However, the structures remain fixed during the whole procedure.

A co-evolution framework in which generators and discriminators are evolved in isolated populations is proposed in [24]. They are, however, combined when evaluating the different individuals of both populations; the generators in the current population are tested against the two best performing discriminators of the previous generation, and vice-versa. This work also fails to take full advantage of the dependencies between G and D during evolution.

The authors of [92] develop a variation of the genetic algorithm presented in [42], where a very unbalanced problem is tackled. The best found GANs are maintained to take part in the crossover step (thus keeping elite components which are always present in all populations), and GAN generations belonging to the less represented class of training data are added for future generations.

A completely different approach to the one introduced above is presented in [18], where the samples themselves are evolved in order to maximize the learning of the GAN model in each step.

Some works that rely on *populations* of discriminators [31, 102], generators [56] or both [60, 127, 133] have also been developed, although these do not follow an evolutionary approach to the design of GANs.

Another interesting approach implemented in [47]. The proposed concept is, in some manner, similar to E-GAN, as the algorithm focuses on the generator. However, in this case, discriminator structures are also developed, although these developments to the discriminator are fixed beforehand and are not part of the automatic design.

Finally, the authors of [64] propose initializing the search for GAN design with shallow GANs, which are trained with very-low resolution version of the original images. As the networks within the GAN are gradually deepened, the resolution of the images is increased, demanding the improved models to solve a more complex task. The authors claim that the increase in performance compared to regular GAN training comes as a result of the *healthier* competition between G and D.

All the evolutionary proposals mentioned in this section start from randomly initialized individuals which, although it allows a wide exploration of the search space, results in several very costly evaluations until an acceptable convergence point is reached. By adjusting the initial individuals to patterns known to be beneficial in the design of GANs, the authors of [64] could be largely more efficient. Furthermore, it is theorized that constraining the NE exploration to reduced areas can produce results as good (if not better) than not doing so [136]. Additionally, all GANs are evaluated, which not only includes the training of the structures, but also computation of the fitness function. This can also significantly increase in the total elapsed time.

3.6.2 Transferability

Recent work [25, 67, 99, 146] on DNNs has increasingly emphasized the importance of model transferability across problem domains as an efficient way to reduce the time-consuming problem of network design.

The authors of [146] use NAS for developing CNN cells (small groups of operations common in CNNs) employing the CIFAR-10 [68] database. These cells are then applied to more complex structures in order to learn a deeper CNN intended for a considerably larger database, ImageNet [28]. This proposal provided in state-of-the-art results for both problems.

In [25], the inverse version of the transferability problem is studied, as the authors employ a DNN model with high performance in a large-scale domain, in a simpler problem with weight fine-tuning as adjustment. The authors, as expected, found high correlation between domain similarity and transferability levels. A similar conclusion was reached in [67]. In this study, the hypothesis that assumes high generalization of DNN models which offer good performance in ImageNet is tested. Despite the fact that fine-tuned models were able to generalize, the authors complemented their findings stating that the features learned by the models, specifically for ImageNet, did not show a high level of transferability.

The research presented by [99] shows that lottery ticket initializations, an approach that searches and identifies small sparsified networks with appropriate initializations, can generalize across a variety of datasets in the image domain. This is a promising approach which could eventually be applied to the GAN search performed in this chapter.

3.7 Experiments

A set of experiments has been designed in order to answer the following questions:

1. Determine the ability of the proposed NE algorithm to evolve GANs whose approximation accuracy improves as generations advance.
2. Study the scalability of the algorithm when increasing the number of decision variables.
3. Ascertain whether the evolved GANs are robust enough to consistently outperform non-evolved ones.
4. Determine whether optimizing the IGD between PFs is an adequate choice for obtaining generators that accurately generate diverse samples in the feature space.
5. Investigate the transferability of the structures in different directions: problem dimensions, across bi-objective functions, and across problem domains.
6. Identify and discuss frequent patterns in the GAN configurations that provided the best results in the different GAN evolution procedures.

Gradually improving structures that are able to consistently produce accurate approximations is the baseline challenge that the evolutionary process has to fulfill. We consider the scalability and the transferability of the algorithm to be relevant. First, because MOPs with a large number of variables are hardly ever addressed in the literature. Second, because being able to obtain GAN structures evolved in a domain under certain characteristics, which can offer consistent performances in other domain specifications or areas would result in large computational resource

savings. Finally, by analyzing patterns in evolved GANs, we can identify structure choices that result in strong-performing GANs, which can be used for simplifying structural designs, regardless of this being manual or automatic.

3.7.1 Experimental framework

The functions used in the experimental framework have their own particularities [78]. The formulas for generating PS points and evaluation functions for $F1$ contain exponential functions. $F2$, $F3$, $F4$, $F5$, and $F9$ contain sinusoidal functions in both formulas (additionally, the PS point generation formulas for $F3$, $F4$ and $F5$ are piecewise continuous functions). Finally, the formulas for generating PS points for $F7$ and $F8$ contain exponential functions, whereas the objective function formulas are defined by sinusoidal functions.

Thirty executions of Algorithm 3.1 are run for each function, in a bi-objective optimization variant. One of the objectives of the NE algorithm is to minimize the IGD metric between the samples generated by the GAN and a test set of PF points. The other objective pursues minimizing the time required for both training and sampling, aiming at reducing structural complexity in the GANs and accelerating the convergence of the algorithm. The GA is run for problem sizes $n = 10$ and $n = 784$[2].

The population size is set to $N = 100$ and the number of generations made available to the algorithm to $n_{gen} = 100$. These hyperparameters are the same for $n = 10$ and $n = 784$, as the parameter range of the GANs does not vary between different problem sizes, thereby keeping the search space equal.

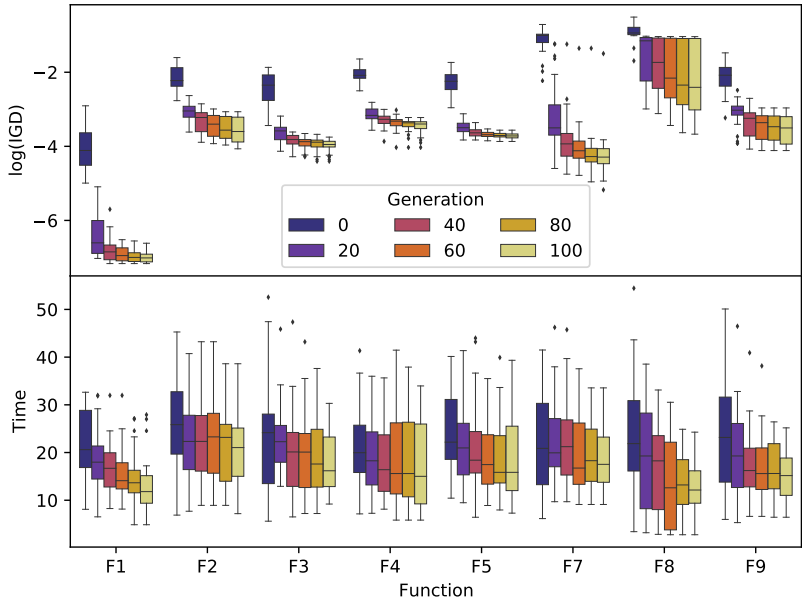
Once the NE process has concluded, we extract the best performing structures in terms of IGD (thus ignoring the secondary objective of elapsed time reduction) and examine the performance of these best GANs using different metrics. Additionally, we analyze the characteristics of fully evolved GANs searching for patterns that could ease the design of future structures.

In all train-sample routines for a given GAN, 1.000 points are sampled from the known PS to train the GAN, and another 1.000 points are sampled from the model after it has been trained. From the latter set of points, the non-dominated points are selected to compute the IGD metric against a new set of 1.000 points sampled from the known PS, using the same procedure employed to obtain the first 1.000 point training set.

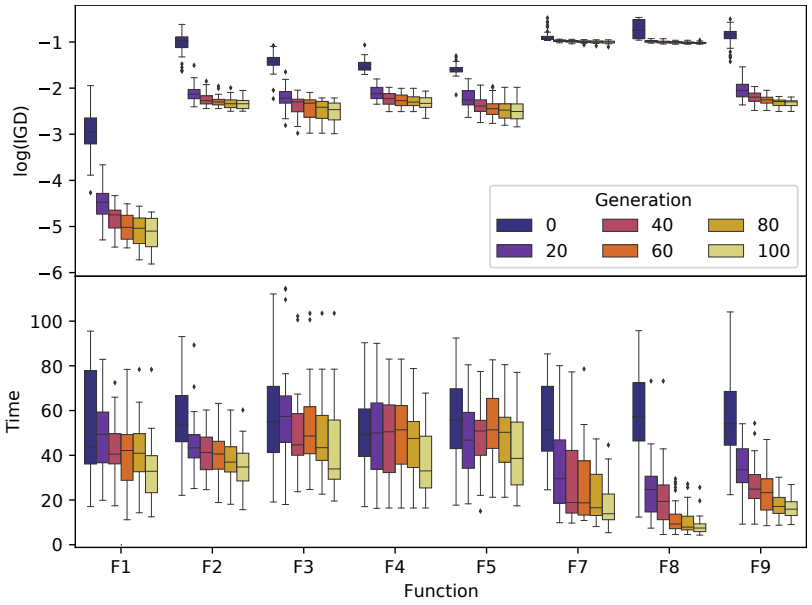
3.7.2 GAN evolution

Figure 3.5 shows the objective values for the best GANs found in terms of IGD value, for each run of the algorithm, in different generations: 0, 20, 40, 60, 80, and 100. The two boxplots on the top show the IGD values (in logarithmic scale) of the mentioned GANs, whereas the two figures on the bottom display the time spent for training and running those same GANs. The two figures on the left present data for GANs trained with $n = 10$, whereas the two figures on the right are relative to $n = 784$.

[2] This choice $n = 784$ was motivated by the MNIST classification problem, as the images have $28 \times 28 = 784$ pixels.



(a) $n = 10$



(b) $n = 784$

Fig. 3.5: Objective values for the best GANs (regarding IGD in logarithmic scale) in different generations: 0, 20, 40, 60, 80, 100, for $n = 10$ and $n = 784$.

As can be seen in the top figures, those showing the IGD values, the proposed approach is able to effectively evolve network structures that gradually improve the approximation of the known PF. This reduction achieves near-zero values for function $F1$. In some cases, convergence was reached considerably early in the evolution. The functions in which GANs obtained the lowest IGD values ($F1$, $F3$, $F4$, and $F5$, those that contain exponential, or sinusoidal and piecewise functions) only experienced minor improvements once the 20th generation was surpassed. On the other hand, functions that seem to be harder to approximate (those with sinusoidal and exponential components in the formulas; $F2$, $F7$, $F8$, and $F9$) appear to continue improving until, at least, the 80th generation, and show a larger variance. Regarding the IGD values obtained, we can clearly perceive different approximation difficulty tiers for the approach presented, corresponding to the function characteristics mentioned earlier. $F1$ is clearly the simplest one, as GANs reach near-zero IGD values. Next, for $F3$, $F4$, and $F5$, the models are able to capture the PS distribution quite accurately. $F2$ and $F9$ PS formulas are slightly more difficult to approximate by the GANs, whereas $F7$ and $F8$ describe significantly more complex distributions.

Considering the differences between the runs for $n = 10$ and $n = 784$, we observe that, as expected, both IGD values and elapsed time increase with the problem dimension.

Specifically, $F7$ appears to lie on the verge between difficulty tiers. With $n = 10$, the IGD values reached by the initial GANs seem close to those achieved with $F8$, whereas the evolved GANs are able to produce IGD values in a similar range to $F2$ or $F9$. With $n = 784$, however, $F7$ clearly falls into the hardest tier.

Regarding the time consumption objective, we observe a significant decrease in the elapsed time, which facilitates a rapid evolution. The increased number of parameters ($n = 784$) results in *slower* GANs, which gives the NE algorithm more room for improvement in that aspect. This shows that the NE algorithm learns to adapt to different scenarios and seizes potential gains wherever it finds them. We observe some instability in the execution times required by the GANs producing the best IGDs, especially with $n = 784$. This reflects the increase in the difficulty of the problem as the trade-off between simplicity and quality causes the best IGD generating GANs to become more complex as generations elapse. The considerable reduction in the time consumption by the GANs further validates the bi-objective approach. Especially when facing complex problems, such as with $n = 784$, it achieves time consumption reductions ranging from moderate (e.g., for functions $F1$ or $F2$, were the average time found in the last generation ranged between 38% and 10% of that found in the first generation) to astounding (e.g., $F7$, $F8$, $F9$, where the elapsed times were reduced to a range between 7% and 23% in the last generation compared to the first one). Anyway, in general, the generated structures result in faster models that are able to generate more accurate samples with lower variance in the metric used for fitness evaluation during evolution.

To present a graphical representation of the PS approximation problem, we provide Figure 3.6. We have selected two functions, $F7$ and $F8$ ($n = 10$). From the 30 runs performed for each one, we selected one run from each, randomly. Samples from the GAN which produced the best approximation in terms of IGD score

at the end of the execution of the chosen run are displayed in the figure. The line in blue represents the original PF, while the red points are the samples produced by the GAN. The first number in the lower part of the figure is the IGD value obtained when comparing the samples obtained from the GAN to the original PF, while the second one shows the number of non-dominated points in the set of sampled solutions. One can clearly observe that the approximation for $F7$ is *better*, as most of the points lie closer to the PF, compared to its $F8$ counterpart. More specifically, the number of points in the PF defined from the $F7$ approximation triples the amount of solutions in the PF generated from the $F8$ approximation (145 to 45). Furthermore, the $F7$ approximation describes a full covering of the PF, whereas the $F8$ one does not cover the PF section with low values for f_1. This is where the IGD value increases the most. The fact that the approximation for $F8$ contains many points which are far from the optimal section does not directly impact the metric.

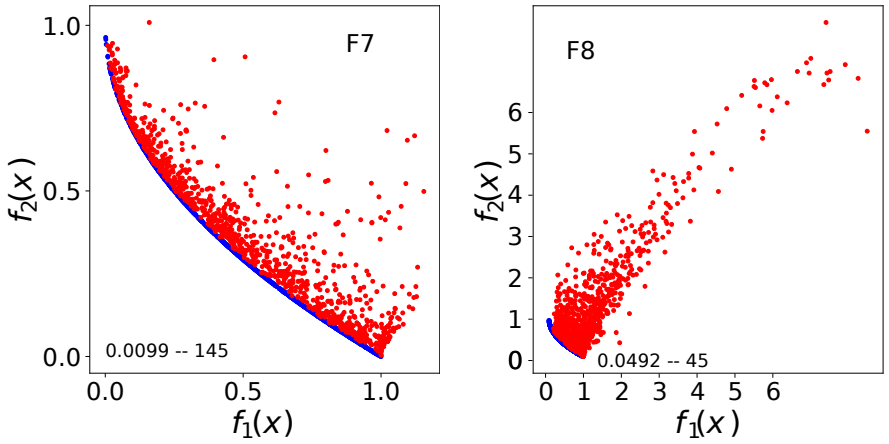


Fig. 3.6: Two PF approximation examples with $n = 10$ in both cases. The line formed by the blue points shows the true PF, and the approximation made by the GAN is represented in red. The x and y axis represent the values achieved in each objective functions by the best approximation found in two random runs for $F7$ (left) and $F8$ (right).

3.7.3 Robustness of the IGD-based approach

In the evolutionary process analyzed in the preceding section we made the choice of evaluating GANs *only* once, so as to keep our procedure agile. We now test whether the apparent success of the algorithm is a result of a random weight initialization that conveniently suits the loss function and the network structure, or if the structural evolution is indeed effective and consistent. With that goal, we select the best GANs

found during each of the 30 runs (in terms of achieved IGD) and use them to generate solutions 15 times. Next, the generated samples are evaluated with the corresponding bi-objective functions, and used to compute the IGD metric. We follow the same strategy with a GAN structure randomly chosen from the first population of each run. The results can be observed in Figure 3.7, which shows, for each function (in the y axis) and each problem dimension, a histogram displaying the frequency of the generated IGD values (x axis, in logarithmic scale). IGD values computed from the solutions generated by the evolved GANs are represented with orange lines, whereas the results of the random GANs are shown in solid blue blocks. At a quick glance, one can see significant differences between the fully evolved GANs and the random ones for all functions, as IGD values resulted from samples by evolved GANs mainly stack near values that random GANs cannot produce at all. For $F8$ and $n = 784$, results are closer than for other instances, showing the difficulty of this particular configuration of the problem. These results are corroborated in all cases by the Wilcoxon signed-rank test [137], which tests the difference of the population mean ranks, by producing near-zero p-values for each of the comparisons. This fact further confirms the proposed algorithm as well as it increases the validity of the analysis performed later, as the evolved GANs are significantly better and more stable than random ones.

3.7.4 IGD metric in the objective space

Intuitively, the usage of the IGD metric as a fitness function for the NE algorithm prevents evolved GANs from indirectly falling for one of their main flaws, mode collapsing. However, this metric might not be the optimal one to evolve GANs using the PS approximation problem. That is, a sparse distribution of the solutions in the objective space does not guarantee the same sparsity in the feature space, which is where GANs operate, and vice-versa. Therefore, using the IGD in the objective space (as used until now, and referred to as IGD_o from now on) could be masking a possible issue of GANs generating solutions from a reduced area in the feature space. Even though we regard this scenario as unlikely, we consider two other metrics that could help determining whether the IGD_o is an appropriate choice for this problem: The GD metric in the objective space (GD_o), and the IGD metric in the feature space (IGD_x) [114].

We select the best GAN found in each run for each function in terms of IGD_o, and a random GAN structure from the initial population, and train and sample each one 15 times. These samples are measured using the three metrics proposed. From all 8 functions employed in this experiment, we choose two representative ones (considering their characteristics and the results discussed earlier) to be displayed in this section, in order to avoid excessively extensive content: F7 and F8 with $n = 10$.

The two plots in Figure 3.8 show one point for each run of each GAN. The best GANs found in the different evolutionary processes are represented with blue dots, while orange ones stand for runs of random GANs. All three metrics denote better performances the closer they are to zero. As can be seen in both figures, by optimizing IGD_o, the algorithm also optimizes the other two metrics to a large degree.

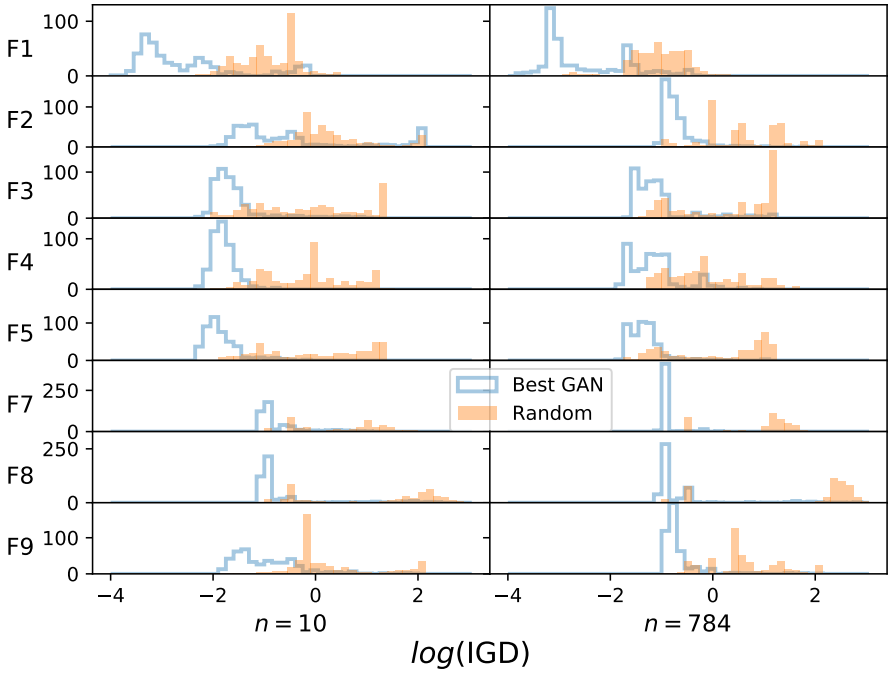


Fig. 3.7: Histograms showing the 30 GANs $\times$ 15 runs $=$ 450 IGD values (in logarithmic scale, the lower, the better) computed for each function (y axis), for the two problem dimensions (x axis). Solid orange lines describe results obtained by a random GAN, whereas the blue contour display the results obtained by the evolved GANs.

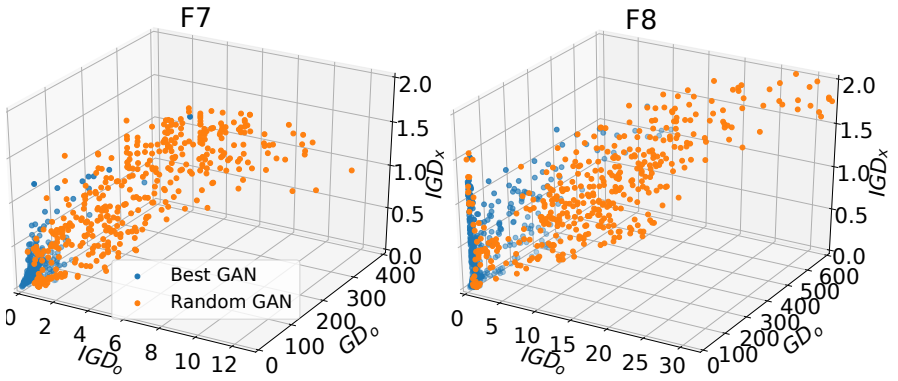


Fig. 3.8: Scatter plots showing the IGD_o, IGD_x and GD_o obtained by the best GANs for $F7$ and $F8$.

Because the increased difficulty at approximating PS solutions, the points displayed in the figure related to $F8$ are considerably more informative in terms of possible weaknesses of the approach. The approximation difficulty makes it harder for the samples of GANs to reach low IGD_o values, and therefore the benefits of evolving structures are much more apparent. In general, a random GAN hardly ever produces solutions in the area where most of the results of the evolved ones collapse (points near 0 values for all metrics, in the lower left corners). Additionally, random GANs produce many poor quality results (sparse points across the whole subfigures), whereas evolved models are able to refrain themselves from doing so. Results for the rest of the functions can be found in Annex A. With this information, we can label the usage of the IGD_o as an effective fitness function for GAN evolution in our particular approach.

3.7.5 Transferability of the evolved GANs in the PF approximation problem

One of the main questions regarding the proposed algorithm is its capacity of generalization, i.e., how much of the performance of an evolved structure (and, in general, of the knowledge generated in the process) can be transferred to another problem version, or a different problem altogether. Ideally, a transferable GAN structure should successfully learn to generate diverse distributions by learning different combinations of its weights. Here, we test to what extent this hypothesis holds.

This transferability study of the structures evolved by the NE algorithm consists of: i) GAN structures evolved using data points from F_i are trained for sampling points from F_j, and ii) GAN structures evolved using problems of dimension $n = 10$ are used to learn the distribution of other problems of size $n = 784$.

For each function, the 30 structures that obtained the lowest IGD_o values in each execution of the GA were selected and used for learning PS approximations for all the functions and the different n. We measure the quality of the approximations using the IGD$_o$ metric. To take into consideration the variability introduced by the random weight initialization, 5 different repetitions of the learning step were performed.

The two plots included in Figure 3.9 show the transferability of GANs evolved using a certain function (x axis) to a target function (y axis). These figures show a circumference for each combination. The size of the circle and the number written inside them represent the mean IGD$_o$ value for all the $30\ GANs \times 5\ repetitions = 150$ IGD$_o$ values generated, while the color represents the logarithm of the variance. Additionally, those functions for which a GAN trained on a different function outperforms the GAN specifically trained for that function are highlighted with a black circle. For example, the GANs evolved with $F1$ (exponential functions) obtained better results than the GANs evolved with $F3$ (sinusoidal functions), when tested on $F3$. We determine a set of IGD$_o$ values to be better than another one if the Wilcoxon signed-rank test produces a p-value lower than 0.05.

Regarding the transferability between two functions for $n = 10$ (top figure), we observe that highly transferable structures have been evolved. For example, structures evolved for $F1$, $F2$, $F3$, and $F9$ have significantly outperformed structures evolved for other functions in those same functions at least three times. At the same

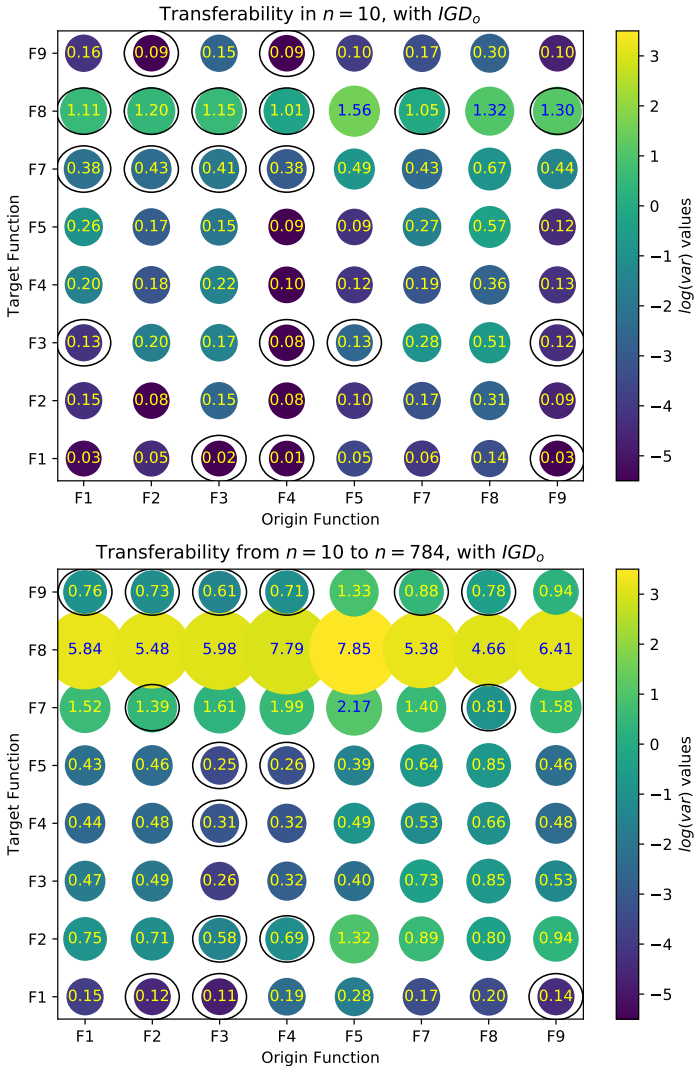


Fig. 3.9: Transferability of the best GANs over different problem dimensions and functions. Origin function refers to the function ($n = 10$ for both figures) with which GANs were evolved. Target function displays the function ($n = 10$ in the top figure, and $n = 784$ in the bottom one) on which the GAN is used to learn an approximation of the PS. The printed number and the circle sizes represent the IGD_o mean of all the runs, while the color represents the variance. If an ellipse is displayed around a circle, the difference between IGD_o values achieved by the GAN evolved with the origin F, in the target function were significantly lower than the ones achieved by the GAN evolved with the target function.

time, these same structures have been outperformed three times by other structures in approximating their own functions. This *instability* comes as a product of the great performance of GANs in functions they have not seen during evolution. Additionally, $F4$ stands out as a very solid *evolving function*, as its structures have outperformed five other sets of structures at approximating their own functions while matching results in the other three, and not being outperformed even once.

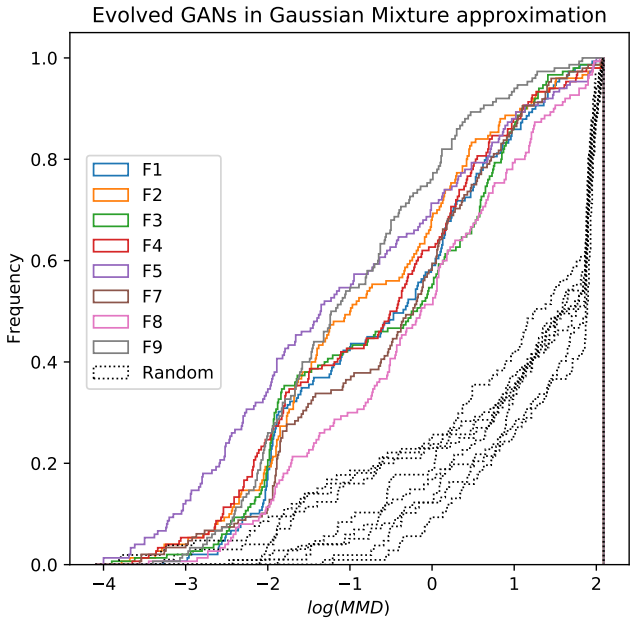
With respect to the comparisons between structures evolved using different problem sizes, we observe a varied scenario. Different GANs obtained mean IGD_o values lower than 0.32 for $F1$, $F3$, $F4$, and $F5$, which shows a remarkable generalization capacity. For $F2$ and $F9$, and especially $F7$ and $F8$, however, this was not the case. Results regarding transferability of GANs evolved with $n = 784$ can be found in the Annex A. As could have been expected, transferability between problems which present a similar challenge to the GANs is more fluid than between problems of varying difficulty. However, the amount of times that GANs evolved with one function Fi performed better than other GANs evolved for Fj ($i \neq j$) when tested with Fj, allows us to safely state that transferability is fairly high in this case.

3.7.6 Transferability across problems

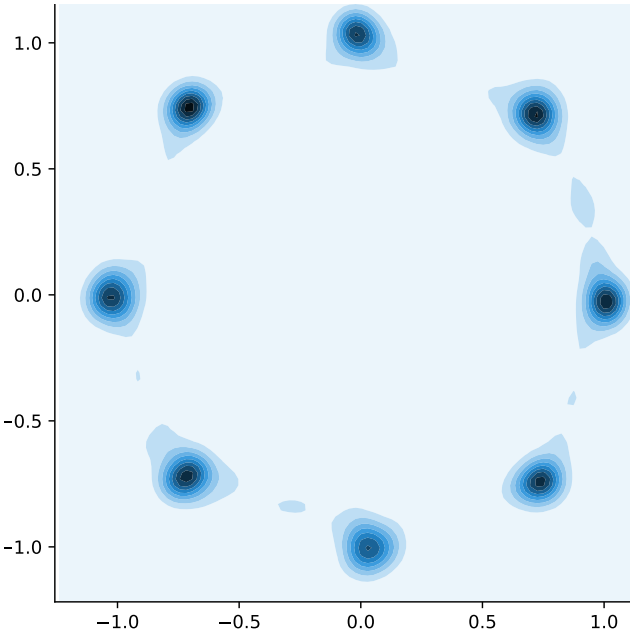
After having observed the transferability between problems of the same suite, where the formulas generating the points have some components in common, we test the evolved structures on another problem which has no relation at all, the 8-mode 2D Gaussian mixture approximation problem. To this end, we again select the structures which generated the lowest IGD_o values in all 30 runs for all 8 problems. Next, similarly to the previous transferability test, all GANs are trained (with 1,000 points and during 10,000 epochs instead of 1,000) and sampled to generate a new set of 1,000 points. Finally, the MMD between the newly generated set and a test set of the target distribution is computed. This process is run five times for each evolved structure. This same procedure is also applied to random structures of the initial populations, in order to put the results of the effect of evolution into perspective.

Figure 3.10a shows the cumulative histogram of the MMD values obtained by the best structures for each function and their random counterpart, in logarithmic scale. Each line represents $30\ runs \times 5\ repetitions = 150$ learning procedures. The profound differences between the solid lines (those representing the evolved networks) and the dashed ones (the random structures) demonstrate the transferability of the evolved GANs, as they obtained significantly lower MMD values compared to the random structures. Figure 3.10b shows the best approximation observed from all GANs (a network evolved with the $F5$ function), which obtained a MMD value of 0.025. Excluding a reduced number of samples generated between modes (small densities shown in Figure 3.10b between the eight Gaussian distributions), the representation is fairly accurate and, most importantly, balanced.

Additionally, to test the consistency of both the random and the evolved structures, we have computed the means and the variances of the 5-repetition sets of all combinations. In this case, all functions are grouped together. Results are displayed in Figure 3.11.



(a) Cumulative histogram of the MMD values.



(b) Best approximation achieved.

Fig. 3.10: Results of the transferability between problems experiments.

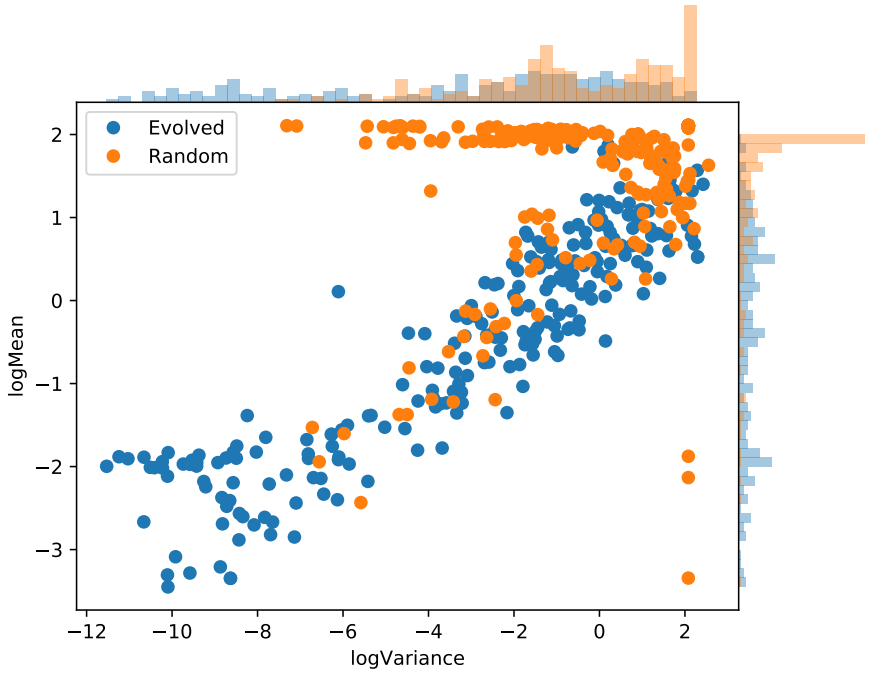


Fig. 3.11: Plot showing the distribution of the variances and the means of the 5-repetition sets. Marginal distributions are also shown.

This figure shows, in this case for all functions combined and for GANs evolved with variable sizes of $n = 10$, the relation between the mean and the variances generated by the 5-repetition sets, in logarithmic scale (some random points which produced very large variance values were reduced to 2 in order to improve the expressiveness of the figure). As can be clearly seen, the top performing structures are the evolved ones, both in quality and consistency, while several random GANs produced significantly worse results. However, it is true that certain overlap between the evolved and random GANs can be observed. This comes as no surprise, as the evolved networks have never been exposed to the problem being treated in this scenario. As a summary, the results from this experiment not only further endorses the previous claim of high transferability, but it takes it to the next level, as now the transferability between problems has also been tested.

Results regarding these two aspects (the frequency of the MMD values and the relation between the mean and the variance) of the structures evolved with variables of size $n = 784$ are available in Annex A.

3.7.7 Good practices when designing GANs

By making the assumption that the GANs present in the last generation of the evolutionary process are composed of some of the best components available in the search space, and knowing about the high transferability of these, we can extract patterns that can be used as guidelines when manually designing GANs, or guiding a future intelligent search. With that goal in mind, we have computed the frequency in which each component is present in the top-performing GANs. This information is displayed in Figure 3.12, which is composed of four different heatmaps. It contains information related to the two aforementioned representative functions ($F7$ and $F8$), for five specific components: Figure 3.12a shows the prior probability distribution and the loss functions employed to train the GANs. Figure 3.12b shows the number of networks with a certain number of hidden layers (the number of neurons in each layer and the loop parameter, the other parameters with the strongest effect in the runtime of the model, follow a similar pattern and are therefore omitted), whereas Figure 3.12c contains the frequency of the functions used to initialize the neuron weights. Finally, Figure 3.12d is related to activation functions. More detailed information about the frequency of the different components in the evolved GANs can be found in the Annex A.

Observing the information on the left-hand side of Figure 3.12a, it is indisputable that the best GANs make use of the random uniform distribution for the latent representation of the data. Even though that distribution can, at least intuitively, be less expressive, one possible reason for this highly balanced result is that the random uniform distribution helps to provide diversity to the GAN generations. The right-hand side of the figure displays the frequency of appearance of the different loss functions used in the GANs. In this regard, we can establish the KL loss functions as the top performing one overall, as they are remarkably present in any function and problem size. Other functions, such as Squared Hellinger and Pearson's χ^2 appear to be viable alternatives as well.

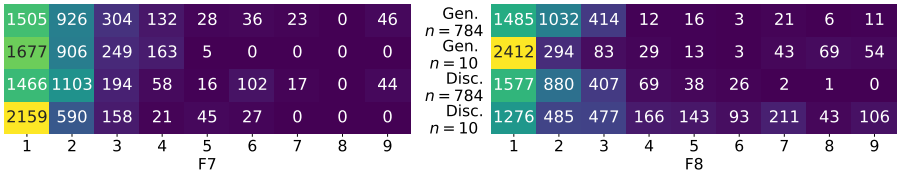
Judging by Figure 3.12b, we observe that most networks are relatively shallow. In most cases, less than three hidden layers have been enough to approximate the data. The number of neurons in the hidden layer (to a lesser extent) and the loop parameter describe a similar pattern towards reduced values. Interestingly, the results with $n = 784$ are considerably similar for $F7$ and $F8$, whereas $F8$ required deeper networks with $n = 10$ as opposed to $F7$. We hypothesize that the time-reduction objective had a strong impact on the evolutionary process and favored networks with fewer parameters, which are faster to train.

Regarding the figure representing the initialization functions, Figure 3.12c shows clear differences between the difficulty tiers we defined in the IGD_o evolution analysis. For $F7$, initializing the networks using the uniform random distribution can, in most cases, be the correct alternative. For $F8$, Xavier initialization is consistently present for any network, especially for the discriminator with $n = 10$, and can be seen as the conservative choice.

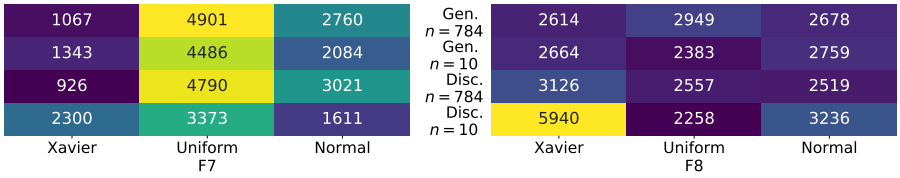
Focusing on the activation functions (Figure 3.12d), we first perceive the notable number of identity functions present. This is probably due to the time reduction ob-

(a) Latent and loss functions.

(b) Number of hidden layers.



(c) Initialization functions.



(d) Activation functions.

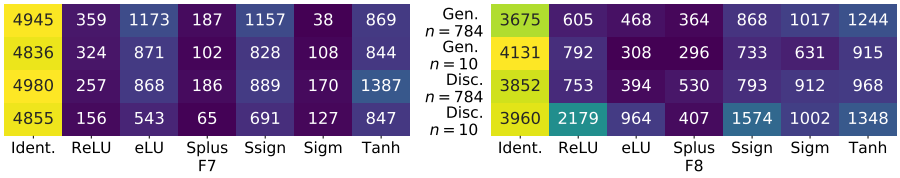


Fig. 3.12: Different characteristics present in the GANs evolved in the last populations of all 30 runs. The x axis represents the different choices for each component, and the numbers and colors represent the appearance frequency on the GANs (the lighter, the more frequent). Colors maintain consistency within subfigures, with the exception of Figure 3.12a, where the color palette is independent for the left and right-hand sides of the subplot.

jective, as a network without activation functions is faster to sample, and especially to train. Therefore, we hypothesize that the NE algorithm converges to networks with one single layer with an activation function, accompanied by more linear transformations. This leads up to the interesting theory that a single activation function is enough to capture non-linear features of the data. Softsign, and Tanh functions are included in both networks for both functions, while ReLU and eLU are also frequent choices for $F7$ and $F8$ respectively.

Overall, we can conclude that, a good starting point for designing a GAN would include the following considerations:

- Uniform random distribution as the prior distribution.
- KL loss functions are generally good choices to train the models.
- Shallow networks provide enough modeling power in most cases, however, depth can be a solution when the given problem is too difficult.
- Weight initialization is highly dependent on the problem, but random uniform or Xavier functions offer better performance.
- For any network, one single non-linearly activated layer is able to capture the non-linearities of the data. The rest of the layers can have an identity activation function.
- Softsign and tanh functions are generally a good choice.

3.8 Conclusions

In this chapter we have performed an extensive analysis of an evolutionary approach for optimizing GAN structures. The problem of PS approximation has been proposed as a convenient benchmark for evaluating the quality, in terms of accuracy and diversity, of the generative capacity of GANs. Furthermore, the evolutionary algorithm also pursued a secondary objective, reducing the computational time. This results in an effective and rapid procedure that is able to produce flexible and fast structures. Additionally, this feature have produced interesting insights about the (lack of) necessity of depth and non-linear activation functions in each layer in GAN networks, at least for the considered problems.

We have shown that the fast method for evaluating the GAN candidates in the different generations is enough to determine whether a candidate is promising or not, as the evolved structures were able to consistently outperform random structures.

We have also confirmed that using IGD, a metric that *only* takes into account the quality of the approximation in the objective space, is capable of developing other interesting characteristics: such as the lack of the generation of *poor* solutions and the diversity in the feature space.

Moreover, with the aim of testing the robustness and generalization ability of the evolved GANs, we have made rigorous insights into the study of transferability, showing promising levels of generalization at three different degrees: Different configurations for the same problem, different problems from the same suite, and most notably, between entirely different problems. This suggests that the usage of the PF

approximation problem is a good starting point for evolving GANs despite the models having an entirely different ultimate purpose. In addition, we have observed a set of regularities among the fully evolved GANs, which implies that some of the GAN components work well with each other. This knowledge can be exploited in different manners to improve future GAN structures. For example, one could simply initialize a GAN with the components appearing more frequently within the mature GANs of the NE procedure proposed in this chapter. Crucially, the fact that the evolved GANs have been able to show strong performances in different problems makes the evolutionary information of these NE runs applicable to potentially any other domain.

Although useful for understanding what makes a GAN structure effective, the methodology presented in this chapter exploits a key domain specific characteristic of the PS approximation benchmark: The IGD metric. This kind of metric is not available for the wide series of tasks GANs are used for, which causes the transition from this domain to others to be non-trivial. Any field that includes an automatically computable evaluation metric, is susceptible of benefiting from the GAN evolution method proposed in this chapter.

The generalization capacity, combined with the algorithm agility, displays the potential of application of this framework in different areas. For example, designing models used to guide population-based optimization algorithms. This is covered in the following chapter.

Once the application of NAS techniques to the GAN domain has been studied, the next step consists of extending the methods to make NAS more efficient to a more general search space. One in which models with different goals (and even multiple goals at the same time) can be developed. In this sense, the efforts of the Part II of this dissertation are devoted to define and explore a search space with these characteristics.

4

Exploitation of Neuroevolutionary Information for NAS efficiency gains

4.1 Introduction

In the previous chapter, we introduced a successful NE approach to evolve GANs for generating Pareto set (PS) approximations. After that, the evolved GANs then proved to be able to generalize to other problems of similar and even different domains, which suggests that the evolved structures are highly transferable. Additionally, it has also been observed that some GAN defining components (e.g., the weight initialization function or number of neurons in a layer) are more frequent than others in fully evolved GANs, which could mean that these components are *more adequate* for GANs than others. However, we regard dependencies between these components being the real reason behind strong performances of GANs as the more likely scenario. Based on this hypothesis, this chapter presents a step forward in the analysis of these patterns.

More specifically, we first record the characteristics of the best GANs found during the evolutionary searches described and analyzed in the previous chapter. By modeling the best found configurations, it is possible to determine the attributes desired in top performing GANs, which would be useful at the time of creating new DNNs (e.g., when initializing individuals for NE algorithms or when aiming for structures with a component combination which could lead to efficient learning without any structural optimization). Additionally, if the metamodel were able to recognize GAN specifications which are unlikely to deliver good results after being trained, multiple unproductive training iterations could be avoided in future GAN NAS runs. This would lessen the computationally costly nature of NE algorithms (at least) when used to evolve GANs, which is one of the main drawbacks of these algorithms [136].

We consider probabilistic graphical models (PGM) [108] an appropriate tool to model the patterns described by the dependencies within the components of the top performing GAN structures.

The chapter is organized as follows: In Section 4.2, we introduce the proposed approach. Once the proposal is presented, its validity and generalization capacities

are tested in Section 4.3. Finally, the most relevant conclusions are summarized in Section 4.4.

4.2 BN-assisted NAS

For a correct operation of a DNN, the dependencies between the different components composing the model need to be addressed. This is especially applicable to GANs, since they are commonly composed of two DNNs, which increases the number of choices to be made when designing them. In this chapter, as in the previous one, the DNNs in a GAN are characterized by the number of layers of each DNN and the specification of these (e.g., their activation and weight initialization functions, their number of neurons in MLPs, the number of filters and their sizes and strides for CNNs, etc.), as well as the frequency the two networks are trained with. Other *global* parameters, such as the distribution followed by the noise fed to the discriminator or the loss function used to train the model are also considered components of a GAN. In this work we have chosen PGMs, and particularly Bayesian networks [14] to model the dependencies between the GAN components described in the patterns, because three main characteristics. First, they are naturally modular so that complex dependency structures can be described and handled by a careful combination of simple elements. Secondly, they are visually representative, which can help to understand the decision making process [73]. Finally, there are different ways of introducing *expert knowledge* into the model.

4.2.1 Bayesian networks

Bayesian networks are probabilistic graphical models that are used to represent sets of variables, together with their (in)dependencies, by means of directed acyclic graphs (DAG) [108, 109]. In addition to the graphical structure, a set of parameters represents the values of the marginal and conditional probabilities represented by the graph. Each node of the DAG represents a variable, and the (non) existence of an arc between two nodes represents the (in)dependency between them. When the graph structure of a BN is given by an expert, only the parameters are learned from the data. If that is not the case, the structure can be learned using the available data.

Therefore, the automatic learning of a BN consists of two phases: i) devising the topology of the network, i.e., which connections should and should not exist, and ii) learning the conditional or marginal probabilities of each variable [72].

In this chapter, we use BNs to codify the probabilistic relationship between the components used to construct the GAN, e.g., activation functions, size of the layers, or weight initialization procedures. Once the BN has been learned from the data gathered during the evolution of different DNNs, we will perform inference, looking for the most probable (promising) configurations, or calculating the probability given by the BN to a particular set of DNN characteristics.

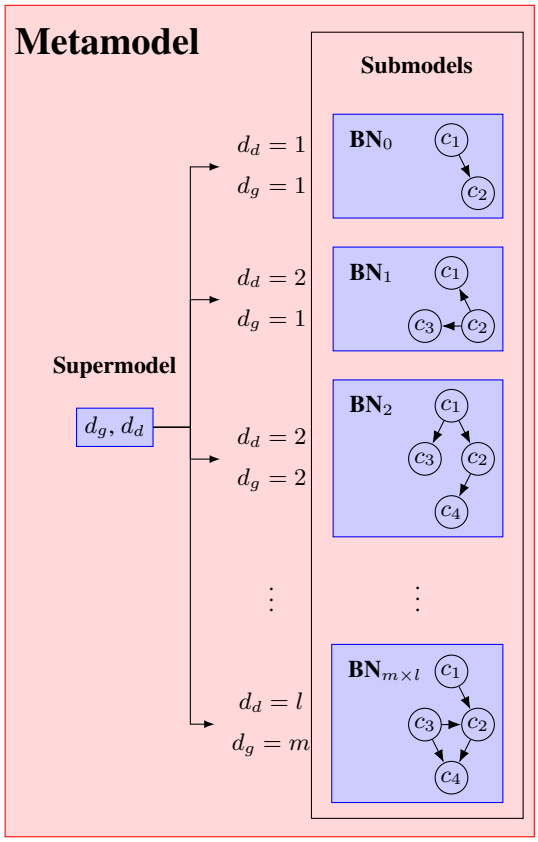


Fig. 4.1: Graphical representation of the metamodel. This visualization assumes that the generator and the discriminator are codified within a single individual. If the DNNs were evolved separately, a single submodel per DNN per depth would be required. The example BN graphs do not represent a realistic scenario in terms of number of nodes. An example of a BN submodel learned for the metamodel can be seen in Figure 4.2. l and m represents the maximum number of hidden layers in the discriminator and generator, respectively. The c_n represent the different components in a GAN.

4.2.2 Metamodel choice

Most current NAS algorithms allow the architectural optimization of DNNs of varying depth, and the more layers in a DNN, the longer the list of components required to characterize it. This leads to DNN characteristic vectors being of variable size. Unfortunately, dealing with a variable number of features is not straightforward for a BN model. Because of this, we have chosen to design a two-level metamodel. The first level, the *supermodel*, represents the number of layers of the generator and the

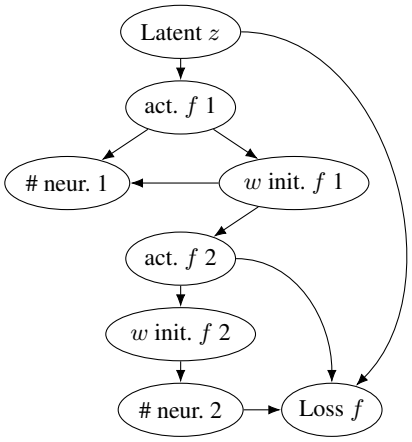


Fig. 4.2: Submodel example. In this case, for the sake of simplicity, only one network of two layers is represented (composed of the number of neurons, the activation function, and the weight initialization function). Two other key components of a GAN, the distribution of the latent variable and the loss function used to train the model are also included.

discriminator. Next, in the second level, one BN is learned for each DNN depth. In cases where the GAN is considered as a sole entity, one *submodel* is learned per each depth combination of the two networks. If, conversely, the structure of the generators and discriminators are optimized independently, a single BN per depth per network is required. Figure 4.1 shows a graphical representation of the metamodel, while Figure 4.2 represents an example of the learned submodels.

In this work, after a preliminary experimentation, we decided to employ the ARACNE [89] algorithm -an extension to the Chow-Liu algorithm [20]- to learn the structure of the BN, as it provides a good trade-off between complexity and efficacy.[1]

4.3 Experiments

The experiments carried out in this chapter are divided into three parts. In the preliminary, isolated from any structural search method, the capacities of the approach to sample and discriminate GAN structures by their quality are tested. After that preliminary experiment, the ability of the metamodel to sample high quality GAN structures that can, among other things, serve as a starting point for other structural searches -one of its main goals- is also tested. In the third and final one, the second main goal of the approach is tested: its capacity to guide a future structural search. Two different problems have been used for the experiments in order to show the applicability of the approach.

[1] The BN related implementation is based on the bnlearn R library [115].

4.3.1 GAN choice

In all cases, the metamodel is nurtured with sets of GAN structures extracted from previous NE runs. We now define three sets which are used throughout this experimental section. The first set, `First`, consists of the n best performing individuals (GAN structures) of each run in the ranking ordered by the chosen quality metric, e.g., the fitness value. The second one, `Second`, is composed of the n second best individuals, that is, from the $n+1-th$ to the $2n-th$. Finally, the `Random` set contains n randomly generated individuals, e.g., from the initial population. The `First` set will be used to learn the metamodel in all cases, the `Second` set will help test its generalization capacity, and the `Random` set acts as a control set. In an ideal scenario, the results produced by the GANs generated by the metamodel would lie closer to those produced by the GANs in `First` and `Second` than to `Random`.

4.3.2 Preliminary experiments

First, we aim at demonstrating the capability of the metamodel to perform the following two tasks:

- Discriminate structures by their potential at describing effective GANs.
- Sample GAN structures poised to be used as efficient GAN definitions.

To that end, we base this set of preliminary experiments on the information saved from the NE runs in the previous chapter. Overall, for this part, there are $8\ functions \times 2\ problem\_sizes \times 30\ runs = 480\ runs$. The functions from the suite defined in [78] being used in this work are $F1$, $F2$, $F3$, $F4$, $F5$, $F7$, $F8$, and $F9$[2]. To select the GANs forming the three sets, only the IGD objective was taken into account. The time reduction objective was ignored.

4.3.2.1 Structure and representation choices

The GAN components represented in the metamodel consist of those described in Section 3.3 of the previous chapter.

Whereas the cardinality of the sets of possible values that most components can take is relatively reduced -the largest number of choices belonging to the loss function component with a total of eight-, the number of neurons in a layer varies between one and fifty. This results in a large metamodel complexity increase, as different probability tables might have to be computed for each of these values. We consider that this would not add much value to the metamodel, and therefore decided to discretize these values in such a way that intervals of 10 values are represented by the first value of that interval (e.g., the "30" symbol represents layers with [30,40) neurons). This way, instead of having 50 possible values, only 5 have to be modeled. Of course, other decision could be taken without loss of generality. When sampling a

[2] Remember that $F6$ was not used in the previous chapter because of it having three objectives rather than two, as the rest of functions.

new GAN, the obtained value will have a random number between 0 and 10 added to compensate the "loss of value precission".

Additionally, in order to avoid mixing continuous and discrete values, we have also chosen to contemplate as discrete the other two values which could be considered continuous: the number of hidden layers, and the training frequency.

In this particular case, we use the best five ($n = 5$) individuals from each run. Because 480 runs are available, the metamodel is learned using a total of $480 \times 5 = 2,400$ GAN specifications.

The DNNs within the GANs available in the NE runs are evolved together and they have a maximum of 11 hidden layers and a minimum of one, resulting in 121 possible combinations. A preliminary analysis of these structures showed that more than 75% of the MLP-GANs in `First` were combinations of generators of a depth up to 3 layers, and discriminators with no more hidden layers than 4. Therefore, in order to reduce the complexity of the model built, we decided to limit the GANs to these $3 \times 4 = 12$ of the total $11 \times 11 = 121$ depth variants and rebuild the three previously described GAN sets.

4.3.2.2 GAN Likelihood

First, the capacity of the metamodel to distinguish between GAN structures with a higher chance of providing good quality results is tested.

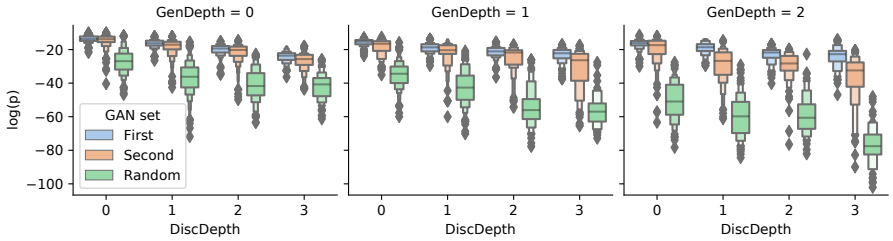


Fig. 4.3: Probabilities (in logarithmic scale) provided by the metamodel to GANs present in the three defined sets.

After training the metamodel using the GAN structures in `First`, we compare the probability assigned by the metamodel to GANs from that same set, and the `Second`, and `Random` sets. If the probability values provided by the metamodel to GANs from the `First` and `Second` set are similar, and the values between `Second` and `Random` are different, we can determine that the model is good at both generalizing to unseen high performing structures and discerning structures likely to result in poor sampling GANs.

Figure 4.3 shows three boxplots (one per generator depth) in which the probabilities (in the y axis in logarithmic scale) assigned by the metamodel to the GANs of

each set are displayed. The x axis represents the depth of the Discriminator. As can be seen, the metamodel assigns similar probability values to GANs in `First` and `Second`, although slightly higher ones to the GANs in `First`. At the same time, it is able to discern GANs from `Random` by assigning them considerably lower probabilities.

The apparent differences between the probability sets are confirmed by the Kruskal-Wallis statistical test [70], which provided a maximum p-value of $\sim 10^{-27}$ for the $3gen. \times 4disc. depths = 12$ different combinations tested. The Dunn statistical test [30] confirmed that all pair-wise comparisons (including those involving probabilities assigned to `First` and `Second`) were different, as the maximum p-value found for the $12comb. \times 3pairs = 36pairs$ tested was ~ 0.04. However, the differences on the p-values provided by the Dunn test varied greatly depending on the pairs being compared. Between the `First` and `Second` sets, the p-values ranged from ~ 0.04 to $\sim 3 \times 10^{-26}$, whereas comparing any of these two sets to `Random` yields p-values between $\sim 10^{-16}$ and $\sim 10^{-161}$. This shows that the *gap* between the probabilities of the `First` and `Second` set GANs is much narrower than that between any of these sets to the `Random` set. The Kruskal-Wallis and Dunn statistical tests were chosen due to their applicability as non-parametric tests, in scenarios in which normality of the data cannot be assumed.

4.3.2.3 Sample generation capacities

Secondly, to assess the sampling capacities of the proposed metamodel, we first define two variations of the `First` and `Random` sets. $\text{First}_{\text{train}}$ is a restricted version of `First` which, instead of including structures from all 480 runs, only those runs involving 5 of the 8 total functions (F1, F2, F3, F4, and F7) were considered, totaling 300 runs. $\text{Random}_{\text{train}}$ is populated similarly, but with n random GANs instead of the best found ones. The selection of the functions in each set has been performed according to the function similarity criterion deduced in [40], also observable in the formulas available in the original work [78]. This way, we simulate a scenario in which the metamodel is learned from data coming from problems similar to those which are going to be dealt with later, while these last ones remain unseen to the model. We regard this scenario as the typical application of the methodology, since, considering the amount of NE procedures carried out in the literature, it is highly likely to find problems which have been dealt with using NE and that are close to the target. Further inclusion of problem characterization into the metamodel to achieve even greater results is left as future work.

A new metamodel is learned using $\text{First}_{\text{train}}$, before being sampled 100 times. The sampling is achieved by using probabilistic logic sampling [53], a method that samples variables following their ancestral order. These samples conform the Sampled set. Another 100 GANs are randomly chosen from $\text{First}_{\text{train}}$ and $\text{Random}_{\text{train}}$. All these structures are trained to reproduce Pareto set approximations of the functions that have remained isolated from this procedure (F5, F8, and F9), for both variable sizes. The obtained IGD values between the solutions sampled from all these GAN structures and the real PF in the different problems are displayed in Figure 4.4.

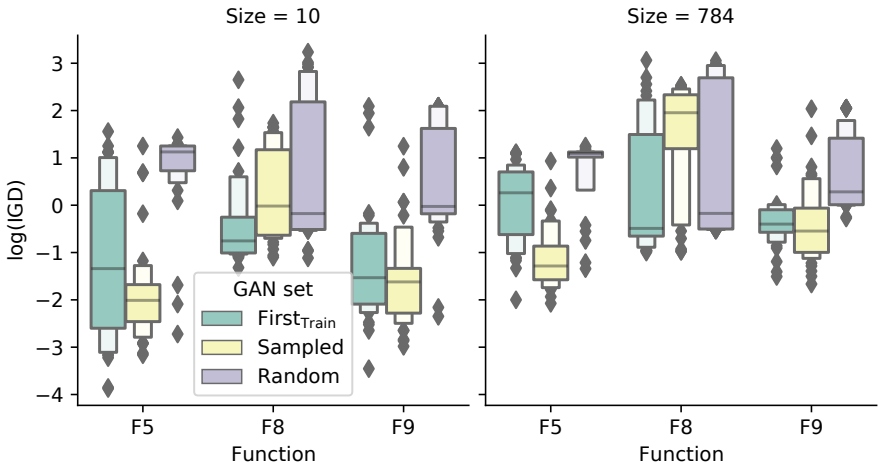


Fig. 4.4: IGD values obtained by GANs in the $\text{First}_{\text{train}}$ set, GANs sampled from the metamodel, and GANs from the $\text{Random}_{\text{train}}$ set.

As can be observed in the figure, the results vary depending on the function on which the test is performed, and are consistent through problem size. Regarding F5, the functions for which the lower IGD values were recorded, the sampled GANs performed surprisingly well, clearly outperforming the random GANs, and arguably performing better than the GANs chosen from $\text{First}_{\text{train}}$. This is not the case for the more difficult function F8, for which the best GANs yielded the best results, and the random GANs offered similar results to the sampled ones. The experiment performed using F9 was not able to discern which one was the best approach between the best and the sampled GANs, while both of these sets clearly outperformed the random GANs. The Kruskal-Wallis test reported significant differences in all 6 trios of IGDs. The Dunn test produced p-values under 3×10^{-7} for all pairwise comparisons for F5 except when comparing the `First` and `Sampled` GANs with 10 variables, where the p-value was 0.12. This means that, in the worst cases, the sampled GANs performed as well as the best ones, if not better. The random ones did not offer competitive results. For F8, comparisons between random and sampled GANs yielded p-values of 0.36 and 0.96 for 10 and 784 variables respectively, whereas all other comparisons resulted in p-values smaller than 0.002. In this case, the sampled GANs did not offer better results than random ones, and both sets of GANs were one step below the best GANs. Finally, for F9, the comparison between best and sampled GANs showed no clear differences (p-values of ~ 0.2), while the other comparisons resulted in p-values below 5×10^{-8}. Similar conclusions to the ones extracted from F5 can be deduced in this case.

As a general conclusion of this analysis, we could say that, in terms of the IGD, (the metric that was minimized during the evolution of the GANs in `First`) the sampled GANs sometimes offer worse results than random GANs would (F8 with 784

variables). However, in most cases, the sampled GANs offer a similar performance to GANs in `First` (F9, or F8 with 10 variables), if not better (F5). Interestingly, a small selection of random structures was able to provide a similar performance to the more sophisticated ones, which indicates that efficient learning procedures can also be carried out even when the GAN structure is randomly constructed.

4.3.3 Metamodel for improving GAN structural searches

In the previous section, the ability of generating good-performing GANs from the proposed metamodel has been shown, away from a NAS environment. Now, that capacity is tested in a NE setting, one of the two main goals of the proposed approach. With that goal in mind, we propose different ways of initializing individuals in other NE algorithms applied to a completely different problem. For this purpose, we design a GAN structural search for the 2D 8 and 25-Gaussian approximation problems [93], which is specifically formulated to expose mode collapsing GANs. Thus, we set a reduced version of the evolutionary process proposed in the previous chapter, consisting of populations of 20 individuals and 20 generations. Three versions of the NE algorithm are run, the difference between them being the way the initial population is created. As in the previous section, the `First`, `Sampled`, and `Random` sets are defined, although this time no runs are left out when learning the metamodel (previously the runs corresponding to three functions were excluded in order to test its generalization capabilities). 30 runs of each evolutionary run were performed. Figure 4.5 shows the per generation evolution of the best GAN in terms of Maximum Mean Discrepancy (MMD) [50], the fitness function used to evaluate the quality of the generations of a GAN (the second objective of the bi-objective evolutionary process being the minimization of the elapsed time during training and sampling the GANs). The MMD improvements for $2\ dimensions \times 3\ initializations = 6$ run types are shown.

As can be seen in the figure, the evolutionary runs with the non-random initialization have a large advantage in the initial stages of the evolution, as could have been expected. Even though it is true that the randomly initialized runs experience larger gains over the course of the procedure compared to the other two algorithms, it cannot outperform them. Something similar can be said about the runs initialized by the GANs in `First` and `Sampled`, as the best GANs found in each step of the runs initialized with the `Sampled` set are not outperformed by any of the found architectures in the runs initialized with GANs in `First`. Additionally, the runs employing the `Sampled` set require much fewer steps to reach the top performing models, ~ 12, resulting in large savings in terms of computational time.

This figure clearly shows the advantages of using the metamodel for initializing the first population of a NE procedure. Instead of using GANs which were specifically evolved for diverse tasks, learning the more general characteristics of GANs evolved across different runs for different problems provides the metamodel-aided approach with the necessary generalization capacities. This could be due to the structures in `First` *overfitting* the problem they were evolved for. Because the metamodel cannot learn to generate GANs from `First` exactly as they are, and it has to

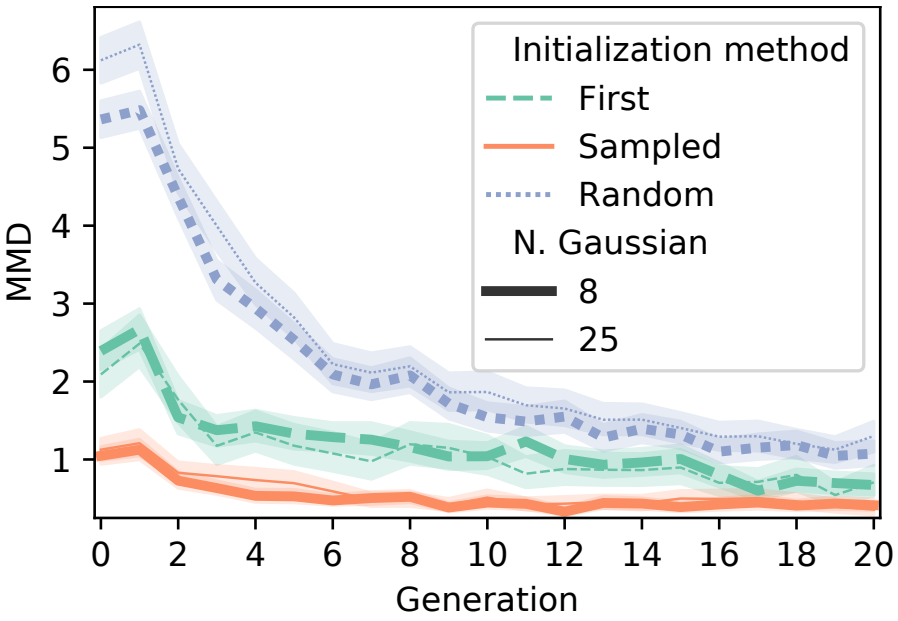


Fig. 4.5: Best MMD value (y axis) corresponding to the samples generated by GANs at different generations (x axis) during the evolutionary procedures. The solid lines represent the mean of the 20 $individuals \times generation \times 30\ runs = 600$ computed MMDs at each generation, whereas the translucent bands show the 95% confidence interval of all these runs. Runs for the 2 variations of the problem and three NE initialization methods are displayed.

focus on *only* capturing the dependencies within the structures that make them perform as they do, the GANs that it posteriorly produces are able to generalize better. This generalization capacity is a desired characteristic in many scenarios, including the one described in this experimentation.

4.3.4 Application to CNN-GANs

In the previous Section 4.3.2.2, the metamodel has shown its capacity to identify GAN structures according to how likely they are to offer a strong performance. NAS algorithms would greatly benefit by the usage of a model that could indicate the structures that are likely to perform poorly, avoiding unnecessary evaluations, and thus speeding up the whole process. GANs composed of CNNs are especially costly to be evaluated, and we therefore consider this special case of NE as a suitable field to test the capacity of the model to make processes more agile, the second main goal of the metamodel.

First, COEGAN (See Section 3.6 for a short introduction or [24] for a more detailed description) is run 20 times to evolve GANs for a single database, Fashion

MNIST [138]. Because the two DNNs are evolved separately, and they are formed of no more than 6 layers, only 12 BN submodels have to be learned [3]. Next, a `First` set of GANs was created, and a metamodel similar to that introduced in Section 4.2.2 was learned. Because of the reduced number of runs, we use $n = 20$ for a total of 400 structures, so that the metamodel has enough examples to learn from.

To test the validity of the model, 30 different runs of two variants of a hill climbing (HC) algorithm are executed looking for GAN structures which can accurately reproduce images similar to the digits available in the MNIST dataset [77]. Each HC run is awarded a limit of 100 evaluations, and the difference between the two variants resides in the usage of the metamodel to guide the direction in which the algorithm will move. While the first variant will simply randomly generate a neighbor of the current GAN, the second variant generates all the possible neighbors(in both cases generated using the mutation operators defined for COEGAN), checks which one is the most likely to make the largest immediate improvement based on the probabilities assigned by the metamodel, and chooses it as a candidate.

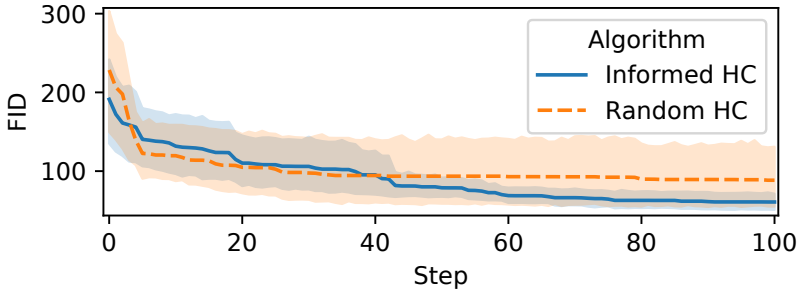


Fig. 4.6: Evolution of the FID (in the y axis) of the best CNN-GAN structure found at each step (x axis). The solid line represents the median of all 30 runs. The translucent bands represent the 95% confidence interval. The broken orange line represents runs performed with the random hill climbing, whereas the blue continuous line does so for the hill climbing guided by the metamodel.

Figure 4.6 displays, for each step in the HC procedure (x axis), the best found FID value (y axis). Similarly to Figure 4.5, the solid lines show the median run, and the translucent bands represent the 95% confidence interval. The figure shows that, during the first 40 steps of the search, both HC procedures behave similarly, with a slight advantage for the random search. In the second part of the search, however, only the guided greedy algorithm is capable of showing steady improvement, whereas most of the random HC runs become stagnant from the 40-th step onward. This displays the benefit of a smart guidance during this search, as apparently its usefulness is at its peak when the algorithm is near convergence, a critical phase of any search. Some

[3] All the DNNs of this section have been implemented using the PyTorch library [107] by the authors of [24].

samples extracted from the best GAN (in terms of Frechét inception distance [54]) are shown in Figure 4.7.

Fig. 4.7: Examples of digits generated by the best structure found by the informed hill climbing runs.

4.4 Conclusions

In this chapter, we have proposed a methodology that makes the most out of the computational effort performed during a neural architecture search. We tested the methodology on two neuroevolutionary algorithms (the one described in Chapter 3 and the model described in Section 4.2 that search for GAN structures whose success has previously been reported in the literature. This proposal consists of modeling the best architectures found in optimization procedures with a metamodel based on Bayesian networks, with two main goals.

First, by training the metamodel with the best individuals found during previous evolutionary procedures, the model has been able to recognize networks which belong to the distribution of GANs likely to deliver a top performance. This way, posterior neural architecture searches, by ignoring individuals unlikely to offer a top performance, could lighten the computational effort required to be carried out. Secondly, the metamodel has been able to sample GAN specifications which are likely to produce a good performance without any structural optimization being performed on them.

We first successfully tested these two capacities of the proposed metamodel without having them participate in a structural optimization process, and posteriorly put them into practice using two successful neural architecture search procedures. Whereas in the first successful test, the metamodel was employed on the same (or similar) problem, the following two tests were carried out in more diverse scenarios, resulting in a strong validation of the proposed methodology.

In the first case, the metamodel learned with GANs evolved for the PS approximation was used to initialize a population. This population was used to conduct a

secondary neuroevolutionary process which seeks GANs for the 2-D Gaussian mixture approximation problem. The NE searches conducted with the sampled GANs as a starting point clearly outperformed equivalent searches starting from the best found GANs in previous searches. Secondly, we also tested the capacity of the model to speed up a neural architecture optimization algorithm. This time, a metamodel learned from GANs evolved for reproducing samples from the Fashion-MNIST dataset was used to guide a local search for GANs whose goal is to reproduce MNIST data. The results showed how the extra information provided by the metamodel helps the search algorithm to keep finding increasingly better structures, where a random version of the same algorithm could not. Overall, it can be said that the employment of this kind of metamodels can positively affect future NAS runs in multiple ways, as well as provide decent results in scenarios in which no structure searches can be conducted.

As a general conclusion, the results reported in this work suggest a large potential for these metamodels, as a version learned from a rather modest variety of data was able to produce all these interesting findings. In some way, the potential for making NAS runs more efficient is explored in Chapter 6, in which a set of *rules* that guides a structural search is defined.

Part II

Efficient search of complex DNN structure spaces

5

Definition of a multi-DNN model for heterogeneous multi-task learning

5.1 Introduction

As it has been clearly demonstrated (both within and beyond the context of this dissertation), the performance of DNNs is highly dependent on its structure. While easier tasks can be solved with relative success using *simple* structures, as the problem complexity grows, so does the necessity of more sophisticated structures. Because of this, multi-network models that can cope with these complex tasks have been designed. One such complex area of study is that which has been investigated in the previous chapters: generative modeling [49, 55, 66]. Even though single DNN models have been proposed for this type of task, the two most popular DNN-based generative models are multi-network models: GANs [49] and VAEs [66] (the original definition of both being composed of two DNNs each). These two models share similarities in terms of structural composition, as they both are composed of two individual - albeit connected - DNNs. The structural design of these two models was originally hand-made according to a predefined goal: the encoder-decoder structure in the VAE, and the generator-discriminator routine in the GAN. Even though extensions of these models have already been proposed [4, 43], these approaches were developed in restrictive frameworks that *only* permitted relatively small, structure-wise modifications. These extensions of multi-network models consist of enhancing the structure using additional sub-networks that can serve different purposes to those already existing in the model.

In addition to multi-network proposals designed for high complexity tasks, other approaches attempt to combine more than one functionality to a single network. This problem is known as multi-task learning (MTL) [13], and consists of using one single model to learn to solve several tasks of similar domains [80, 94, 141], such as classification, regression or reinforcement learning (e.g., learning the inverse dynamics of a Robot [142], or news classification [140]). The usage of DNNs to manage this kind of problems has produced interesting approaches in a very wide range of domains, e.g., using a single *supernetwork* to automatically play different Atari games with reinforcement learning [35], or classifying characters from different alphabets [80] employing combinations of *convolutional cells*.

Furthermore, employing the MTL approach has proven beneficial in different aspects. By sharing a subset of parameters of the DNN, the training algorithm has fewer variables to optimize, saving time and computational resources. What is more, the usage of the same parameters for multiple tasks makes each task act as a regularizer for the others, as the same parameters have to generalize to two (or more) problems, which lowers the risk of overfitting. Finally, it has been theoretically proven that the value of the loss function of a task within an MTL framework tends to the same value that it would have obtained had it been learned separately, as the number of training observations increase [84].

In this chapter, we address another, more general class of MTL, in which the different tasks not necessarily have to belong to the same domain (e.g., class prediction **and** data generation) and are solved simultaneously, rather than sequentially. Some works have already made some insights in this matter by employing variations of existing models, coining the Joint Learning term [79, 143]. In this conjuncture, we take a step forward, introducing a model that can be composed of several interconnected DNNs of different types, capable of handling the heterogeneity in the set of tasks of the heterogeneous MTL (HMTL). We understand that such complex combinations of tasks require an advancement in the multi-network model design; a step forward in the automated generation of this kind of models.

With the goal of extracting the positive aspects of all the ideas and research fields introduced above, we focus our efforts on providing a modeling scheme consisting of DNN building blocks placed in an interconnected structure, flexible and scalable, so that the model structure could easily be optimized; the VALP.

The main contribution of this chapter is twofold: (i) We provide a formal definition of the VALP as a general neural-network-based model to deal with HMTL. This high-abstraction definition aims at setting as few constraints as possible in terms of structural flexibility when designing a VALP implementation. (ii) From the abstract formalization, we present a functional framework to illustrate the potential of the model. It is accompanied by an example of how a model instantiation which has to deal with three tasks of different characteristics can be created using the VALP definition. Optimizing the structure of a model that can cope with such an extensive variety of problems is not a trivial task, and it will not be discussed in this chapter, but in the next one.

The rest of the chapter is organized as follows: We introduce the VALP definition in Section 5.2. We then identify a set of components that can be included into any VALP implementation in Section 5.3. In Section 5.4, we provide an illustrative example of a VALP instantiation. This is followed by the experimentation part that shows the viability of the VALP for the HMTL problem in Section 5.5. Regarding these results, we continue with a detailed future work part, in Section 5.6. Finally, conclusions drawn from this experimentation part can be found in Section 5.7.

5.2 VALP definition

Although traditional DNN models can be represented with rather simple notation, this notation shows various shortcomings when trying to describe sub-DNNs which are part of more complex models, particularly those that target different tasks simultaneously. Therefore, we present an enhanced notation, which will help to define and describe each and all the possible component and parameters of any VALP.

Definition 1. *A **data unit** is a pair $d = (\boldsymbol{v_d}, t_d)$. $\boldsymbol{v_d} = \langle v_d^0, v_d^1, ..., v_d^k \rangle$ represents k data variables, which share the t_d type.*

Definition 2. *A **model input** is a data unit $i_j = (\boldsymbol{v_{i_j}}, t_{i_j})$ provided to the VALP.*

I is the set of all the inputs of the model: $I = \{i_0, i_1, ...i_p\}$.

Definition 3. *A **primary network** n_w is a a 5-tuple, $n_w = (i_{n_w}, f_w, a_w, p_w, o_{n_w})$ which defines a DNN and its context within the VALP. $i_{n_w} = \{i_{n_w}^0, i_{n_w}^1, ...i_{n_w}^x\}$ is a set containing the inputs of the network, where each $i_{n_w}^j$ is a data unit. f_w is a function representing how all $i_{n_w}^j$ are combined to form another data unit, the definitive input of n_w (e.g., concatenation). The value of $i_{n_w}^j$ can vary over the different phases of the model life cycle. a_w contains the type (e.g., Decoder) of the primary network, and p_w, its parametrization (hidden layer specification, e.g., number of neurons in each layer for MLPs, number of filters and their size, and the stride for CNNs). o_{n_w} is a data unit, and represents the output of n_w. It can be also considered an intra-model output.*

$N = \{n_0, n_1, ..., n_y\}$ is the set containing all the primary networks in the model.

Definition 4. *A **model output** is a pair $o_j = (\psi_j, f_{o_j})$. $\psi_j = \{\psi_j^0, \psi_j^1, ..., \psi_j^d\}$ is the set of data units that o_j receives from the networks in N. f_{o_j} represents how all ψ_j^i are combined to form the final j-th output of the model (a data unit), a functionality similar to f_w.*

O is the set of all model outputs; $O = \{o_0, o_1, ..., o_r\}$.

Definition 5. *A **VALP** is a 4-tuple $M = (V, A, L, P)$. $V = I \cup N \cup O$, represents the model components. A is a set of connections that determine how the model components are interconnected. $L = \{L_0, L_1, ..., L_q\}$ is a set of triples that defines how the model performance is assessed. $L_j = (l_j, p_{l_j}, \boldsymbol{g_j})$, where l_j represents a loss function, $p_{l_j} \in \bigcup_{n_w \in N} \{o_{n_w}\} \cup \bigcup_{0<j<|O|} \{f_{o_j}(\psi_j)\}$ is a prediction (a data unit) made by the model (note that it can be either an intra-model or a model output), and $\boldsymbol{g_j}$ is the ground truth that l_j uses to measure and improve the performance of the model with respect to a particular task.*

P represents the model hyperparametrization. It contains, at least, the parameters that specify how the different L_j are combined to form a single loss function that can be used to optimize all the tasks of the model in a single step.

Definition 6. *A **model connection** is defined as $c_j = (i_{c_j}, o_{c_j}, \boldsymbol{\psi_{c_j}})$, where $i_{c_j} \in \bigcup_{n_w \in N} \{o_{n_w}\} \cup I$ represents the data unit providing the information, and $o_{c_j} \in$*

$N \cup O$ represents the model component the information is delivered to. $\boldsymbol{\psi_{c_j}} = \langle \psi^0_{c_j}, \psi^1_{c_j}, ..., \psi^z_{c_j} \rangle$, $\psi^b_{c_j} \in \mathbb{Z} \mid 0 \leq \psi^b_{c_j} < |v_{i_{c_j}}|$ represents indices of the variables transported from the connection input i_{c_j} to the connection output o_{c_j}.

We define $A = \{c_0, c_1, ..., c_z\}$ as the set of all the connections of the model.

Taking into account how DNNs have been traditionally defined (their architecture type and the number of neurons, filters, etc.), the p_w and a_w components from the proposed notation would have been enough to fulfill these needs. However, when these networks are supposed to be parts of a larger model which has to manage as many connections and data types as the VALP, the rest of the components of the proposed notation are also required.

The V and A sets of the $VALP$ can be used to form a directed graph (digraph) $G = (V, A)$. In a digraph, the number of arcs (connections) ending in a vertex is called the indegree of that vertex (or node), whereas the number of arcs starting in a node is called the outdegree. Regarding these two characteristics, we differentiate three types of nodes in this digraph: (i) source nodes; those having an indegree value of 0, (ii) sink nodes; those with an outdegree value of 0, and (iii) internal nodes, which have both indegree and outdegree values strictly larger than 0.

In the VALP, the model connections are represented with arcs and there is a source node for each element in I, a sink node for each element in O, and an internal node for each element in N.

5.3 VALP instantiation

Once the VALP has been formally defined, we identify a collection of elements that can be part of an instance of a VALP model. More specifically, we enumerate two key components indispensable for the correct operation of a VALP instance: data types and primary networks. We would like to make clear that the following list neither intend to encompass all the items a VALP can be composed of, nor limit future additions to the pool of VALP components. In this regard, we expect the VALP to be able to embrace an extensive set of data types and a variety of primary networks.

5.3.1 Data types

The data type component in the data unit defined in the previous section presents an elegant manner to handle the heterogeneity required to the model in terms of data outputs. We therefore define four data types to which data units of a VALP could adhere to:

- Discrete : This data type consists of a vector codifying discrete values.
- Numeric : This data type consists of a vector of numeric values.
- Samples: Similarly to Numeric, this data type consists of a vector of numeric values.
- Blob: This data type consists of a matrix of d dimensions of numeric values.

Despite technically containing the same type of information, we choose to define Numeric, Samples, and Blob as different types to improve the expresiveness of the model. For example, at the time of setting restrictions when creating a VALP instance (e.g., guaranteeing that samples are provided where samples are supposed to be produced, and idem for other types, such as regression or classification), the data types will turn out useful. They will improve the simplicity of the restrictions as well as the understandability of the model.

5.3.2 Primary networks

Once we have identified a subset of data types that can be used within a VALP, we can similarly characterize a set of primary DNNs (or sub-DNNs) that can be included in the N component of a VALP.

- **Generic MLP, g:** A regular MLP that maps the provided input to an output. It can take any type of data unit as input. The data unit it produces can have different interpretations: numeric values (in any case) or samples (exclusively if it received samples). The activation function in the output layer is the identity function.
- **Discretizer, δ:** Similar to a regular MLP, this network takes data units of any type as input, and produces data units of the Discrete values type. Its goal is to discretize values, mainly for classification purposes. It has a softmax activation function in the last layer.
- **Decoder, d:** The decoder receives Numeric or Blob data units, interprets them as means and variances of a $\mathcal{N}(\mu, I \times \sigma)$, and uses samples generated from that distribution to produce Samples data units. Its internal structure is also an MLP. This concept is borrowed from the VAE, on which the generative capabilities of a VALP can be based.
- **Convolutional network, c:** This primary network exclusively consists of operations commonly found in Convolutional neural networks (CNNs): convolutional and pooling layers. It can only take and produce data of the type Blob. Its goal is to maximize the performance of a VALP instance when working with certain data structures (e.g., image or sequential data).
- **Transposed-convolutional decoder, t:** This primary network can be seen as a combination of c and d. It can only be composed of transposed-convolutional operations, and it can take Blob, Numeric (guaranteed), or Discrete (optional) values. It produces Samples.

Although CNN layers can be implemented as a combination of MLPs with shared weights [48], in order to bestow future uses of the VALP with the necessary tools to implement and describe models as easily as possible, we have decided to add the CNN networks, in their classical formulation comprising convolutional and pooling layers, as a possible component of VALPs, aiming at the best balance possible in the formality-usability trade-off.

5.3.3 Model loss function

A straightforward approach for training the model is to use the regular backpropagation algorithm combined with a variant of the stochastic gradient descent (SGD) algorithm. These techniques optimize a loss function defined on the parameters of a model, so that the performance of the model can be as close to perfection as possible. The VALP performs many approximations at a time, which means that various loss functions (with respect to both the model and intra-model outputs) need to be optimized in parallel. For a model that needs to optimize, for example, three different tasks (regression, classification, and data sampling), the following four kinds of loss functions have been identified as necessary.

- Regarding the sample-generation outputs of a model ($t_{o_0^d}$ =Samples), we need a loss function l_0 that can measure the likelihood of the model generating data with respect to the distribution we are interested in.
- Related to the regression output of the model ($t_{o_1^d}$ =Numeric), we need a metric l_1 that can compute the difference between two vectors of numeric values.
- For the classification outputs of a model ($t_{o_2^d}$ =Discrete), we need a loss function l_2 that can compare discrete outputs with predictions of the same type.
- Regarding the primary networks whose outputs are used in a t or d, we need a metric l_3 that forces that output to approximate a distribution that we can reproduce.

It is important to note that some of these loss functions could have larger magnitudes than others, thus hoarding the effectiveness of the SGD algorithm. This could lead to some loss functions being ignored by the algorithm, producing a defective model. To address this issue, the VALP model includes one hyperparameter β, which scales the different sub-loss functions. This approach is inspired by the β-VAE [12]. It contemplates a β parameter that scales the two sub-loss functions present in a common VAE.

5.3.3.1 Data unit combination

We also define example functions that can be used for the f_n and f_{o_i} presented in the VALP networks and outputs, respectively:

Example 1. Being $d^i = \{d_0^i, d_1^i\}$ a set of data units, we define the concatenation function as a function that receives a set of data units, and produces another one:
$\zeta(d^i, d^j) = \big(\langle v_{d^i}^0, v_{d^i}^1, ..., v_{d^i}^n, v_{d^j}^0, v_{d^j}^1, ..., v_{d^j}^m\rangle, t_{d^{i,j}}\big)$
And, similarly, the addition function:
$\Lambda(d^i, d^j) = \big(\langle v_{d^i}^0 + v_{d^j}^0, v_{d^i}^1 + v_{d^j}^1, ..., v_{d^i}^n + v_{d^j}^n\rangle, t_{d^{i,j}}\big), n = min(|v_{d^i}|, |v_{d^j}|\big)$.
In both cases, $(t_{d^{i,j}} = \text{Sam.}) \leftrightarrow ((t_{d^j} = \text{Sam.}) \vee (t_{d^i} = \text{Sam.}))$,
$(t_{d^{i,j}} = \text{Num.}) \leftrightarrow ((t_{d^j} = \text{Num.} \vee t_{d^i} = \text{Num.}) \wedge (t_{d^j} \neq \text{Sam.} \wedge t_{d^i} \neq \text{Sam.}))$,
$(t_{d^{i,j}} = \text{Discrete}) \leftrightarrow (t_{d^j} = \text{Discrete} \wedge t_{d^i} = \text{Discrete})$,
$(t_{d^{i,j}} = \text{Blob}) \leftrightarrow (t_{d^j} = \text{Blob} \wedge t_{d^i} = \text{Blob})$.
When $d^i = \{d_0^i\}$ consists of a single element, $f(d^i) = d_0^i, \forall f$.

5.4 VALP implementation

The study performed in this chapter considers an implementation[1] that consists of a reduced version of the general VALP defined in Section 5.2. It does not cover all its possibilities, but rather is an initial exploration with the aim of displaying its potential, and has therefore many extension possibilities. The reduced VALP version considered in this chapter contemplates *only* three of the data types introduced in the preceding section. We restrict all data types t_d of all the data units in a VALP to the following values $t_d \in \{\text{Numeric, Discrete, Samples}\}$, $\forall d \in I \cup \bigcup_{o_j \in O} \{o_j^d\} \cup \bigcup_{n_w \in N} (i^{n_w} \cup \{o_{n_w}\})$.

Accordingly, only a subset of networks can be part of the VALP model definition of this work: $a_w \in \{g, d, \delta\}, \forall w \in \mathbb{Z}, 0 \leq w < |N|$. All DNNs in this version of the VALP are based on multi-layer perceptron (MLP) architectures.

The Generic MLP essentially transforms information into a different *encoding*, therefore, it can serve as an encoder that complements a Decoder as in a VAE structure. To avoid defining a primary network with the sole functionality of encoding data, we allow the interpretation of the output of any Generic MLP (a vector of numeric values) as the μ and σ parameters a decoder needs.

Fig. 5.1 describes these primary networks and the way they are related to the different data types.

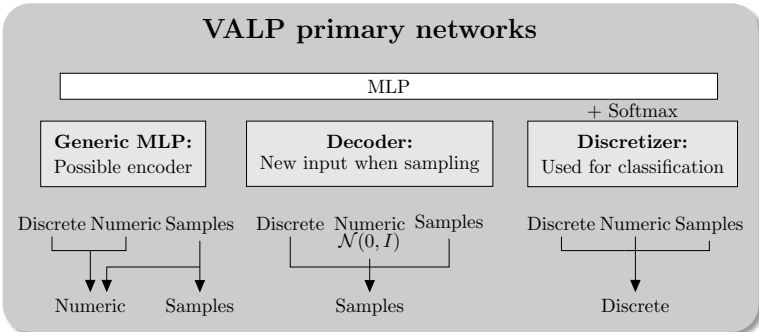


Fig. 5.1: Primary networks and their functionality inside VALP. The decoder must take (at least) numeric values when training. When running the model, these values are replaced with samples from a $\mathcal{N}(0, I)$ distribution to ensure that the Decoder can create new data.

Because we prime structural flexibility in the general VALP (and thus, in this example), the primary networks introduced in the previous section should be free to

[1] The code developed to perform the initial exploration in this chapter can be found in `https://github.com/unaigarciarena/VALP`

interact with each other in any possible way. However, in order to maintain type consistency (both in the output and throughout the model), a set of rules that restrict the model structure need to be imposed. In this regard, we force the decoders in a VALP configuration to have at least one Generic MLP primary network providing input to a decoder. This requirement is introduced due to the necessity of the decoders to have a numeric, optimizable input that can be trained to follow a certain distribution and can later be changed by new samples that follow that same distribution.

In Fig. 5.2 an example of a VALP model designed for solving three different tasks (sampling, regression, and classification) is shown. It is composed of five primary DNNs, it receives a single input i_0 (Data), and it provides three outputs, o_0, o_1, and o_2, where $t_{o_0^d}$ = Samples, $t_{o_1^d}$ = Numeric, and $t_{o_2^d}$ = Discrete. In this figure, circle nodes represent source nodes, triangular nodes represent internal nodes (networks), and square nodes are sink nodes.

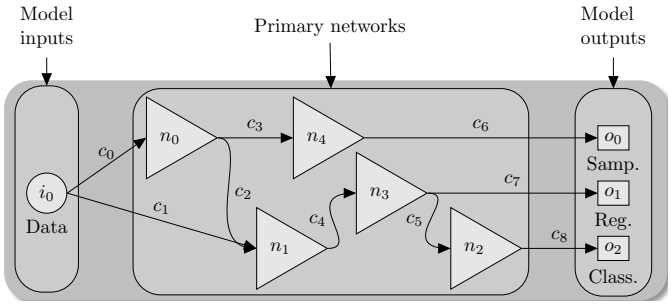


Fig. 5.2: Schematic representation of a VALP.

5.4.1 Formal definition of a VALP instance

We formally define the model instance $M = (V, A, L, P)$ shown in Fig. 5.2. This VALP example is required to produce numeric and discrete predictions, as well as a sampling output. We assume that we are working with a single dataset "Data", that is composed of 10 features, and where each example is labeled. The vector of these labels forms $\boldsymbol{C}$. Analogously, we have an $\boldsymbol{R}$ vector with a numeric value associated to each entry in the dataset. Finally, we have 5 extra features that we would like to reproduce (generate new samples), grouped in a vector $\boldsymbol{S}$ (one attached to each example in the Data, similarly to $\boldsymbol{R}$ and $\boldsymbol{C}$).

We define $I = \{i_0\}$. $i_0 = (\boldsymbol{v_{i_0}}, t_{i_0})$, where v_{i_0} are the 10 features of the database, and t_{i_0} = "Numeric".

We define $N = \{n_0, n_1, n_2, n_3, n_4\}$.

Because all the network types a_w can take are based on MLPs, any p_w is composed of three vectors: $init = \langle init_0, init_1, ..., init_l \rangle$, which specifies the function used to randomly initialize the weights of a n_w, $act = \langle act_0, act_1, ..., act_l \rangle$, referring

to the activation functions in each layer, and $ns = \langle ns_0, ns_1, ..., ns_l \rangle$, the number of neurons in each layer. l is the number of layers in the primary network, excluding the input layer, which needs no parametrization, and including the output layer.

We do not define every aspect of each component in this VALP example for the sake of keeping this illustrative definition compact. We avoid definitions of elements that are repetitive or redundant and do not contribute to the further understanding of the model concept. For example, we do not define p_w for each n_w.

$n_0 = (i_{n_0}, f_0, a_0, p_0, o_{n_0})$, where $a_0 =$ Generic MLP, $i_{n_0} = \{i^0_{n_0}\}$, $v_{i^0_{n_0}} = \langle v^0_{i^0_{n_0}}, v^2_{i^0_{n_0}}, v^5_{i^0_{n_0}}, v^6_{i^0_{n_0}}, v^7_{i^0_{n_0}} \rangle$ (Note the correspondence later, when defining the connections), $t_{i^0_{n_0}} =$ Numeric, and $f_0 = \zeta$. $o_{n_0} = (v_{o_{n_0}}, t_{n_0})$, where $v_{o_{n_0}} = \langle v^0_{o_{n_0}}, v^1_{o_{n_0}}, ..., v^6_{o_{n_0}} \rangle$, $t_{o_{n_0}} =$ Numeric.

$n_1 = (i_{n_1}, f_1, a_1, p_1, o_{n_1})$, where $a_1 =$ Generic MLP, $i_{n_1} = \{i^0_{n_1}, i^1_{n_1}\}$, $t_{i^0_{n_1}} =$ Numeric, $t_{i^1_{n_1}} =$ Numeric, and $f_1 = \zeta$. $o_{n_1} = (v_{o_{n_1}}, t_{o_{n_1}})$, $t_{o_{n_1}} =$ Numeric.

$n_2 = (i_{n_2}, f_2, a_2, p_2, o_{n_2})$, $a_2 =$ Discretizer, $t_{o_{n_2}} =$ Discrete, $f_2 = \emptyset$

$n_3 = (i_{n_3}, f_3, a_3, p_3, o_{n_3})$, where $a_3 =$ Generic MLP, $f_3 = \zeta$

$n_4 = (i_{n_4}, f_4, a_4, p_4, o_{n_4})$, where $a_4 =$ Decoder, $t_{o_{n_4}} =$ Samples. Note that, when training the model, $i_{n_4} = \{i^0_{n_4} = o_{n_0}\}$, but that would change to $i_{n_4} = \left\{i^0_{n_4} = (x \sim \mathcal{N}(I, 0), \text{Numeric})\right\}$ at the time of running the model.

We define $O = \{o_0, o_1, o_2\}$.

$o_0 = (\psi_0, f_{o_0})$, where $\psi_0 = \{\psi^0_0 = o_{n_4}\}$, and $f_{o_0} = \Lambda$

$o_1 = (\psi_1, f_{o_1})$, where $\psi_1 = \{\psi^0_1 = o_{n_3}\}$, and $f_{o_1} = \Lambda$

$o_2 = (\psi_2, f_{o_2})$, where $\psi_2 = \{\psi^0_2 = o_{n_2}\}$, and $f_{o_2} = \Lambda$

$L = \{L_0, L_1, L_2, L_3\}$, where $L_0 = (l_0, p_{l_0}, \boldsymbol{g_0})$. l_0 is the log-likelihood function, $\boldsymbol{g_0}$ is the $\boldsymbol{S}$ data provided in the beginning of the problem definition, and $p_{l_0} = \zeta(\psi_0) = \psi^0_0 = o_{n_4}$. $L_1 = (l_1, p_{l_1}, \boldsymbol{g_1})$. l_1 is the mean squared error (MSE) function, g_1 is the $\boldsymbol{R}$ data provided in the beginning of the problem definition, and $p_{l_1} = o_{n_3}$. $L_2 = (l_2, p_{l_2}, \boldsymbol{g_2})$. l_2 is the cross entropy function, g_2 is the $\boldsymbol{C}$ data provided in the beginning of the problem definition, and $p_{l_2} = o_{n_2}$. $L_3 = (l_3, p_{l_3}, \boldsymbol{g_3})$. l_3 is the Kullback-Leibler divergence (KL), $g_3 \sim \mathcal{N}(0, I)$, and $p_{l_3} = o_{n_0}$.

The hyperparameter of the model is a tuple of a single element $P = (\beta)$, which parametrizes L. β is a set of tuples $\beta = \{(l_3, 0.5), (l_0, 0.8), (l_1, 0.9), (l_2, 1)\}$, where each tuple contains model components and a scalar. The loss functions defined in the model components and the scalar in the tuples are multiplied together, and then added up to form $0.5 \times l_3 + 0.8 \times l_0 + 0.9 \times l_1 + l_2$, the definitive loss function used to train the model.

We define $A = \langle c_0, c_1, ..., c_8 \rangle$.

$c_0 = (i_{c_0}, o_{c_0}, s_{c_0})$, where $i_{c_0} = i_0$, $o_{c_0} = n_0$, $s_{c_0} = \langle 0, 2, 5, 6, 7 \rangle$ (note the correspondence between this connection and the previously defined $v_{i^0_{n_0}}$).

$c_1 = (i_{c_1}, o_{c_1}, s_{c_1})$, where $i_{c_1} = i_0$, $o_{c_1} = n_1$. $c_2 = (i_{c_2}, o_{c_2}, s_{c_2})$, where $i_{c_2} = o_{n_0}$, $o_{c_2} = n_1$. $c_3 = (i_{c_3}, o_{c_3}, s_{c_3})$, where $i_{c_3} = o_{n_0}$, $o_{c_3} = n_4$. $c_4 = (i_{c_4}, o_{c_4}, s_{c_4})$, where $i_{c_4} = o_{n_1}$, $o_{c_4} = n_3$. $c_5 = (i_{c_5}, o_{c_5}, s_{c_5})$, where $i_{c_5} = o_{n_3}$, $o_{c_5} = n_2$. $c_6 = (i_{c_6}, o_{c_6}, s_{c_6})$, where $i_{c_6} = o_{n_4}$, $o_{c_6} = o_0$. $c_7 = (i_{c_7}, o_{c_7}, s_{c_7})$, where $i_{c_7} = o_{n_3}$, $o_{c_7} = o_1$. $c_8 = (i_{c_8}, o_{c_8}, s_{c_8})$, where $i_{c_8} = o_{n_2}$, $o_{c_8} = o_2$.

Although the introduced VALP model assumes fixed I and O, we can think of a scenario where this is not the case. In this regard, the VALP is not a static model structure-wise, as I and O can be expanded once the model has been constructed (and even learned), which would trigger additions in at least one of A or N, or both. For example, if we added a new o_j to O, we could update the model by adding a new n_w to N and two connections, one from $n_j, j \neq w$ to n_w and another one from n_w to o_j. Such a scenario could be particularly useful for incrementally learning VALP instances.

5.5 Testing the potential of a VALP

We designed an artificial problem to illustrate the potential of the VALP. We have selected the widely known Fashion-MNIST [138] dataset, which is more complex than MNIST, and has been extensively used for research on DNNs [21, 62]. This dataset consists of 60,000 train and 10,000 test images, which are 28 $\times$ 28 pixel, gray-scale images of clothing, each belonging to a certain class. There are 10 items of clothing overall; T-shirt, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, and Ankle boot. The usual supervised task associated to this dataset is to predict the class each example belongs to. The number of instances of each class falls near 10% of the total instances, which indicates that this is a balanced problem.

To illustrate the model instantiation at its fullest potential, we define the multitask fashion-MNIST problem as composed of three different tasks:

1. Classification: 10-class classification as usually defined in the fashion-MNIST problem.
2. Regression: Firstly, we have computed a histogram for each of the images regarding the gray-scale values, with 32 bins. Then, these histograms were scaled between 0 and 1. This way, we have a 32-value regression problem to solve.
3. Generation: The generation of samples similar to those given to the model in the input.

In Fig. 5.3, we show three examples of the data available in the fashion-MNIST dataset, with the values desired to be obtained for each one, in each task.

5.5.1 Model parameters

To initialize the model structure, the only information required is the number and types of inputs and outputs together with their corresponding dimensions. In our case, we can define $I = \{i_0\}$ and $O = \{o_0, o_1, o_2\}$, where $i_0 = (v_{i_0}, t_{i_o})$, $|v_{i_0}| = 28 \times 28 = 784$ and $t_{i_0} =$ Numeric. $o_0 = (\psi_0, f_{o_0})$ where $|\psi_0^0| = 32$, and $t_{\psi_0^0} =$ Numeric, $o_1 = (\psi_1, f_{o_1})$ where $|\psi_1^0| = 10$, and $t_{\psi_1^0} =$ Discrete, and $o_2 = (\psi_2, f_{o_2})$, where $|\psi_2^0| = 28 \times 28 = 784$ and $t_{\psi_2^0} =$ Samples. $f_{o_0} = f_{o_1} = f_{o_2} = \Lambda$.

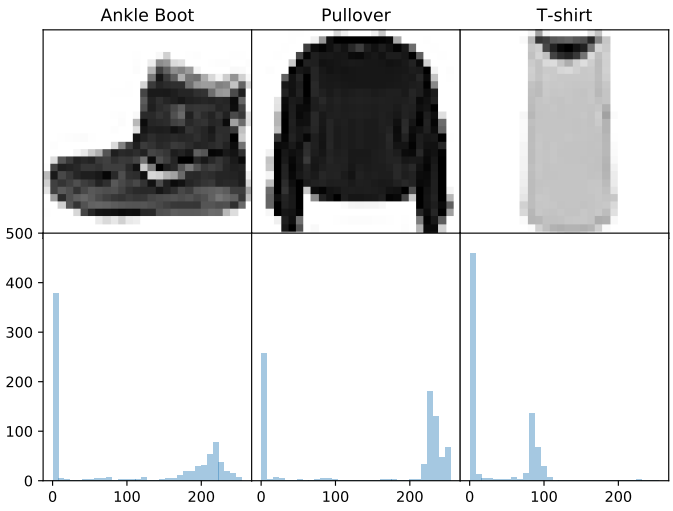


Fig. 5.3: This figure displays three gray-scale images that correspond to three different examples in the fashion-MNIST database. Their respective titles show what class they belong to (those used for the classification task), whereas the bar plots show the 32-bin-histograms representing the frequency (y axis) of pixel values (x axis) in the image (used for the regression task). The clothing image itself is used for the sampling task.

The next concern is to design a model i.e., the primary networks and connections in the model (as well as the loss functions and other hyperparameters), that can provide predictions for all required outputs, while matching the appropriate data type (Section 5.3.1).

The choice of the loss functions is a relevant decision to be made in this framework. As mentioned in Section 5.3.3, four widely used loss function types have been chosen for this problem:

- For the regression output, $L_0 = (l_0, p_{l_0}, \boldsymbol{g_0})$, where l_0 is the mean squared error between p_{l_0} and $\boldsymbol{g_0}$:

$$\underset{\theta^{VALP}}{\arg\min} \frac{1}{|p_{l_0}|} \sum (p_{l_0} - \boldsymbol{g_0})^2 \tag{5.1}$$

- For the classification output, $L_1 = (l_1, p_{l_1}, \boldsymbol{g_1})$, where l_1 is the cross entropy between p_{l_1} and $\boldsymbol{g_1}$:

$$\underset{\theta^{VALP}}{\arg\min} - \sum_x p_{l_1} \log \boldsymbol{g_1} \tag{5.2}$$

- For the sampling output, $L_2 = (l_2, p_{l_2}, \boldsymbol{g_2})$, where l_2 is the log-likelihood of p_{l_2} being $\boldsymbol{g_2}$:

$$\arg\min_{\theta^{VALP}} \mathbb{E}_{\mathbf{x}\sim \boldsymbol{g_2}}[\mathbb{E}_{q_\theta(\mathbf{z}|\mathbf{x})}[-log(p_{l_2})]] \tag{5.3}$$

where $q_\theta(\mathbf{z}|\mathbf{x})$ represents the probability distribution (predicted by a Generic MLP) inside the model, before any Decoder in a VALP.

- For each output of the generic MLPs whose output (p_{l_3}) feeds a decoder, $L_3 = (l_3, p_{l_3}, \boldsymbol{g_3})$, where l_3 is the KL:

$$\arg\min_{\theta^{VALP}} \mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})}[KL(p_{l_3}||\boldsymbol{g_3})] \tag{5.4}$$

As commonly, $\boldsymbol{g_3} \sim \mathcal{N}(0, I)$.

This model also considers different optimization pressures [12] on each of the terms of the global loss function. We add a scaling vector parameter (β), so that the optimization of the combined loss function is correctly performed.

5.5.2 VALP structure designing algorithm

For this example, we have designed a procedure for creating the VALP structure. As it will be later discussed in Chapter 6, the conception of efficient algorithms for designing VALP structures is an interesting open challenge.

The structure initialization algorithm employed in this work, which follows a back-to-front building approach, ensures that the adequate data type is provided to each model output. The strategy is based on an updated set of *model components* that have not had an input assigned; $act\_out \subset N \cup O$. The algorithm is a recursive function that incrementally and randomly develops the model until the maximum number of primary networks for the model (a parameter given at initialization) is met. It produces a configuration in which each component is guaranteed to have an input (except the model inputs), and the output types match the requirements. The model outputs can receive their predictions from a network that produces the required data type, while the networks can receive inputs from either another network, or a model input, i.e., the data. The only restriction applied to this algorithm is that a decoder must have at least one Generic MLP providing input. The reason behind this constraint is that the input of a decoder must be numeric and optimizable, as we require it to follow a certain continuous distribution ($\mathcal{N}(0, I)$).

The pseudo-code of this strategy is shown in Algorithm 5.1. It considers two parameters; max_n= 11, which handles the maximum number of primary networks in a VALP, and $\alpha = 0.5$, which regulates the reutilization of existing primary networks. More specifically, the α parameter is used to decide whether a source of data e_1 (if available) is used as an input for another component e_2 ($e_1 \in I \cup N$, $e_2 \in N \cup O$).

In addition to these parameters, this algorithm uses a set of auxiliary functions, which are explained below:

- complete_model(model): If $|N|$==max_n, then |act_out|==0, and the function does nothing. If $|N| <$ max_n, it searches for components that can serve the elements in act_out (in terms of data typing) and establishes connections between

```
1  Function initialize(A, I, N, O, act_out)
2      if (max_n − |N|) == |act_out ∩ O|
3          return complete_model(model)
4      con_out = random(O ∪ N)
5      found, con_in = random(I ∪ N, con_out)
6      if random_numb(0, 1)< α ∨ ¬found
7          con_in = create_rand_network(con_out)
8          N = N ∪ {con_in}
9          act_out = act_out ∪ {con_in}
10     A = A ∪ (con_in, con_out)
11     act_out = act_out − {con_out}
12     return initialize(A, I, N, O, act_out)
```

Alg. 5.1: VALP initializing algorithm.

them. If no such components exist, this function creates new primary networks, and uses them as bridges between the available components and those in act_out.
- random_numb(a, b): It returns a random number in $[a, b)$.
- random(set[, out]): If only the set parameter is provided, this function returns a random element from the set. If both parameters are given, it returns a random component from the set, such that it can serve as input to the out component $e_x | e_x \in N \cup O$ (in terms of typing, and not allowing recurrent connections) and found=$True$. If no such element exists, found=$False$.
- create_rand_network(con_out): This function creates a new network to be added to the model. The data type produced by that network must be compatible with the data type accepted by the con_out component.

The initial call to the recursive algorithm is $\texttt{initialize}(\{\}, I, O, \{\}, act\_out)$, where act_out is a copy of O, which represents the model components in need for an input.

The first *if* statement is the exit condition. In case it is not met, the algorithm selects a random component (con_out) that can take an input, i.e., a model output or a primary network. con_out will have a new input once the current recursion is finished. The algorithm searches for another component (con_in) that can provide a data unit which could serve as an input for the first component.

If no such element is found, or if a random number is lower than the α parameter, a new primary network (con_in) that can serve as input to the first component is created and added to N and act_out. Finally, con_out and con_in are connected. Because con_out now has an input, it cannot be part of act_out. This algorithm avoids recurrent connections.

For example, when using this algorithm to create the example displayed in Fig. 5.2, the first step would consist of $N = \{\}$ and act_out=$\{o_0, o_1, o_2\}$. In the first

recursion, n_3 could be added, a generic MLP, since it needs to provide a regression prediction. In the following recursion, act_out=$\{o_0, n_3, o_2\}$, because o_1 already has n_3 giving it an input, and n_3 would be added. This recursion would end up with at least three primary networks providing information to the outputs, and no component without a data input, act_out= $\{\}$.

Once the model structure is defined, the parameters of the primary networks (weights and biases) are randomly initialized and trained with regular backpropagation, taking as the global loss function a combination of the elements in L (according to the β parameter).

5.5.3 Experimental design

For the multitask fashion-MNIST problem, we performed a preliminary experiment and found that optimizing L_0, L_1, L_2 with $\beta_0, \beta_1, \beta_2 = 1$ simultaneously produced no undesirable effects. L_3, however, did present an obstacle when being optimized simultaneously with $L_0, L_1,$ and L_2. Many local optima for the KL divergence consisted of loss values higher than poor cost values for the other three loss functions. Therefore, L_3 tended to oscillate over those values once one of these local optima was reached. Because the optimizer often found larger potential improvement modifying the weights that alter the cost of L_3, it used to ignore the rest of the loss functions and weights. Therefore, we set $\beta_3 = 10^{-4}$. This way, we prevent the oscillating effect of the KL loss function rendering the optimizer futile once L_0, L_1, L_2 reach a value below that local optimum of L_3.

The max_n was arbitrarily set to 11. The mini-batch size for training the model was 50, and the model was trained for 40,000 batches (33 iterations of the whole database). Therefore, $P = (\beta = (\beta_0 = 1, \beta_1 = 1, \beta_2 = 1, \beta_3 = 10^{-4}), \alpha = 0.5, \text{max\_n} = 11)$. The DNNs can have as much as 10 layers, each of which can be composed of 100 neurons at most.

In order to have a glimpse of the performance of this VALP instantiation, the structural generation and learning procedures were randomly initialized and performed 500 times. This way, we perform a random search using Algorithm 5.1 to generate and train 500 different structures.

5.5.4 Experiments results

After training the 500 random VALP configurations with the train set, we employed each model to perform the described tasks considering the test set of the Fashion-MNIST dataset, and recorded different metric values to get a measurement of their performance.

5.5.4.1 Classification and regression

In Fig. 5.4, we can observe how the 500 models performed on the regression (MSE represented on the y axis in a logaritmic scale) and the classification problem (accuracy, on the x axis). Each point in the grid represents a random VALP configuration.

Analyzing the quality of such a complex model definition is not straightforward. To put these results in perspective, we also performed a more traditional search in the HMTL model architecture space in order to have a baseline to which the results obtained by the VALP can be compared. Assuming that one cannot have expert knowledge for any given problem, and that there is not a consensus on how to design DNNs, a generic way of designing a DNN is required so that the results obtained by the VALP can be contrasted. Facing this difficulty, we resort to a grid search for the structural design of a DNN as the baseline to which the VALP can be compared, as the grid search is a common choice in this kind of scenarios [41, 104, 126]. This search consists of testing several different hyper-parameter combinations to design a single DNN model which firstly has to deal with the Fashion-MNIST classification problem before being extended by adding another output for the regression problem. The grid search extends over the following options: The number of layers and neurons on them, the weight initialization functions used for giving initial values to the neurons, the activation functions applied after each layer, and whether that network employs batch-normalization and/or dropout. A description of the set of values considered for each of the hyper-parameters follows:

- Layer configuration: 2, 3, 4, or 5 hidden layers with gradually descending number of neurons (between each two consecutive layers in a DNN, the same number of neurons is reduced), starting from 784 (the input size) to 10 (the number of classes).
- Weight initialization functions: The weights of the DNN can be initialized using random uniform, random normal, xavier [46] uniform, or xavier normal distributions.
- Activation functions: eLU, ReLU, sigmoid, softmax, softplus, softsign, tanh, or linear activation functions can be applied after each layer in the DNN, with the exception of the last one, which mandatorily has to be softmax, due to the problem being dealt with is a classification one.
- Batch normalization: Whether batch normalization is applied before the output layer.
- Dropout: Whether dropout is applied before the output layer.

This results in $4 \times 4 \times 8 \times 2 \times 2 = 512$ combinations, which is very close to the 500 VALPs generated on the other part of the experiment.

From all the models tested, the one with the best accuracy is selected, and retrained now with a new additional loss function in a new output, so that it can also deal with the regression problem. The best found model consists of a four-hidden-layer MLP, whose weights are initialized using xavier normal initialization, and with softplus activation functions after each layer, and batch normalization and dropout before the last layer. Once found, the model was run 30 times with different weight initializations (obtained with different seeds and xavier initializations) obtained a mean accuracy value of 0.79 for the classification problem, and a 0.0032 mean MSE for the regression problem. These two values are visualized too in Figure 5.4, as a vertical red line (classification accuracy) and horizontal green line (regression MSE).

Also, their corresponding confidence interval of 95% is shown as a more translucent area.

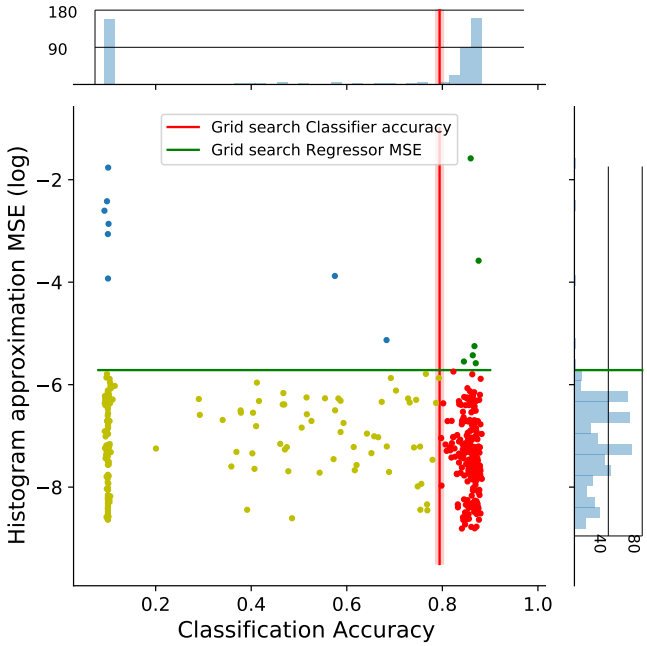


Fig. 5.4: Performances of the models plotted in terms of both classification accuracy and MSE reached on the regression problem. The vertical red line represents the performance of the MLP specifically learned for the classification problem. The horizontal green line represents the performance of that same MLP extended for addressing the regression problem. Blue points represent VALP configurations that performed worse than both baselines, yellow points are models that improved only the regression problem results, and models represented in green did so in the classification problem. The red ones offered better results than both baselines.

Regarding the classification problem, 280 VALPs (those represented as red and green points in Figure 5.4) improve the averaged accuracy achieved by the MLPs optimized with the grid search (79%, represented by the vertical red line). The superiority of VALPs is more remarkable when considering the regression problem, where 486 models (points in yellow and red) produced better results that the mean result obtained by the MLP configurations produced by the grid search (0.0032, represented by the horizontal green line).

While in general the VALP approach outperforms the baselines, the fact that VALPs architectures have been randomly generated can also be noticed in the clas-

sification and regression results. It is possible to observe that 158 VALP configurations produced a classification accuracy of nearly 10%. Considering that there are 10 classes and that the problem is balanced, this resembles random classification. The architectures producing these results probably have a decoder that deletes the path between i_0 and o_1 after training. The other weak outcomes are probably a result of similar structures, or models whose optimizing algorithm has focused on the other loss functions involved. With respect to the regression problem, we can observe that only 14 configurations performed poorly compared to the vast majority of the models; MSE superior to the baseline.

As a general remark on joint performance, and perhaps the main take-away message of this analysis, we can observe that 274 models performed better than both baselines (in classification and regression). The fact that a random search has a larger than 50% of probability of finding VALP configurations that have stronger performances in both objectives, while optimizing the sampling loss functions at the same time, shows, in our opinion, the validity of the proposal at tackling HMTL problems.

5.5.4.2 Generation

Once it has been shown that the VALP can perform the tasks classically attached to the *simple* DNNs, it is time to investigate the sampling capabilities of the model. Specifically, we want to extract information of two different aspects from the generative power of the model, which are related to the mode collapsing problem. Mode collapsing is an issue that concerns the generative modeling community, specially that part focused on GANs [118]. In the experimentation carried out in this chapter, we identify two types of mode collapsing. The *global* case, the worst one, is that in which all the generated samples look very similar to each other. In the *local* mode collapsing scenario, not as serious as the previous one, the model would learn to generate samples from the different classes of the dataset, yet those belonging to a certain class are still too similar to each other.

To test whether the model configurations suffered from global mode collapsing, an auxiliary DNN that classifies samples of the fashion-MNIST was trained. This DNN was used to classify the generated samples, giving a metric of how distributed the generations are class-wise. This DNN was based on the MobileNet model [59, 63], which was chosen due to the classification accuracy it obtained, followed by a single dense layer. This model reached a 99.5$\%$ accuracy on the training set and 94.5$\%$ on the test set of the Fashion-MNIST dataset, and it was used to classify the samples generated by each VALP configuration. Given that the classification problem being addressed is balanced, one could expect that the perfect model generated the same number of instances of each class. Because this is a 10-class problem, a perfect generator would generate an example of a certain class with 0.1 probability. Once the labels from the classifier were obtained, the capacity of the model for generating samples from different classes was measured as the entropy of the set of predicted labels. The higher the entropy value (which ranges between 0 and 1), the better the model is considered.

The class distribution can be visualized in Fig. 5.5. The histogram chart in the top represents the distribution of the entropy values of the 500 VALP configurations. We have chosen 7 representative examples from the whole set (pictured as vertical lines in the chart). The probability distributions of these examples are represented in the general parallel coordinates. In this example, we can observe how several VALP configurations (low entropy values) are poor data samplers, as they tend to generate images that MobileNet classifies as Pullover (class 2). This category usually consists of a set of pixels with high values in the middle of the image. The usual result of poor generators is producing images that consist of means of all the images which results in blurry images that posteriorly MobileNet classifies as Pullover. A similar effect happens with class 6 (Shirt). In contrast, the VALP configurations with higher entropy values (equal to or larger than 0.8) show much more distributed generation probabilities.

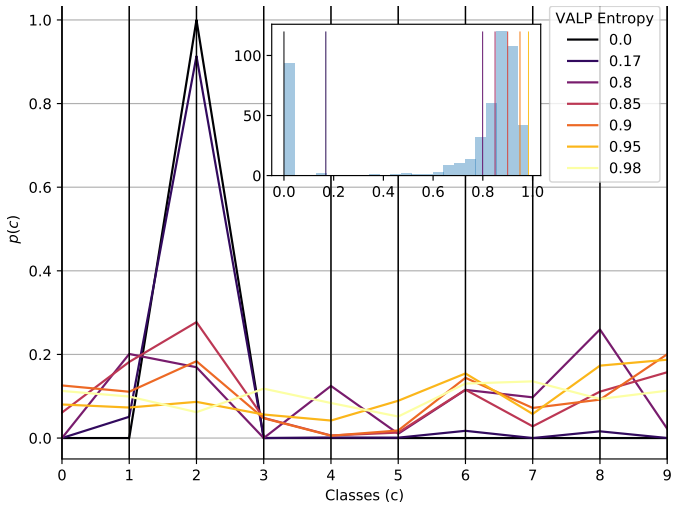


Fig. 5.5: The box in the top shows the distribution of entropy values of the 500 VALP configurations, and the seven representative VALP examples (the vertical lines). Each representative example is displayed by a vertical line. The probabilities of generating a sample of a certain class by each representative example are shown in the general parallel coordinates.

5.5.4.3 Conditioned sampling

Regarding the sampling capabilities of the VALP, we recognize that some problem domains could benefit by being able to *force* the model to create data that meets certain criteria. This is, rather than creating it from the whole known space, data is created from a specific sub-region of the distribution. By designing the VALP as

highly flexible as it is, we have made the VALP instances in this work capable of carrying out this specific kind of sampling. This is an interesting characteristic that can be exploited for multiple tasks, such as generating images with certain attributes [139] or reconstructing corrupted images [61].

To address this issue, we have generated 100 VALP configurations with at least one decoder which receives input from multiple sources. After the training phase, when using the VALP to create new samples, the common procedure (borrowed from the decoder) would be to *remove* all the information inputs to the decoder, and replace them with random values from a $\mathcal{N}(0, I)$ distribution. However, by guaranteeing that one decoder has multiple data sources, we can delete one of them, and maintain another one. This way, the decoder would have to generate new samples (because it is receiving random noise from one input), while also receiving some information about the data that is placed in the input. If a decoder had three or more inputs, a parameter ϕ that regulates the probability of a decoder not having an input deleted is added, and set to 0.5.

These 100 models have been trained as the previous 500 ones, and then used to generate samples conditioned by the examples in the test set.

To test how conditioned the samples can be towards the input of data given to the VALP, we computed a precision metric that evaluates the agreement (measured as an accuracy) between MobileNet's prediction for the test image used to condition the generation and the prediction for the corresponding VALP generated sample.

The results of this metric along with the entropy can be observed in Fig. 5.6. Each VALP configuration is represented by a point, and they are located in the grid regarding the class entropy and the conditioning accuracy defined in the previous paragraph. It can be observed that most models have a high entropy value, which means that there is not a strong global mode collapsing problem. Part of the figure has been cut out, as there were no models that generated *conditioning values* between ~ 0.1 and ~ 0.35. Additionally, this figure shows a heatmap representing the confusion matrix comparing the class that MobileNet predicted to the conditioning examples, and the class the Mobilenet predicted for the produced samples by the VALP represented with the red star (the one with the highest conditioning accuracy).

Firstly, we observe that *only* 16 models generated a class entropy value lower than 0.8, therefore, in that aspect, the models offered a good behavior. Fig. 5.6 also shows that only a small proportion of models (a total of 27) produced a conditioned accuracy value near 0.1. This means that, to some degree, almost three fourths of the models were able to condition the generated samples towards the image they were shown in the input. The configurations in the lower part of the left-hand side of the figure are the poor quality generators.

Regarding the heatmap, we can observe that this particular VALP found difficulties at generating samples of some classes, from which the 9-th and 4-th ones are the worst case. The VALP was unable to generate any sample from this category according to MobileNet. The next worst case is class 6, with 315 generations, and the rest have at least 824 representatives out of 10.000 generations (A perfect model would have produced 1.000 from each category).

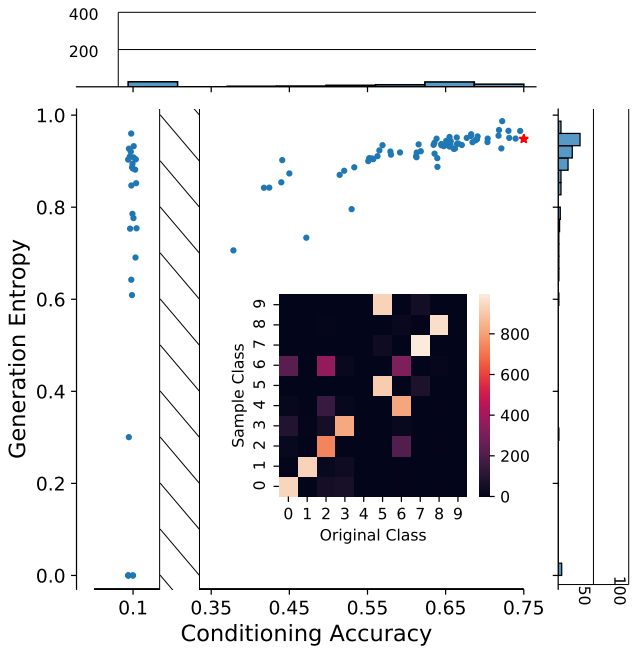


Fig. 5.6: Sampling performance of the 100 VALPs according to conditioning capability and distribution over the possible classes. The red star represents the VALP configuration with the largest conditioned accuracy. The small heatmap represents a confusion matrix, comparing the class of the conditioning examples and the generated sample images by the VALP represented with the red star.

Fig. 5.7 contains a schematic representation of the configuration of the VALP represented with the red star in Fig. 5.6, which, it is worth noting, achieved 85% and 0.00053 in classification accuracy and in regression MSE, respectively.

In this figure, we can observe that there are two decoders in the model, d_2 and d_8. The samples produced by d_8 *only* suffered one transformation before being used for o_2. The conditioning part happens with d_2, as it receives two inputs while training, and only one of them is deleted in the feed-forward phase (represented as a dotted blue line). This enables the VALP to recycle a piece of information introduced in i_0 using the path represented in dashed red arrows, to generate the samples, ultimately producing conditioned examples.

Determining whether a model suffers from local mode collapsing is harder than the global type. In Fig. 5.8, we have displayed 10 random generations of the model represented with the red star in Fig. 5.6 from each class they were conditioned towards.

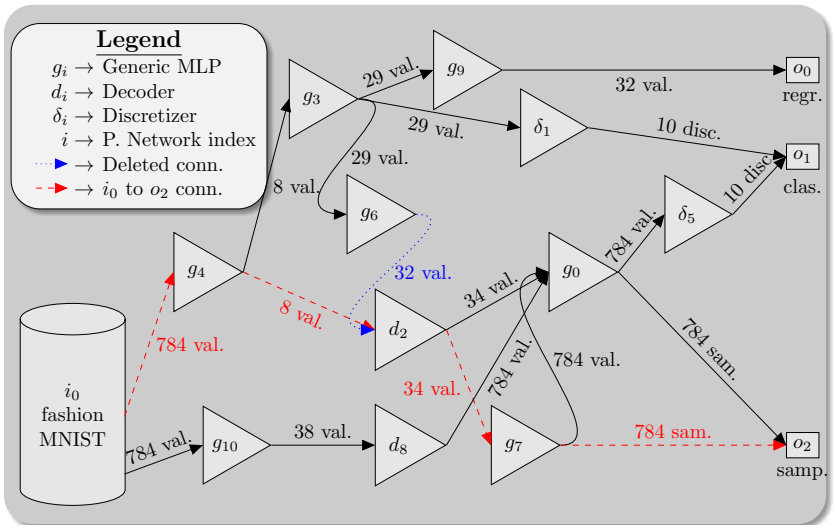


Fig. 5.7: Schematic representation of the VALP configuration that showed the maximum conditioning from the input example in the produced sample. The arrow tags represent the size of the data; how many variables there are, "val." stands for numeric values, "disc." are discrete values, and "sam." stands for samples. The dashed red lines denote the conditioning path. The dotted blue lines denote connections that are removed and replaced with $\mathcal{N}(0, I)$ when running the model.

From this figure, three different patterns can be identified. The first pattern consists of particularly weak samples. We can observe some blurry examples of Dresses. However, in most cases, it is possible to identify what class each image belongs to.

Then, it is possible to appreciate a second pattern, which are those classes for which the generated samples are easily identifiable, even though they look very similar to each other. A good example of this is the Trouser class, which generated characteristic trouser images, but which are very similar to each other.

Finally, there is the third pattern, which are those classes that had identifiable generated samples, while keeping differences between them. Examples of this *good* generation are the Sandal, Ankle Boot, and, in some cases, Bag classes.

Additionally, we can observe in this figure that, even though MobileNet failed to classify any generation as an Ankle Boot (classifying them as Sandal or Sneaker), the Ankle Boots were present among the generations of the model.

5.6 Open Challenges

After having empirically shown that the VALP is a viable approach to give a solution to the HMTL problem, the next step in the path is to identify the directions towards

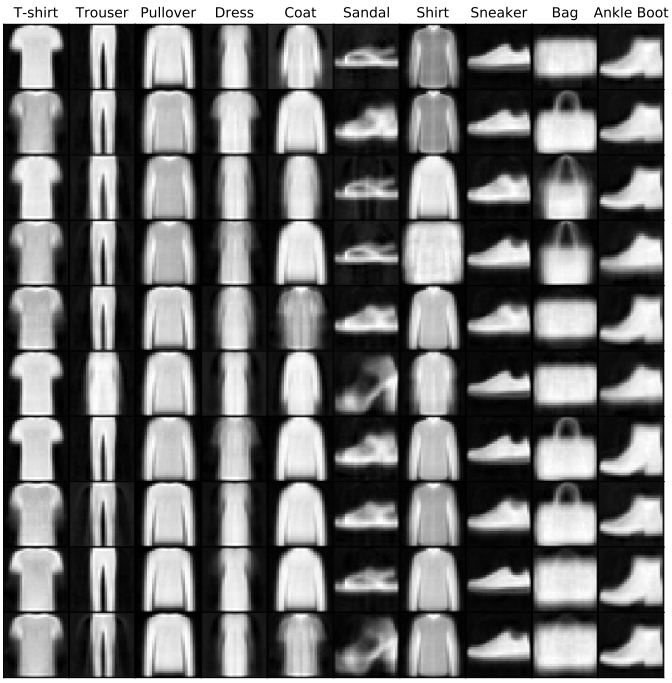


Fig. 5.8: Randomly chosen samples generated by the model represented with the red star in Fig. 5.6. The column names show the class the MobileNet assigned to them.

which the research over the VALP could be developed. In this section, we enumerate some of these research lines.

5.6.1 Structure search

We regard this extension of the study presented in this chapter as the most interesting one, and, accordingly, it is addressed in the following chapter. In that study, structural variation operator-based methods are considered. However, the investigation of several other NAS techniques can also be implemented or adapted to fit the VALP. We enumerate some of them.

The authors of [35] design a sizable DNN and collect sets of related and homogenous problems such as image recognition (MNIST [76], CIFAR [68], SVHN [101]) and reinforcement learning (several Atari games). The EA consists of a population of agents that determine a *path of neurons* across the randomly initialized DNN. Each individual is evaluated by training the parameters in the path of the network using the standard procedure -backpropagation- for the first task. Then, the next task is selected, and the weights of the connections that have been trained are frozen. Subsequently, the network is trained with the second task using another agent. This process is iteratively repeated until all tasks have been learned by the super network.

One of the most recent novelties in the area of automatically generating optimal DNN structures is the usage of reinforcement learning. The authors of [145] implement a RNN that predicts the parameters of an already fixed DNN structure. Each of the predictions of the RNN is fed to it in the next time-stamp, so that the hyperparameter setting is kept coherent across the whole design. The RNN is updated using the REINFORCE reinforcement learning technique over the error committed by the automatically designed DNNs. This approach is tested generating both CNNs and RNNs. The CIFAR10 dataset is used as a test to determine the quality of the CNN model, while the Penn Treebank (language modeling) dataset is chosen for the RNN. In both cases, the approach improved state-of-the-art results in the same conditions. The developed RNN cell is also proven to be transferable, as it also reached state-of-the-art results in character modeling in the same dataset. Using the same NAS technique, in [145], convolutional cells are evolved. The main contribution relies on the idea of evolving CNNs for *simpler* DBs and assuming that they can provide equally good performance when transplanted to a more difficult domain.

A research with a similar focus was carried out later in [146], based on [145]. In this case, convolutional cells were evolved using the same NAS technique, using as a reference point a *simple* DB; CIFAR10. Then, assuming the cell is optimal for the image recognition problem, and given the repeated motifs in hand-made state-of-the-art DNNs, a new *superstructure* that organizes repetitions of the optimal cells is developed. The authors compared the results to a random search (instead of the RNN that dictated the hyperparameters for each layer in the CNN), which also produced considerably good results comparing to other approaches to this problem. They conclude that the great performance by the random search was due to the optimal design of the search space.

5.6.2 New components

In the VALP architectures considered in this work, all the primary networks were fully connected layers sequentially placed. Future VALP variants could implement other architectures (some of which have already been identified in this work) that offer an excelling performance in different areas. One clear example would be adding recurrence to the VALP for an improved addressing of problems with sequential data. One way to incorporate this concept to a VALP instance would be to allow recurrent connections within the primary networks, which was not contemplated in the random structure search used in our example. A more straightforward (and even complementary) way would consist of designing new primary networks that contained recurrent connections within themselves, e.g., primary networks consisting of LSTM [45] or GRU [19] cells.

CNNs would also play an important role in VALP, as they would allow the model to maximize its performance in problems with image (or similar) data. These type of networks have already been defined in this work as primary networks of a VALP. However, they are yet to be evaluated in this context.

5.6.3 Loss functions

We selected one single loss function per task type, namely, cross-entropy for classification, mean squared error for regression, and the log-likelihood and KL for the sampling problem. However, more options exist for each of these tasks, and these could be included in the optimization process of the model as performed in Chapter 3, since some loss functions could have positive contributions towards the optimal training of the model [40] (for example, any of the GAN loss functions for the generative task [49]). The incorporation of the GAN concept would remove the necessity for a Generic MLP-Decoder structure in a sampling VALP. Not having a Decoder in a VALP configuration would mean not having to change the model structure (connection deletion) once it has been trained. This would be an upgrade in terms of flexibility.

Linked to the loss function selection topic, the weights applied to each of the sub-loss functions in the overall loss function have also been manually selected, after observing that the KL component could neutralize the effectiveness of the gradient descent algorithm. These parameter selection issues could be taken care of in the aforementioned optimization procedure, as a parameter tuning. Moreover, the multi-loss-function issue could also be addressed as a multi-objective problem of different VALP configurations that offer performances of varying quality over the different loss functions. A deeper insight in this matter is described in the next chapter.

Finally, this work has expanded MTL to the combination of different types of tasks: prediction (regression and classification), and data generation. Other popular problems, such as reinforcement learning, have not been included. Combining so many loss functions of different natures and training them all in the same model presents itself as a very challenging task.

5.7 Conclusions

In this chapter, we have addressed the heterogeneous variant of the multi-task learning problem; the HMTL. This problem consists of training a single model to perform several tasks simultaneously, these tasks being of different natures (e.g., regression, classification, and data sampling). To deal with this problem, we have proposed the innovative VALP model, a DNN-based approach. We have firstly provided a formal definition of the approach to lay the groundwork over which several different work directions can be developed. The main strength of the VALP is its capacity to manage different kinds of sub-DNNs and loss functions, which enables the model to produce different types of data that accurately approximate any distribution, using an optimization procedure over the different loss functions.

In this work, we have defined and focused on the fashion-MNIST HMTL problem, which consists of three different tasks (classification, regression, data generation). A particular VALP implementation has been designed to fit the particularities of this problem. A random search over the many-dimensional search space has

showed that the VALP can effectively and simultaneously carry out various tasks of different types, which also involve loss functions of completely different nature.

More specifically, we have found VALP configurations that were able to optimize the classical prediction tasks (classification and regression), while still producing reasonably good results at data sampling. Some configurations were even able to partially avoid one of the most concerning issues in the generative community, mode collapsing.

We have also enumerated a collection of detailed future research lines that the newly created VALP model can benefit from. Applying techniques that have produced high-quality results in other models to the VALP will help to determine where the strengths and limitations of the model lie.

However, the random search performed over the VALP structures is a mere demonstration of the potential of this model. Several more sophisticated options exist for this purpose. In the following chapter, we treat the question of efficient NAS, specifically applied to the VALP model.

6

Efficient exploration of a complex DNN structural search space

6.1 Introduction

As it has been mentioned in the previous chapters, the lack of efficiency is a flaw often held against NAS algorithms, as, commonly, assessing the quality of a DNN structure involves weight optimization procedures, which tend to be rather costly [10, 11, 87, 136]. The efficiency of operator-based NAS algorithms is largely dependent on the effectiveness of the operators they employ. However, that effectiveness could fluctuate depending on when or where they are applied. In other words, the most productive operator can end up being useless if applied in the wrong circumstances.

In this chapter we aim at opening a new research line in the direction of making variation operator-based NAS more efficient. In this context, we identify a large potential for improvement in these NAS methods, as most of them apply modification operators randomly, hoping for an improvement in the resulting model. Developing criteria to select the most suitable choice from a pool of operators, and the part of the model to be the target of that modification during the search would improve the efficiency of this kind of NAS algorithms.

More specifically, we attempt to illustrate the effectiveness of an *intelligent* NAS approach by reducing the random component that characterizes current structural search algorithms. With that goal in mind, we define a set of guidelines which can help a NAS algorithm to make an informed choice between all the variation operators at its disposal. These guidelines rely on a first step in which the model status is diagnosed using a set of metrics, which dictate the variation operator to be applied in order to improve the model in the second step.

Because our final goal is to propose an approach which can influence as many operator-based NAS algorithms as possible, we aim at improving NAS methods within a framework generic enough to encompass different types of neural models. The model described in the previous Chapter 5, the VALP, fits that characteristic, and we therefore base the study carried out in this chapter on that model. By proving the proposal of this chapter in such a complex environment as the HMTL, its application to other *simpler* scenarios -such as single task ones- is, in our opinion, guaranteed.

This chapter is organized as follows. Next section introduces the literature on which the proposal of this chapter is based. In Section 6.3, the ideas which are the main contribution of this chapter are described. These ideas are materialized into mechanisms for improving NAS runs, which are described in Section 6.4. The experiments designed for showing the potential of the proposal are presented in Section 6.5, and the obtained results are summarized and discussed in Section 6.6. Finally, Section 6.7 contains the conclusions drawn from this work, as well as some future research lines.

6.2 Background

As stated in the introduction, the approach presented in this chapter consists of the smart application of variation operators in order to increase the efficiency level of NAS algorithms, particularly -but not only- when applied to HMTL. In this section, we first introduce relevant work on the NAS area, classified by the type of search and operators they employ, so that the beneficial aspects of each type are identified. By integrating multiple perspectives on the NAS problem, an algorithm with access to multiple options from which to choose when making decisions can be designed. We next discuss some work performed on DNN model diagnosis, as the metrics proposed in these works will be useful to make an estimation of the role played by each sub-network (in the context of the experimentation of this chapter, a primary network of a VALP). This estimation can ultimately lead to informed decisions about the application of the correct operator to the correct location within the model, among all the possibilities imported from the different NAS approaches.

6.2.1 Neural architecture search

We limit this review to the approaches that have the largest influence in the proposed set of guidelines: neuroevolution (evolutionary algorithms, which usually rely on neural variation or mutation operators) and network morphism (NM).

6.2.1.1 Neuroevolution

The traditional approach to NE commonly considers relatively low-parametrized networks, both regarding the number of layers and the amount of neurons in them. However, as the hardware supporting DNNs has improved, these methods have shifted from performing low level modifications, e.g., the addition of one neuron or connection [119, 121], to more complex operations, like the concatenation of full neural cells (which can be considered sub-networks) to the DNN [80, 87]. This second kind of evolution has proven competitive against hand-crafted structures, and is the most popular approach considering the amount of recent work devoted to it. Currently, these two scopes of variation operators are known as micro (modifications limited to the small cells or sub-networks within the model) and macro search (altering the general structure of the neural model) [136].

The work in [111] presents an NE approach which adopts the NASNet search space (initially designed for a reinforcement learning based NAS) [146]. The authors propose the incorporation of an age property for all individuals in an NE procedure carried out in the NASNet space, in order to favor individuals of recent creation at the time of performing the tournament selection.

Some other approaches, while still being framed in the image treatment scheme, have variations especially relevant to the approach presented in this chapter. For example, in [87], NSGA-Net is proposed. This population-based algorithm permits the inclusion of (potentially conflicting) objectives other than the classic single error metric-minimization. This way, the authors address the problem of low efficiency on state-of-the-art NE algorithms by introducing a second objective which seeks the minimization of the computational complexity of the models.

The work presented in [17] introduces ModuleNet. This NE algorithm is largely inspired by [87], although new mutation operators are introduced. This NE algorithm is based on connecting sub-networks found in top-performing DNNs proposed in the literature.

These works, as well as many others not included in this review, mostly follow the same pattern: They introduce a framework different from those that have already been proposed, and design ad-hoc operators for it. In this chapter, we aim at forming a set of guidelines which are able to operate in different schemes in terms of model structure and problem domain by encompassing different types of operators and applying them when their positive impact towards the search can be maximized.

6.2.1.2 Network Morphism

Another research subfield which has grown separately, but is strongly related to NE (due to being it based on neural structure variation operators), is network morphism. It consists of a special set of operations for extending DNN structures in such a way that the performance of the network is not altered.

The work described in [16] proposes two operators for expanding DNN architectures, *Net2Net*, applicable to both MLP and CNN architectures. Their effectiveness is tested in a framework in which, initially, a relatively shallow and narrow *teacher* network is trained for the objective task. Next, one of the two operators; `Net2WiderNet` (enlarging the size of a layer) or `Net2DeeperNet` (introducing a new layer to the DNN), are applied to increment the number of parameters of the model. By conveniently initializing the added weights and/or the activation functions, the newly created *student* network is able to produce the same result as the *teacher* network. However, because its modeling power has increased, better results can be expected with further training.

These operators are first employed to gradually transform a shallow *teacher* into an inception network. Results show higher accuracy and faster convergence than training the same structure from scratch.

The authors of [134] then extended that work by resolving some of its inherent limitations: e.g., the inclusion of non-idempotent activation functions in the network modifications. Besides, the subnet adding operator was also presented, which

is equivalent to adding several layers at once. The authors finally used the *network morphism* term to name the framework containing these kinds of operators. This extension of *Net2Net* is able to outperform the original proposal both in learning speed and final accuracy.

The work in [33] takes full advantage of the framework defined in [134] and uses it as a tool for a Neural Architecture Search by Hillclimbing (NASH), using a simple structure as the starting point.

To the authors' knowledge, the research carried out using these operators is rather limited, considering the complementary role they can play for the more common operators usually employed for NAS. Because of this, we integrate them into the NAS framework which is governed by the guidelines proposed in this chapter, along with more traditional operators for NAS algorithms.

6.2.2 Model internal diagnosis

Studies attempting to understand the way DNNs operate have yielded many interesting approaches [3, 8, 116].

The authors of [116] propose the diagnosis of a neural model by computing the mutual information between the representations of the information found in the different layers of a DNN and the input and the output of the network during the training of the model. They concluded that, when trained with the common combination of stochastic gradient descent (SGD) and backpropagation, the weight optimization procedure of a DNN consists of two different phases, the information compression phase and the error minimization period.

In [3], a similar approach is presented. The information representation in each layer of a classification DNN is extracted for a set of observations, and a linear classifier is fitted between each of these representations and the original classes of the observations, independently. The errors reported by the classifiers from the different layers can serve as a measure of the quality of the information representation at each level of depth in a DNN. Because linear classifiers are rather limited and require rich representations of the data to perform well, it can be expected that, the deeper the layer -and therefore, the richer the representation-, the better a linear classifier will perform.

In the both previous approaches, [3, 116], comparing the values given by the metrics (the mutual information and the classification error) between layers can help to understand the role played by each layer within the general model context.

In an attempt to identify the origin of an issue in the DARTS [83] search space, the authors of [132] propose the deletion of different parts of a DNN, and using the observed performance decrease of the DNN as an estimation of the relevance of that part to the general model.

Although the explainability of the decisions made by DNN models is not among the objectives of the approach presented in this chapter, trying to estimate the role of a sub-network (named sub-DNNs indistinctly) within a neural model composed of multiple sub-networks is. In this chapter, we propose the exploitation of this kind of metrics so as to assess the role played by each sub-DNN to the general model in order

to determine the structural variations which have more potential for improvement in terms of model performance.

6.3 Intelligent search

Due to the costly nature of some NAS algorithms caused by the magnitude of the search space and the computational complexity of the weight optimization procedures, an efficient structural search of neural models based on sub-networks is crucial, especially when dealing with HMTL problems. This efficiency is mainly dependent on the operators integrated on the search algorithm, and, most importantly, how they are employed. In this section, critical aspects of different search methods are identified, before reflecting on how to exploit these characteristics in order to improve the efficiency of the search algorithms. First, we categorize the search methods by the number of neural models being taken into account at any given moment.

- Single model search: In this instance, the search algorithm consists of improving a single model at a time, as in a local search (e.g., hill climbing).
- Population based search: This second formulation considers several models at each time during the search.

6.3.1 Model internal diagnosis

The first key question in the proposed intelligent structural search is identifying *which* component or part of the structure (in our case, a sub-network) should be improved at a given point. To that end, we identify diverse sources of information depending on the type of search, which could help to make the right decision in this matter.

In a single model scenario, the main sources of information consist of:

1. Comparisons with the performance of models evaluated in previous iterations of the algorithm.
2. The relevance of the different components within the model to the final predictions made by the model.
3. The effectiveness of the training procedure to improve the prediction.

In the population-based search, along with these three information sources, other information sources are also available. These can be used to gauge the performance of a given model with respect to its peers, which can provide a more accurate idea of which component of the model, when modified, can provide a better gain in terms of model performance or loss function optimization.

6.3.2 Metrics

With the information sources identified, the second step is to determine how the knowledge is going to be captured. Focusing on the single model scenario, we formalize four different metrics:

1. *Historic sub-loss information:* Performance metrics extracted from the loss functions associated to the sub-network. The performance of a sub-DNN (or a subset of them) along time can be estimated comparing and combining metrics -e.g., the loss function- at each iteration.
2. *Module intervention:* Inspired by neural architecture selection methods [132], here we modify some element of a sub-network (e.g., setting the weights to random values) to estimate its importance as the loss in performance as a result of the intervention.
3. *Input intervention:* Similarly, it is possible to intentionally modify input values (i.e., a subset of the features of the data) and estimate their relevance with respect to the predictions.
4. *Dependency measures:* Metrics (such as the mutual information or a classification algorithm error [3]) between the output of each sub-network and the model output(s) it contributes to, would ideally improve the closer we get to the model output in the path followed by the information within the model [116]. If this is not the case, it could be interpreted that the component is not helpful.

For population-based approaches, we define an additional metric based on comparisons between models (although it could also be applied to the isolated model search by comparing the current model with other models in previous stages of the search).

5. *Relative performance:* several rankings -at least one per model output- can be arranged, according to the performance of the model in each output, relative to the rest of models. The position of a model in the ranking of a given output can determine the quality of the sub-networks related to that output.

6.3.3 Variation operator types

The third step is to define variation operators that cover the different needs that the models can present at different points during their development.

In this chapter, we categorize the variation operators according to two attributes: their *aggressiveness*, and the effect they have on the complexity of the model. Regarding aggressiveness, we distinguish two types of operators:

1. We consider an operator to be *aggressive* when it performs drastic alterations to the model structure, in such a way that the performance of the model can be severely changed in at least one of the objectives (e.g., operators commoly used in NAS).
2. On the contrary, an operator is considered as *gentle* if the performance of the model does not vary after its application (i.e., morphism operators).

When discriminating operators by the effect they have on the complexity of the model, we also divide the set of mutators into two subsets;

1. An operator is considered a *reducer* when its application decreases the number of weights in the model, and thus, theoretically, the modeling capacity.

2. Alternatively, an operator is an *extender* in case the model sees its number of weights increased.

Because we have two categories for each characteristic, we can define four type combinations. First, an aggressive extender operator would increase the number of weights of a model at the same time as the performance of the model is altered. For example, integrating a new random sub-network to the model and connecting it to other sub-DNNs which are ultimately connected to an output could alter the performance of the model in that output.

Secondly, a gentle extender operator would increase the modeling capacity of the model without modifying the performance of the model, e.g., by modifying other components already present in the model and cautiously designing and placing the new component.

Thirdly, an aggressive reducer would decrease the modeling capacity and have the collateral effect of altering the model performance, e.g., by deleting a sub-DNN or a connection which was relevant to the overall model.

Finally, a gentle reducer would delete certain parts of a model, without affecting the performance of the model. The deleted parts would need to be irrelevant to the model.

6.3.4 Donation operator

In population-based searches, mutation operators are not the only method to perform alterations to models. In this case, although they have been widely omitted by the NE community [39, 129], we define a special version of crossover, traditionally referenced as the *conjugation* operator [51]. In this method, one model, the donor, donates a part of itself (e.g., the output exclusive subgraph) to another model, the host.

6.3.5 Principles for using the metric information

In this section, we propose a set of criteria aiming at optimizing NAS procedures for HMTL models (although their application is not limited to that kind of models), exploiting the metrics defined in Section 6.3.2 to guide the selection of the variation and donation operators, as defined in Section 6.3.3 and Section 6.3.4.

6.3.5.1 Historic sub-loss information

This metric can be used to observe the behavior of one or more model outputs by fitting a linear regression model which attempts to predict the sub-loss value of an output, given the training step. This way, the slope of the loss function can be approximated with a line and, depending on that value, different approaches can be taken. For example:

- If the slope is close to 0 or positive, it can be concluded that the output has converged. In that case, an aggressive operator could take the model away from that local optima
- When the slope is slightly smaller than 0, it can be interpreted that the output is still improving, although a major improvement is unlikely. In this case, an extender gentle operator could add more modeling power, helping the model perform another significant gain without losing the current performance.
- In the final case, in which the slope is considerably smaller than 0, the output is still in the early phase of accuracy gaining, and should be left as it is until a certain level of convergence is reached, i.e., the previous two scenarios.

6.3.5.2 Module intervention

This metric can be used to determine the relevance of a given sub-network to the overall model by measuring the performance loss after resetting the weights of that sub-DNN.

- If the performance loss is not great for any output, the importance of the sub-DNN to the model is low, and a reducing operator is advised.
- On the contrary, if the drop off is significant, the component is assumed to be working as intended, and should either remain intact or be expanded using a gentle operator.
- Finally, if the sub-network is connected to an output which was not affected, a connection deletion would reduce the model complexity without deteriorating the overall model performance.

6.3.5.3 Input intervention

Similarly to Module intervention, this metric would estimate the importance of a given input to the final prediction of the model. This could be done by observing the performance change in the different outputs when randomly setting a subset of the features of the data:

- If the performance loss is not great, then the input is not very relevant to the output, and deleting a connection that connects the path between the input and the prediction would be advisable, so that the model graphically represents that *independence*.
- If a significant percentage of performance is lost, then the input is relevant to the output, and no connection should be deleted.

6.3.5.4 Dependency measures

As was the case for the module intervention, this metric serves the purpose of measuring the importance of a component for the model. In this case, a metric (e.g., the mutual information or the error of a linear estimator) is computed between a

model output and the outputs of the components on its subgraph. Next, for each sub-network, the obtained value is compared to the values of its predecessor in the model.

- When the measure indicates a larger dependency between the values, it can be assumed that the component is performing satisfactorily, and could be either gently expanded or left unchanged.
- If the value does not improve, the component is not performing as expected, and a reducer operator can be applied without losing much potential.

6.3.5.5 Relative performance

By constructing rankings of models according to their performance in the different outputs, it would be possible to estimate the relative performance of a model in that output. A model with all but one output in the higher part of their corresponding rankings could become the host of a subset of sub-networks related to an output from a model with a high rank in that specific output ranking. This vision is closely related to multi-objective optimization, as one model can be viewed as valuable or useless depending on different factors, like the output being evaluated, or the current state of the search algorithm.

6.4 Searching for optimal VALP structures using variation operators

The previous section presented a general approach and guidelines towards an intelligent structural search. In order to show its utility, this theoretical framework is implemented into the VALP NAS context. In what follows, we introduce variation operators which can be applied to the VALP, but could, generally, be applied to any other neural model based on sub-DNNs. We decided that the defined operators must comply with the characteristic of having to produce structurally valid VALPs as those defined in the previous chapter. In other words, it is guaranteed that the operators, when applied over a structurally valid VALP, will produce another structurally valid VALP. The operators are classified according to the characteristics described in Section 6.4.2 (aggressiveness and effect over the complexity of the model) and the scopes of application:

1. Sub-networks, operators used in micro searches.
2. General model structure, macro search operators used for modifying the connections between sub-DNNs.
3. Hyperparameters
4. Crossover operator

We understand that a scenario in which a set of weights is irrelevant to all the outputs of a model is highly unlikely in the context of the VALP, and therefore assume that a model losing some of its weights cannot maintain its performance. Consequently, we do not define any gentle reducer operator.

6.4.1 Sub-networks

We start with the operators with the most reduced performance scope (micro search): layer-wise modifications of the sub-networks in a VALP. Three different mutation operators have this scope:

- `add_layer`: This extender operator adds a layer in the network. Depending on how the weights are initialized and where it is added, this operator can be aggressive (e.g., by randomly initializing the weights) or gentle (e.g., by using the morphism approach).
- `remove_layer`: This operator deletes a layer from the network. The rest of the layers remain the same. As a reducer, this operator is aggressive.
- `extend_layer`: This operator adds neurons to a layer from the network. The remaining layers stay the same. This extender operator can be aggressive or gentle.

6.4.2 General model structure

The next set of operators is capable of affecting the VALP structure in its higher level (macro search), i.e., the interconnections between the different sub-DNNs in a VALP. We define five modifiers with this capacity:

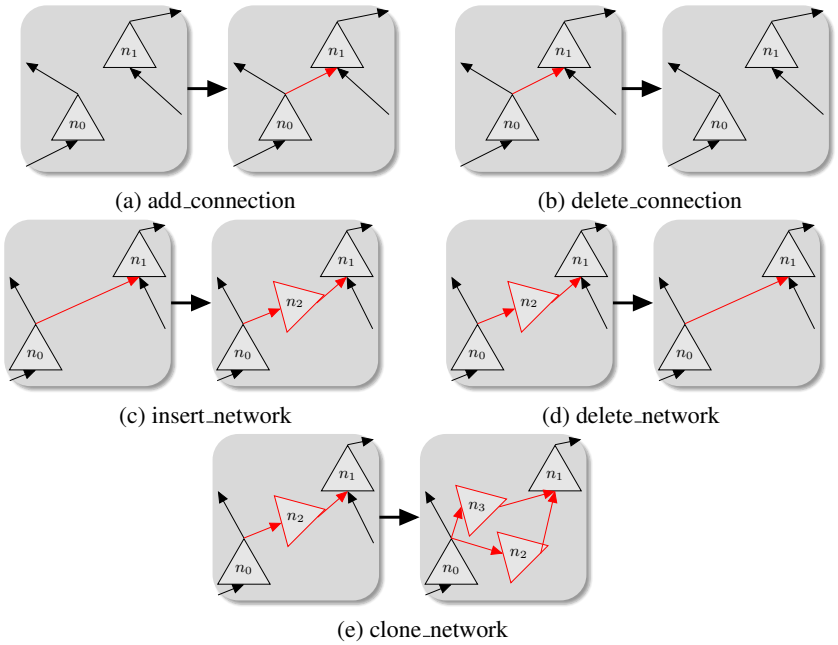
- `add_connection`: Given two currently unlinked sub-networks of a VALP, this operator links them by creating a new connection. In other words, the second sub-DNN receives the output of the first sub-network as additional input. This extender operator can be both gentle or aggressive.
- `delete_connection`: Given a connection of a VALP, this operator deletes it. This operator is aggressive and reducer.
- `insert_network`: Given a connection of a VALP, this operator inserts a network in the middle of the connection. For example, if a connection c_0 that links n_0 to n_1 is chosen, a connection c_1 between n_0 and the newly created n_m, and a connection c_2 between n_m and n_1 are created, and c_0 is deleted. This expander operator can be both aggressive or gentle.
- `delete_network`: Given a network n_m of a VALP, this operator deletes it. Each sub-network providing data to n_m switches to supplying data to each and every sub-DNN n_m provided data to. This operator is reducer (and thus, aggressive).
- `clone_network`: Given a network of a VALP, this operator duplicates that network and all the connections related to it. This operator is an expander and can be both gentle or aggressive.

These last five methods will be applied only if structural correctness (as defined in the previous chapter, i.e., complying with data type restrictions, guaranteeing that all DNNs receive and provide information, etc.) is guaranteed. For example, `delete_connection` will not, under any circumstances, delete a connection when it is the only source of data of a sub-network, or `delete_network` will never

suppress a sub-network when it is the only one between a model input and a model output.

The gentle operators defined in this chapter depend entirely on reusing and adequately modifying the weights optimized in the previous training epochs. We reuse the weights learned by a model before being altered, i.e., we apply weight inheritance, whenever it is viable (when a random sub-DNN is added to a VALP, no weight inheritance is possible).

Graphical examples of how these operators work are shown in Fig. 6.1.



(a) add_connection (b) delete_connection

(c) insert_network (d) delete_network

(e) clone_network

Fig. 6.1: Examples of the different operators. In all cases, the variation is performed relative to the VALP component in the middle of the figure (in red). For Figures 6.1a, 6.1b, and 6.1c, a connection. For Figures 6.1d, and 6.1e, a network (n_2).

6.4.3 Hyperparameters

Searching for the optimal model architecture (the sub-networks and how they are interconnected within the model) would only raise the model up to a certain point, as the rest of model components need to be synchronized to obtain an optimal performance. This is the case of the loss functions used to optimize the weights of the neural model, and other hyperparameters, such as the SGD algorithm. Other aspects related to training, such as the learning rate and batch size, also have to be properly

set. With this in mind, we define the following variation operators, all of which are gentle:

- `change_lr` changes the learning rate of a model output. For example, if convergence is detected in the *Historic sub-loss function information*, the learning rate of the loss function of that output can be decreased, aiming at improving the effectiveness of the training procedure over that specific objective.
- `change_sgd` changes the SGD algorithm used to optimize the weights of the model with respect to a model output.
- `change_bs` changes the size of the batch used at each training epoch.

6.4.4 Crossover operator

In this multi-objective scenario, each objective is independent of each other to some degree, as normally each output will have some exclusive sub-DNNs (and therefore, weights). Employing crossover-like operators enables parts to be cherry picked of models for constructing other models with the *best parts* of each one. We define a crossover operator based on the donation between models:

- *Exclusive subgraph* crossover: This aggressive operator can be applied when, based on the Relative Performance measure, a model which behaves adequately in multiple tasks fails at another one. A model with a top performance in that last task is selected as the donor of the exclusive subgraph of that output for the first model, the host, which has its exclusive subgraph replaced by the donation.

We define the *output subgraph* and *output exclusive subgraph* of the set of vertices G as follows.

The *output subgraph* of an output o_l consists of all the components that, upon modification, alter the prediction in o_l. The *output exclusive subgraph* consists of a similar subgraph, although in this case, the components that affect multiple outputs are not included in neither *output subgraph*.

Figure 6.2 shows two examples of subgraphs. The first one, colored in blue, is the exclusive subgraph correspondent to o_1. It contains three networks (n_1, n_3, n_5), all of them only ultimately connected to o_1. n_2, and n_6 cannot be part of that exclusive subgraph because they also provide data to o_2. The exclusive subgraph correspondent to o_2, in red, is composed of just n_4, as it is the only network exclusively connected to that output. The output subgraph of o_1 is composed of all the components ultimately connected to o_1, i.e., n_1, n_2, n_3, n_5, and n_6. Finally, n_4, n_6, and n_2 would form the output subgraph of o_2.

6.5 Experiments

We have designed a set of experiments in order to validate some of the general guidelines for the NAS framework proposed in this chapter.

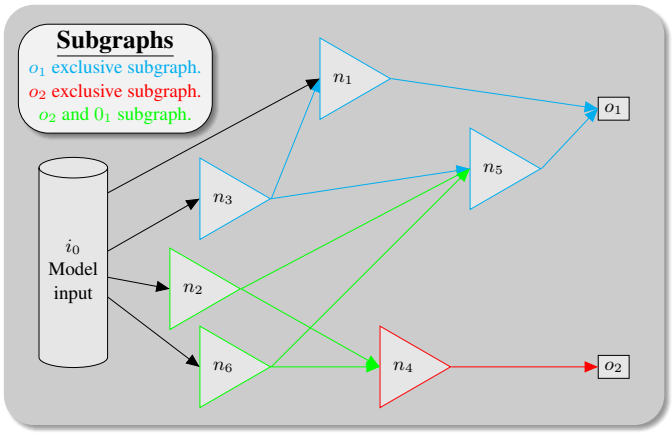


Fig. 6.2: Example of a VALP, with its different subgraphs. The n_1, n_3, and n_5 sub-DNNs would be part of the exclusive subset of o_1, as they are exclusively connected to o_1. Idem for the n_4 sub-network and o_2. Adding the n_2 and n_6 sub-DNNs to either subset would result in o_1 and o_2 subsets respectively, as these would contain all the sub-networks involved in these outputs, including sub-DNNs which are involved in other outputs.

Several works have reported that starting from *simple* neural models with relatively few weights and allowing them to evolve towards more complex structures yields positive results [86]. The experiments described in this section report results of the employment of the proposed search guidelines with this same mindset. We consider a model with a number of components close to the minimum (roughly one sub-network per model output) to provide the required output to be on its initial stages, whereas a mature model would consist of a more complex structure with more sub-networks and connections.

6.5.1 Test Benchmark

For the different parts of the experimentation, we use the multi-objective Fashion-MNIST problem described in the previous chapter, and also define the analogous MNIST [77] multi-objective problem. This way, we can test the performance of the operators when acting in an environment where the outputs are related to a single data input. Additionally, we simulate the scenario in which the rules are inferred from one set of experiments, and are then applied to another, more complex problem. The two problems having very similar characteristics in terms of number of examples and features as well as data type and number of classes is purely coincidental, as this approach could be tested in problems of varying data inputs, outputs, and characteristics of both.

6.5.2 Initial experimentation

The first step consists of testing the proposed metrics and operators isolated from the NAS framework. This way, we will be able to extract valuable information about the operators, and how to use the information given by the metrics with the final goal of deciding which operator and where it should be applied in a NAS process.

In order to assess the impact that each operator can have in different model scenarios (these scenarios being described by the values obtained from the different metrics), we perform an exploratory search over the space of *medium-sized* VALPs (i.e., twice as many sub-networks as model outputs). In this experimentation, we will be able to observe the difference between applying gentle mutation operators over their aggressive counterparts.

Additionally, and this is the main goal of this experimental section, we aim at setting the grounds of the set of rules which will be helpful to improve the efficiency of future NAS runs. To that end, we attempt to identify which operators offer the largest improvement potential. Because the rules we are looking for should not be tied to the particular problem used in this instance, we are relying on the metrics defined in Section 6.3.2 instead of the common metrics for assessing the performance of a prediction model (e.g., accuracy for a classification model).

Choosing the MNIST problem, we test the effect of the mutation operators defined in Section 6.4.2. To that end

1. 100 VALPs are randomly created and trained during $\sim$67 epochs (20,000 batches of size 200).
2. Every operator is applied to different clones of each VALP. The operators are applied to each component of the VALPs if and only if structural correctness is guaranteed.
3. Every VALP is retrained to adjust the weights of the model to the variation for $\sim$17 additional epochs (5,000 more batches).

To determine the quality of the VALPs at each point, we have evaluated them before the modification and after the secondary training.

6.5.3 Main experimentation

In this second step, we want to employ the knowledge obtained in the first step on a NAS procedure. With that goal in mind, we propose a common HC algorithm (Figure 6.1 contains a pseudo-code form of the method) with two different implementations: the common approach, in which operators are chosen randomly, and the smart approach, in which the most promising operator is chosen. The pseudo-code makes use of the following hyperparameter and functions:

- step_limit: This parameter sets the number of evaluations awarded to the algorithm. In this case, it is fixed to 60.

- **random_VALP()**: This function randomly initializes a VALP as described in the previous chapter, with a limited number of components.
- **evaluate(**$model$**)**: Given a VALP, this function evaluates the model and returns one value per model output. In this chapter, it consists of a triple, since the problem has three objectives. The chosen metrics are the classification accuracy, the MSE for the regression problem, and the FID [54] for the sampling output.
- **select_operator(**$model$**)**: Given a VALP, this function selects the operator to be applied and the target of that operator. The difference between the random and the *smart* approach resides in the implementation of this function. The application of the operator guarantees structural correctness.
- **variation(**$model, op$**)**: Given a VALP and an operator, this function generates a neighbor of the VALP by applying the operator.
- $<$: This operator compares two tuples of values. In this case, if at least two of the three values of the operand on the left are lower than their corresponding values of the operand on the right, it returns True. Otherwise, False is returned.

```
1  current ← random_VALP()
2  curr_fitness ← evaluate(current)
3  step ← 0
4  while step < step_limit
5      op ← select_operator(current)
6      candidate ← variation(current, op)
7      cand_fitness ← evaluate(candidate)
8      if cand_fitness < curr_fitness
9          current ← candidate
10         curr_fitness ← cand_fitness
11         step ← step + 1
12 return current, curr_fitness
```

Alg. 6.1: HC approach used in the experiments.

The two variants of the algorithm are tested on the (more difficult) Fashion-MNIST multiobjective problem. Using the start-simple-and-sophisticate approach, a random solution is initialized containing between one and two times as many subnetworks as model outputs. Then, both variations of the algorithm are applied to search for more complex VALP structures.

The initial VALP configurations used as the starting point is trained for 5.000 batches. At each step of the HC algorithm, the modified model is retrained for 1.000 additional batches. Each HC variation is run 30 times with different random seeds in order to avoid biased conclusions, product of the stochastic component of the method.

As in the initial experiment, we constrain the set of variation operators to be investigated to those defined in Section 6.4.2.

6.6 Results

6.6.1 Initial experimentation

First, we want to investigate whether the gentle operators consistently perform better than their aggressive counterparts. To that end, we have computed the improvement observed in the VALPs between the end of the first training step and after it has been modified and retrained. With the improvement measured -general loss function value after second training divided by performance after first training, both measurements in logarithmic scale- we subtract the improvement observed due to the application of gentle operators to the improvement caused by their corresponding aggressive counterparts. This metric G serves as a measure of the gain or advantage of using one class of operator over the other. Figure 6.3 shows the frequency (y axis, in logarithmic scale) of the G difference values (x axis). The more positive they are, the bigger the difference in favor of the gentle operator. Any difference superior to one is cut to that value to improve the visualization of the figure.

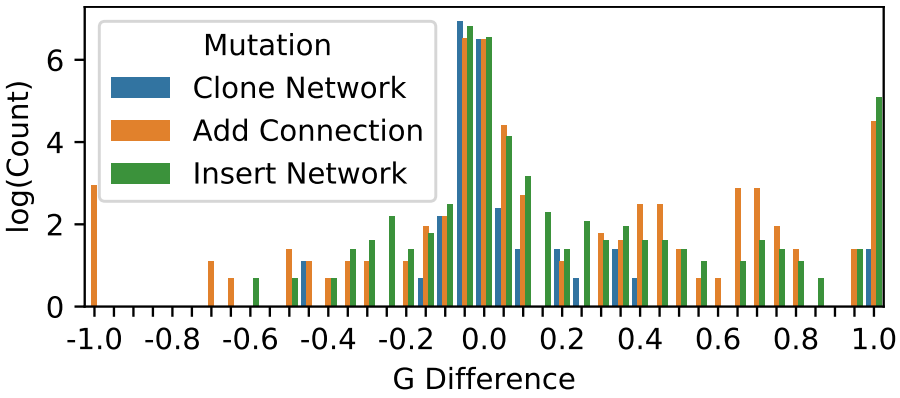


Fig. 6.3: Frequency (y axis, in logarithmic scale) of the G difference values (x axis) observed when comparing the improvements obtained by VALPs after being modified by gentle operators, and their aggressive versions. Positive values indicate a better performance of the gentle operator, whereas negative ones do the exact opposite. Larger numbers represent larger differences between operators.

As can be observed in Figure 6.3, the gentle operators have outperformed the aggressive ones considerably more frequently than the other way around, especially taking into account extreme differences (values over 1 and below -1). Gentle operators are, in general, conservative variations when it comes to increasing or decreasing

the performance of the model. In closer comparisons, the gentle operators also tend to produce the bigger improvement in model quality. However, although less frequent, there are many cases in which the aggressive operator had a more positive impact than their gentle counterparts. The *noise* these operators introduce into the model in the form of random weights, appears to be able to *shake* the model from a local optima, from which the gentle operator could not make it escape. This is especially visible in the cases in which extreme improvements were achieved by the aggressive operators (values under -1).

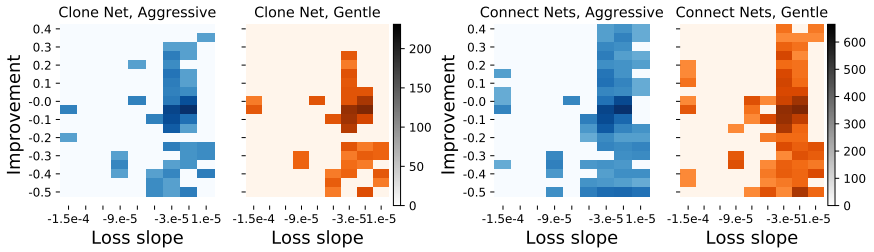
The presence of these last cases suggests that the employment of aggressive operators is not only viable, but advisable in some scenarios. This theory is also backed up by statistical testing. After the null hypothesis of all mutations producing the same effects being rejected by the Kruskal-Wallis statistical test [70], the Dunn post-hoc test [30] found significant differences between all pairwise comparisons between mutations -p-value < 0.0007- except for one, the comparison between the aggressive and gentle version of the connection adding operator. This is probably due to the high number of extreme differences in improvements. We now address the question of how to create the set of rules which helps NAS algorithms to correctly identify the *best operator* given one model.

Regarding the second part of this experimentation, we attempt to define the metrics that will eventually guide future NAS runs. With that goal in mind, we aim at observing, given the metric values (from those defined in Section 6.3.2), which operator(s) produced the largest gains. The two metrics which have produced the most significant differences among the analyzed mutation operators were the historic subloss information and the module intervention.

Regarding the loss function slope, Figure 6.4 shows the percentage of improvement observed in a VALP after it was modified by the gentle operators and their aggressive counterpart (in the y axis, in logarithmic scale), regarding the estimated slope of the loss function of the regression output of the VALP (x axis). The improvement percentages (the lower, the more improvement) have been cut to the $[-0.5, 0.5]$ range. The loss slope also only considers a minimum value of -0.00015. As can be seen in the figure, most improvements are marginal, just below the 0 mark. However, some interesting insights can be extracted.

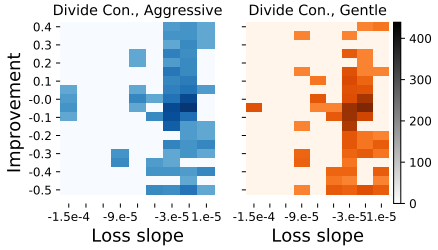
For example, because of the lack of existence of large performance decline when applying the clone net operator (Figure 6.4a) if the loss function is still decreasing (left-hand side of the figures), we can conclude that this is usually a beneficial mutation. The connection adding operator (Figure 6.4b) was also able to produce large performance gains when the loss function of an output is still steeply decreasing. This means that these kinds of changes are beneficial, especially the gentle form, when the loss function is still decreasing.

The mutation that places a network in the middle of a connection (Figure 6.4c) was able to produce significant changes (both improvement and deterioration) when the slope of the loss function is smaller, i.e., it is close to converging. This means that applying it on an output which has saddled in a poor local optima, the model can dramatically improve, while an abrupt performance loss would not hurt the model, as the local optima was not desirable anyway.



(a) Clone Net mutation operator.

(b) Add Connection mutation operator.

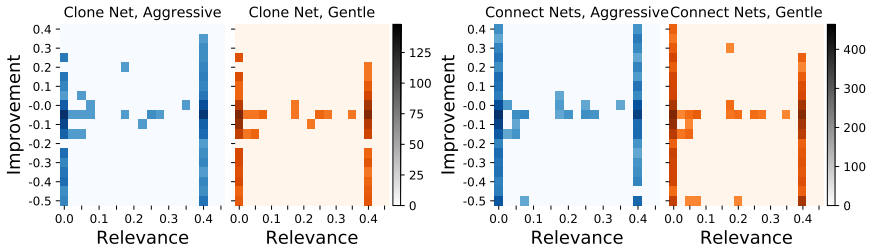


(c) Insert Network mutation operator.

Fig. 6.4: Percentage improvement observed over the regression output of the VALP (in the y axis, the lower the value, the more improvement) in logarithmic scale, by different mutation operators, with respect to the slope of the loss function evolution (x axis). The color darkness represents the number of mutations that registered the improvement in the y axis. The subfigures on the left-hand side represent data relative to the aggressive version of the operators, whereas the ones in the right-hand side show information about the gentle ones.

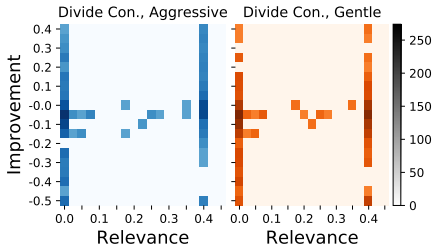
A similar set of figures has been generated for the network relevance metric. Relevance consists of the change observed between the two stages of the model, before and after being affected by the module intervention approach, and it is also measured in percentage points obtained by dividing the original metric value by the one obtained after the module intervention. This way, if no change was observed in a model output after being affected, a 1 is recorded. If the performance was halved (e.g., only half of the observations previously correctly classified are correctly classified after modifying the model), a 2 is recorded.

In the case of Figure 6.5, because the performance of a regression output can decrease indefinitely, the relevance has been cut to 0.4 (in logarithmic scale). As can be observed in the top right corner of the figures, when modifying a network relevant to an output, the result, as expected, can be very bad if the mutation is an aggressive one. The gentle network cloning appears to be a conservative choice when it comes to a relevant net, given the few cases in which performance declines have been observed. A similar effect can be observed with the insert network operator,



(a) Clone Net mutation operator.

(b) Add Connection mutation operator.



(c) Insert Network mutation operator.

Fig. 6.5: Percentage improvement observed over the regression output of the VALP in logarithmic scale (in the y axis), by different mutation operators, with respect to the relevance of the network affected by the operator, also in logarithmic scale (x axis, the larger, the more relevant a network to the output).

the latter producing more improvements when applied to relevant networks. The connection adding operator in this case is not advisable with relevant networks.

6.6.2 Operator per network characterization

With the insights made in the previous section, we have defined the following set of rules to display the potential of this kind of guided searches.

- When a network involved in a loss function registers a steep descent, the gentle version of the network cloning mutation can be applied.
- In case a network is part of an output which is moderately descending, the add connection operator can be applied.
- When a network is part of an output in a local optimum, the insert network can be applied.
- If a network is not relevant for some outputs it is connected to, but is for other ones, the delete connection, the insert network, or the aggressive version of the clone network can be applied.
- If a network is not relevant for any output, the network deletion operator can be applied.

These rules have been compiled into the mutation selection guidelines, which are going to be used in the HC algorithm.

6.6.3 Main experimentation

The threshold values for determining whether a loss function is descending or not, or how relevant a sub-network is, are parameters of the NAS algorithm. In this case, they are estimated from the initial experimentation. A loss slope larger than -10^{-10} is considered to be stuck, and if smaller than -2×10^{-5}, it is determined to be steeply descending. Anything in between these two values is considered to be moderately descending.

A network is considered to be relevant to an output if the performance of the VALP in that output decreases by 20% or less.

Although these values have been estimated using only this experimental section, as similar loss functions have similar behaviors, we consider them to be transferable to other problems.

6.6.4 Operator selection

With these defined criteria, all sub-networks within a VALP can be modified by several operators at each stage. Therefore, we define a hierarchy in which the operators are organized according to the priority they are given to modify the models.

1. Reducers: Because we pursue efficient models, any network or connection which is not valuable for the overall performance should be deleted.
2. Aggressive expanders: Any network which, according to the rules defined in the previous section, can be affected by an aggressive expander operator, is assumed not to be working properly, and this is the second priority.
3. Gentle expanders: Giving more modeling power to a model only makes sense when all its resources are being used, and therefore this is the last type of operators to be taken into account.

In the smart approach, when selecting the operator to be applied, within the set of operators with the highest priority, one is chosen at random. If a selected operator is not able to improve the current model, in the next step, it is not included among the candidate operators to generate a neighbor of the model. When no operator from those selected by the described method is able to create a candidate model which could replace the current solution (taking into account all three preference level sets), a random gentle operator is applied to a random network in the model.

Since we deal with multi-task problems where multiple objectives have to be simultaneously optimized, the question of deciding what model is the better one is not trivial, and, therefore, neither is when comparing search algorithms. We thus resort to using two Pareto front-based approaches to compare the quality of the VALP structures found.

For the first comparison, we take each of the 30 pairs of runs separately, considering as pairs those runs which start from the same random VALP structure and use

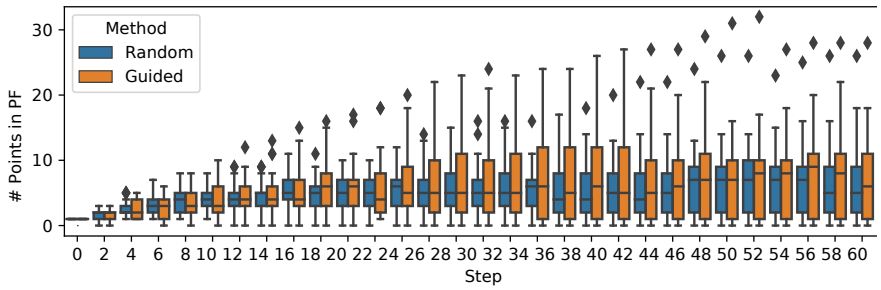


Fig. 6.6: Boxplot showing the number of points (y axis) in the PFs generated from combining the pairwise HC runs (blue for random, orange for guided), per step (x axis).

the random and guided HC approaches. At each step, all the structures found (across the whole search) by a pair of runs are compiled into a single set, and a Pareto front is computed, considering the three outputs of the model. This way, in, for example, the fourth step, we have 30 different PFs, each being composed of at most eight points, four from each of the corresponding runs (one per completed step). Next, the number of points in the PF from each HC approach are counted. In Figure 6.6, boxplots which display the number of points in the PFs (y axis) by each approach (orange for the guided HC and blue for the random version), in each step (x axis) are shown.

As can be seen in Figure 6.6, in the initial 20 steps, both versions of the algorithm work similarly, with a slight advantage for the random HC. This trend changes after the 20-th step, where, although the median remains similar, the top results are clearly produced by the guided version of the algorithm.

Interestingly, both the random and the guided versions have produced one run each which generate a number of points in their corresponding PFs far superior to the rest. These outliers are also higher in the guided version.

Secondly, we consider all 30 runs together, in order to know what algorithm is able to obtain the best results, overall. In this case, instead of constructing one PF per step and pair of runs, we simply construct a single PF from all the points found across the 30 runs limited only by the step. Again, all the found structures until a step are considered in each step. The results are shown in Figure 6.7.

Although Figures 6.6 and 6.7 look dissimilar, the information shown coincides. During the initial stages of the search (the initial 16-17 steps), the algorithms are searching for the best area to exploit, at which the random HC seems to outperform the guided version. This comes as no surprise, as the randomized approach does not focus on a *search path* to follow. Because it can perform modifications in any place within the model structure, the model can improve or lose performance continuously in different outputs. This helps a larger presence of points generated by the random HC in the PFs shown in Figure 6.7, as opposed to the guided version, which focuses on improving certain aspects of the model -the efficiency of the sub-networks- before starting to seek performance improvements. That first phase ends near the 18th step,

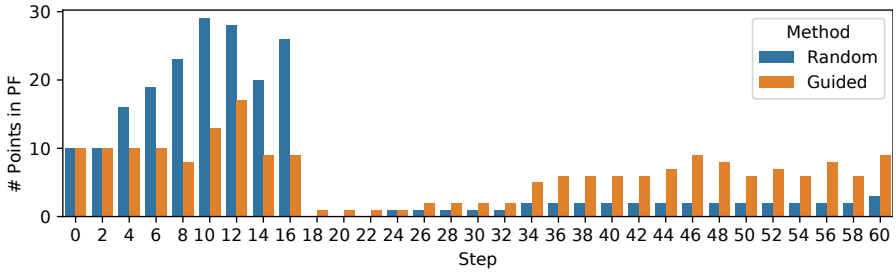


Fig. 6.7: Barplot showing the number of points (y axis) in the combined PF from each approach (blue for random, orange for guided), per step (x axis).

as one of the guided runs achieves one VALP configuration capable of dominating all the ones found during all searches. Slowly, other points start to form the PF, most of which belong to the guided runs. This shows the benefit of the guided search over the randomized one in the long term when performing intelligently chosen moves.

Figure 6.6 displays the gains obtainable from using the *smart* approach in medium term, as it shows how the guided search gains an advantage early in the search, and the randomized HC spends several steps trying to catch up, until it eventually does, after ~ 40 steps. On the contrary, Figure 6.7 shows the benefits of the proposed algorithm in the longer term, as it is able to produce more top results than the simpler HC version as steps are taken.

6.7 Conclusions

Our efforts in this chapter are focused on compiling a set of guidelines which aims at maximizing the effectiveness of the application of variation operators to model structures during a structural search procedure, illustrated using a complex scenario, that of HMTL. More specifically, we have first identified several metrics which can be used to determine the *role* that different sub-DNNs play on the overall performance. Secondly, we have compiled a set of variation operators previously used in NAS procedures described in the literature and classify them according to the effect they have on the complexity and performance of the model. Next, we have conducted an extensive exploratory search on how a subset of the operators affects the performance of a *medium-sized* model, in order to identify patterns that relate the defined metrics and the largest improvements in the models. These patterns have later been transformed into a set of guidelines for enhancing the efficiency of future NAS searches. These guidelines add one level of sophistication to current NAS algorithms, as, opposed to the common practice of randomly selecting a variation operator, a more informed choice is made, which can save the need to evaluate DNN structures affected by the *wrong* operator. Finally, a comparison of the performance of the two variants of NAS search -blind versus guided by the introduced rules- has been presented as an illustration of the gains that could be obtained in NAS efficiency.

The main contribution found in this chapter is the methodology for diagnosing the state of an VALP model and identifying the role fulfilled by its different components, and the application of these metrics for more efficient NAS algorithms. One key for this goal is the set of metrics defined with this purpose, although others which complement those introduced in this chapter could result in more valuable information about the model, ultimately making the processes more efficient.

The experiments conducted in this chapter serve as a blueprint for implementing the presented ideas to other problems and domains, as they have already served the purpose of exploring a complex search space using a rather simple algorithm more efficiently than the common, random-based approach, both in the medium and long term. The architecture searches conducted in this chapter have been limited to *simple* local searches. However, the defined methodology is not restrained to be applied in such scenarios, and its effectiveness on other search algorithm types is left as future work.

7

Conclusions and future work

This chapter summarizes the main contributions drawn from this dissertation, as well as a compilation of general directions for future research.

7.1 Contributions

In the last decade, DNNs have been applied to solve multiple and diverse problems, having to deal with increasingly complex challenges. For this purpose, NAS methods have emerged as a very suitable proposal to automatically design strong-performing DNN architectures. The ever-increasing complexity of the search spaces needed to be explored by these methods, however, is reducing the feasibility of the efficient performance of NAS algorithms. In this conjuncture, we set the focal point of this dissertation on reducing the high computational workload that NAS algorithms entail in their current definition. In order to achieve this goal, the first step is to deeply understand the inner mechanics behind these algorithms. In order to accomplish that understanding level, we have tackled the problem of NAS efficiency in two different scenarios, both in complexity level and the detail in which they have been studied by the research community: generative modeling and multi-task learning. In both cases, our approach to the problem has consisted of a first step in which an extensive study of different NAS searches has been carried out, aiming at validating and understanding different characteristics of the processes. Once the transferability of the methodology was validated, the approach is studied in detail in order to extract valuable knowledge which can benefit the efficiency future structural optimization procedures.

The main contributions of this dissertation consist of: i) We have investigated the behavior and transferability of GANs obtained by means of structural optimization using genetic algorithms. ii) We have defined a model capable of capturing and exploiting the information resulting from the investigation performed in the previous point, which can positively impact future NAS searches. iii) We have defined a novel model for heterogeneous multi-task learning, whose main goal is to be flexible enough to deal with multiple tasks and data inputs and outputs simultaneously,

while being defined by a structure which can be continuously optimized. iv) We have defined a framework capable of characterizing both models and variation operators (which are ubiquitous in many NAS algorithms) in order to discover synergies between both of them.

In the following paragraphs, we further elaborate on the four subjects mentioned.

In chapter 3, we have presented the initial foundations of this work. It consists of an exhaustive study carried out over a NE procedure used to optimize the structure of a vastly recognized model, the GAN. For conducting NAS over GANs, two main challenges have to be addressed: i) it is not a trivial task to assess how realistic the outcome of a generative model is, and ii) mode collapsing, that is, the GAN limiting all its generations to one of the modes of the original data. In the Pareto set approximation problem, we found the perfect environment to evolve GANs, as metrics such as the IGD allow the automated and objective measurement of the quality and diversity of a set of points with respect to the reference Pareto set. Thus, we set a neuroevolutionary framework for evolving GANs using the PS approximation problem aiming for efficient structures by simultaneously optimizing the capacity to sample good quality PS approximations -according to the IGD-, as well as the time spent on training and sampling the model. The first key finding of this chapter is the high transferability and robustness levels of the learned structures between problems, which suggests that the residual information generated from a set of NAS runs can positively and consistently impact future searches. This way, we designed a methodology capable of automatically evolving efficient GANs, which can be later transferred to other problems where the unbiased and automatic evaluation of structures is not trivial. With that finding in mind, the characteristics of the top structures found by the algorithms were analyzed, seeking patterns which could confirm that mature structures share multiple commonalities, as some GAN components were present substantially more often than others in these developed structures.

In Chapter 4, we have made an initial attempt to capitalize on the study performed in the previous chapter. Under the assumption that the GAN components which are more frequently found on fully evolved GANs cannot work properly if the rest of the configuration of the structure does not fit these components, we aimed at defining a model which can capture and represent these dependencies. We identified two manners in which future structural searches can benefit by such a *metamodel*: generating and discriminating GAN structures. We chose to base the metamodel on Bayesian networks, as they can fulfill both tasks successfully. From the experiments conducted, two main conclusions were drawn. First, the learned structures were transferable between problems, as seen in the previous work. Secondly, the metainformation generated by the evolutionary process itself was also transferable, as the metamodel had been able to produce efficient GAN structures for problems different to those from which it was learned, as well as distinguishing between GANs which were highly likely to perform strongly from those which were not. This second capacity enabled the metamodel to be used as a discriminator of GAN structures not worthy of being evaluated during a search, considering the low chance of them adding positive value to the search.

After the initial, in-depth study on the extensively applied framework of GANs, in Chapter 5, we have extended our scope towards a more challenging research area, that of heterogeneous multi-task learning. The complexity of having to deal with multiple tasks of potentially diverse types simultaneously has oriented research towards hand-made model definitions in which different *submodules* of a larger model play predefined roles. These models, being tailored to fit certain specifications, are forbidden from exploring performance improvements by architectural optimization. In this chapter, we have defined a new model, the VALP, capable of supporting heterogeneous DNN structures for solving tasks of the previously defined characteristics, and without a predefined structure, which enables structural optimization. A simple random search over the space of VALP structures showed significant differences between the VALPs generated during the search in all three objectives involved in the framework. These differences clearly show the importance of structural optimization of VALPs.

Taking the VALP model as a generalization of the common single-task models (e.g., generative modeling with a GAN, or classification using a CNN), in Chapter 6, we have defined a set of rules which can improve the efficiency in future VALP structural searches. Because any single-task model can be defined using a VALP, these rules can also be applied to any of these neural structural searches. To achieve this goal, we first defined a set of metrics which helps to identify the *weak* and *strong* parts of a model, thus marking some of the parts of the model as more susceptible of being optimized than others. Along with that model part identification, we also characterized a set of variation operators by how they affect a model, both in terms of the effect on the current performance, and the number of weights of the model. Pairing these metrics and the identified variation operators, we performed an exploratory search over the application of the classified variation operators on multiple VALPs characterized by the metrics, and we defined a set of rules which indicate the most adequate operator to be applied to which part of the model when optimizing the structure of that model. The proposed rules were able to guide a NAS algorithm towards better quality VALPs compared to those found by the same algorithm without the guidance of the rules.

The code developed for the work reported in Chapter 3 was later generalized to form EvoFlow [44], a library which encompasses the evolution of a far wider set of models than the initial GAN. Besides the MLP-based GANs, it supports the evolution of CNN-based models, including convolutional layers, different pooling operations, and skip connections. Moreover, different levels of customization are available for the user. These levels range from single-DNN classification or regression models to fully customized ones, in which the user can choose the number of DNNs, the interaction between them, the problem they are covering, their loss function, etc. The concept of weight inheritance is also included in the framework. Although on its conception it was limited to Tensorflow as the backend for the DNN implementation, it is currently being extended by different authors to support PyTorch and Tensorflow 2.0.

7.2 Future work

As neural models get increasingly more complex, the attention that the efficiency of NAS algorithms will receive will intensify accordingly. In the following paragraphs, we enumerate a list of topics which research could be extended towards. First, one aspect applicable to all the work developed in this dissertation is mentioned. Next, elements worth considering for each specific chapter are specified.

- One of the main goals of all the methodology developed during this dissertation was to achieve maximum transferability between the structures found employing NAS searches, and therefore, all research was conducted using generic MLP architectures. The framework developed in this dissertation could be extended to include other, more purpose-specific DNN architecture types, such as CNNs.

The methodology proposed in Chapter 3 can be extended and applied to other different fields:

- This kind of analysis could be adapted and applied to the area of DNN explainability. We have found that it is unnecessary to non-linearly activate each and every layer of a DNN. Similar insights in this direction can be found with this kind of analysis.
- After the success of the development of the efficient GANs through the secondary objective of elapsed time reduction, the approach could be applied to evolve DNNs intended to operate in restricted environments in terms of hardware, such as cell phones or small single-board computers.
- Because the evolved GANs are fast to train and effective when sampling solutions for optimization problems, using the found GANs as solution generators for other evolutionary algorithms, e.g., estimation of distribution algorithms, could also lead to interesting research paths.

The work developed in Chapter 4, because it belongs in a relatively unexplored field, offers a wide variety of potential research lines.

- The barely restricted scheme in which DNN architectures are defined (e.g., unlimited number of layers, neurons, filters, skip connections) makes their representation as a simple set of hyperparameters a complex task, thus describing a DNN using a list of variables is nearly unfeasible. Vanilla Bayesian networks are not prepared for capturing information from examples consisting of different numbers of variables and value ranges, for which reason the metamodel had to be designed in two parts. This could limit the ability of a metamodel to correctly capture the dependencies. Employing other model families could lead to better results.
- In the same manner as with the previous chapter, the learned metamodel (in the GAN framework or in any other), because it is based on Bayesian networks, can be examined in search of interesting patterns between the key components of a GAN, aiming at the DNN explainability goal.

Because of the recentness of the VALP, and for that matter, of the heterogeneous multi-task learning problem itself, various options for future work can be proposed along with those already proposed as open challenges in Chapter 5.

- The application of the VALP to a real-world problem which could benefit from the structure of this model is an interesting future research line. Problems with many variables and tasks to be performed simultaneously, such as self driving cars. These problems receive information from cameras, GPS location, proximity sensors, etc., and have to make several decisions at each moment, such as the speed at which the car should advance, the direction in which the steering wheel should rotate, etc.
- One shortcoming of the VALP instantiation was the restriction of recursive connections. Allowing this kind of connections would help assess the potential of the model in other areas.
- The possibility of presenting the data input in different partitions has not been explored. This possibility could open the door to different sub-networks within the VALP specialized in dealing with certain parts of the data, which could be a better approach than presenting all the information to all the components connected to the data input. It would also result in the model graphically representing the dependencies between data variables and the tasks performed by the model.
- Considering sequential introduction of outputs in a VALP has also been left unexplored. While training random, fully formed VALPs, it was observed that optimizing weights considering multiple objectives at a time could lead to conflicts. By sequentially introducing them into the model, the training phase could intensify its focus in different parts of the model at each time (without losing its simultaneity characteristic).

The model and operator characterization proposal for NAS efficiency is also a new approach, which is why several directions can be followed when researching deeper into this topic. Some of which consist of:

- The deduced rules are static, in the sense that they are learned before the architecture optimization process is begun, and are not altered throughout the whole procedure. It is possible that the performance of the rules as a guide for searches decreases as the search intensifies in certain areas of the structural space. In this case, an on-line approach in which the rules can be updated as the search progresses can result in significant performance improvements.
- The addition of more characterization criteria to the proposed framework would probably result in larger efficiency gains for NAS algorithms.
- The adequacy of the methodology has not been tested in other search types, such as evolutionary algorithms. This could add further insights into the utility of the proposal.

8

Publications

8.1 Main research line

The research work carried out during this thesis has produced the following publications and submissions. Each of these publications corresponds with a chapter (3-6) in this document.

- **U. Garciarena**, A. Mendiburu, R. Santana (2020) Analysis of the transferability and robustness of GANs evolved for Pareto set approximations. *Neural Networks*, Volume 132. Pp. 281-296.
- **U. Garciarena**, R. Santana, A. Mendiburu (2021) On the Exploitation of Neuroevolutionary Information. *In Proceedings of 2021 Genetic and Evolutionary Computation Conference (GECCO-2021), Lille, France, 10-14, July 2021. Pp. 1-2. Accepted for publication.*
- **U. Garciarena**, A. Mendiburu, R. Santana (2021) Towards automatic construction of multi-network models for heterogeneous multi-task learning. *ACM Transactions on Knowledge Discovery from Data*, Volume 15-2. Article No. 33.
- **U. Garciarena**, R.Santana, A. Mendiburu Redefining neural architecture search by characterizing models and variation operators. *IEEE Transactions on Neural Networks and Learning Systems.* Submitted.

The first publication mentioned in this subsection is an extended version of the following contribution:

- **U. Garciarena**, R. Santana, A. Mendiburu (2018) Evolved GANs for generating Pareto set approximations. *In Proceedings of 2018 Genetic and Evolutionary Computation Conference (GECCO-2018), Kyoto, Japan, 15-19, July 2018. Pp. 434-441.* Nominated for the best paper award in the Evolutionary Machine Learning track.

8.2 Other developed work

Throughout these years, other research areas have been also explored, focusing on the synergy between evolutionary algorithms and deep learning. These works have not been included in this dissertation, but the contributions are listed below:

- **U. Garciarena**, R. Santana, A. Mendiburu (2018) Expanding variational autoencoders for learning and exploiting latent representations in search distributions. *In Proceedings of 2018 Genetic and Evolutionary Computation Conference (GECCO-2018), Kyoto, Japan, 15-19, July 2018. Pp. 849-856.*
- **U. Garciarena**, R. Santana, A. Mendiburu (2018) Analysis of the Complexity of the Automatic Pipeline Generation Problem. *IEEE Congress on Evolutionary Computation (CEC-2018), Rio de Janeiro, Brazil, 20-23 June 2013. Pp. 1841-1841.*
- **U. Garciarena**, A. Mendiburu, R. Santana (2020) Automatic Structural Search for Multi-task Learning VALPs. *In: Dorronsoro B., Ruiz P., de la Torre J., Urda D., Talbi EG. (eds) Optimization and Learning. OLA 2020. Communications in Computer and Information Science, vol 1173. Springer, Cham.*
- **U. Garciarena**, A. Mendiburu, R. Santana (2020) Envisioning the Benefits of Back-Drive in Evolutionary Algorithms. *IEEE Congress on Evolutionary Computation (CEC-2020), Glasgow, United Kingdom, 19-24 July 2020. Pp. 1-8.*
- **U. Garciarena**, R. Santana, A. Mendiburu (2020) EvoFlow: A Python library for evolving deep neural network architectures in tensorflow. *2020 IEEE Symposium Series on Computational Intelligence, Camberra, Australia, 1-4 December 2020. Pp. 2288-2295*
- I. Esnaola, **U. Garciarena**, J. Bermúdez (2021) Semantic Technologies towards Missing Values Imputation. *33rd International Conference on Industrial, Engineering & Other Applications of Applied Intelligent Systems (Lecture Notes in Computer Science), Accepted for publication*
- **U. Garciarena**, A. Mendiburu, R.Santana On the role of the gradient optimizer on evolved GANs. *Congreso Español de Metaheurísticas, Algoritmos Evolutivos y Bioinspirados.* Accepted for publication.

References

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., and others (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*.

[2] Al-Dujaili, A., Schmiedlechner, T., Hemberg, E., and O'Reilly, U.-M. (2018). Towards distributed coevolutionary GANs. In *AAAI 2018 Fall Symposium*.

[3] Alain, G. and Bengio, Y. (2016). Understanding intermediate layers using linear classifier probes. *arXiv preprint arXiv:1610.01644*.

[4] Ali-Gombe, A., Elyan, E., Savoye, Y., and Jayne, C. (2018). Few-shot Classifier GAN. In *2018 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8.

[5] Antoniou, A., Storkey, A., and Edwards, H. (2018). Augmenting image classifiers using data augmentation generative adversarial networks. In *International Conference on Artificial Neural Networks*, pages 594–603. Springer.

[6] Arjovsky, M. and Bottou, L. (2017). Towards Principled Methods for Training Generative Adversarial Networks. *arXiv:1701.04862*.

[7] Arjovsky, M., Chintala, S., and Bottou, L. (2017). Wasserstein Generative Adversarial Networks. In Precup, D. and Teh, Y. W., editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 214–223, International Convention Centre, Sydney, Australia. PMLR.

[8] Arras, L., Montavon, G., Müller, K.-R., and Samek, W. (2017). Explaining recurrent neural network predictions in sentiment analysis. *arXiv preprint arXiv:1706.07206*.

[9] Assunçao, F., Lourenço, N., Machado, P., and Ribeiro, B. (2019). DENSER: Deep Evolutionary Network Structured Representation. *Genetic Programming and Evolvable Machines*, 20(1):5–35. Publisher: Springer.

[10] Bender, G., Kindermans, P.-J., Zoph, B., Vasudevan, V., and Le, Q. (2018). Understanding and Simplifying One-Shot Architecture Search. In Dy, J. and Krause, A., editors, *Proceedings of the 35th International Conference on Machine Learn-*

ing, volume 80 of *Proceedings of Machine Learning Research*, pages 550–559. PMLR.

[11] Brock, A., Lim, T., Ritchie, J. M., and Weston, N. (2018). SMASH: One-Shot Model Architecture Search through HyperNetworks. In *International Conference on Learning Representations*.

[12] Burgess, C. P., Higgins, I., Pal, A., Matthey, L., Watters, N., Desjardins, G., and Lerchner, A. (2018). Understanding disentangling in -VAE. *arXiv preprint arXiv:1804.03599*.

[13] Caruana, R. (1997). Multitask learning. *Machine Learning*, 28(1):41–75.

[14] Castillo, E., Gutierrez, J. M., and Hadi, A. S. (1997). *Expert Systems and Probabilistic Network Models*. Springer.

[15] Che, T., Li, Y., Jacob, A. P., Bengio, Y., and Li, W. (2016). Mode regularized generative adversarial networks. *arXiv preprint arXiv:1612.02136*.

[16] Chen, T., Goodfellow, I., and Shlens, J. (2015). Net2net: Accelerating learning via knowledge transfer. *arXiv preprint arXiv:1511.05641*.

[17] Chen, Y., Gao, R., Liu, F., and Zhao, D. (2020). ModuleNet: Knowledge-inherited Neural Architecture Search. *arXiv preprint arXiv:2004.05020*.

[18] Cho, H.-Y. and Kim, Y.-H. (2019). Stabilized Training of Generative Adversarial Networks by a Genetic Algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 51–52, New York, NY, USA. Association for Computing Machinery.

[19] Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.

[20] Chow, C. K. and Liu, C. N. (1968). Approximating discrete probability distributions with dependence trees. *IEEE Transactions on Information Theory*, 14(3):462–467.

[21] Ciregan, D., Meier, U., and Schmidhuber, J. (2012). Multi-column deep neural networks for image classification. In *2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3642–3649. IEEE.

[22] Coello, C., Lamont, G., and Van Veldhuizen, D. (2007). *Evolutionary Algorithms for Solving Multi-objective Problems*. Springer-Verlag New York Inc.

[23] Coello Coello, C. A. and Reyes Sierra, M. (2004). A Study of the Parallelization of a Coevolutionary Multi-objective Evolutionary Algorithm. In Monroy, R., Arroyo-Figueroa, G., Sucar, L. E., and Sossa, H., editors, *MICAI 2004: Advances in Artificial Intelligence*, pages 688–697, Berlin, Heidelberg. Springer Berlin Heidelberg.

[24] Costa, V., Lourenço, N., and Machado, P. (2019). Coevolution of Generative Adversarial Networks. In *International Conference on the Applications of Evolutionary Computation (Part of EvoStar)*, pages 473–487. Springer.

[25] Cui, Y., Song, Y., Sun, C., Howard, A., and Belongie, S. (2018). Large scale fine-grained categorization and domain-specific transfer learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4109–4118.

[26] Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314.

[27] Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE transactions on evolutionary computation*, 6(2):182–197.

[28] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255. IEEE.

[29] Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(7).

[30] Dunn, O. J. (1964). Multiple comparisons using rank sums. *Technometrics*, 6(3):241–252.

[31] Durugkar, I., Gemp, I., and Mahadevan, S. (2016). Generative multi-adversarial networks. *arXiv preprint arXiv:1611.01673*.

[32] Eiben, E. A. and Smith, J. E. (2003). *Introduction to Evolutionary Computing (Natural Computing Series)*. Springer.

[33] Elsken, T., Metzen, J.-H., and Hutter, F. (2017). Simple and efficient architecture search for convolutional neural networks. *arXiv preprint arXiv:1711.04528*.

[34] Elsken, T., Metzen, J.-H., and Hutter, F. (2019). Neural Architecture Search: A Survey. *Journal of Machine Learning Research*, 20:1–21.

[35] Fernando, C., Banarse, D., Blundell, C., Zwols, Y., Ha, D., Rusu, A. A., Pritzel, A., and Wierstra, D. (2017). Pathnet: Evolution channels gradient descent in super neural networks. *arXiv preprint arXiv:1701.08734*.

[36] Fernando, C., Banarse, D., Reynolds, M., Besse, F., Pfau, D., Jaderberg, M., Lanctot, M., and Wierstra, D. (2016). Convolution by evolution: Differentiable pattern producing networks. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pages 109–116. ACM.

[37] Fortin, F.-A., Rainville, F.-M. D., Gardner, M.-A., Parizeau, M., and Gagné, C. (2012). DEAP: Evolutionary Algorithms Made Easy. *Journal of Machine Learning Research*, 13(70):2171–2175.

[38] François-Lavet, V., Henderson, P., Islam, R., Bellemare, M. G., Pineau, J., and others (2018). An Introduction to Deep Reinforcement Learning. *Foundations and Trends in Machine Learning*, 11(3-4):219–354. Publisher: Now Publishers, Inc.

[39] Galván, E. and Mooney, P. (2020). Neuroevolution in Deep Neural Networks: Current Trends and Future Challenges. *arXiv preprint arXiv:2006.05415*.

[40] Garciarena, U., Mendiburu, A., and Santana, R. (2020a). Analysis of the transferability and robustness of GANs evolved for Pareto set approximations. *Neural Networks*, 132:281–296. Publisher: Elsevier.

[41] Garciarena, U., Santana, R., and Mendiburu, A. (2018a). Analysis of the Complexity of the Automatic Pipeline Generation Problem. In *2018 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8.

[42] Garciarena, U., Santana, R., and Mendiburu, A. (2018b). Evolved GANs for generating Pareto set approximations. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 434–441. ACM.

[43] Garciarena, U., Santana, R., and Mendiburu, A. (2018c). Expanding variational autoencoders for learning and exploiting latent representations in search distributions. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 849–856, Kyoto, Japan. ACM.

[44] Garciarena, U., Santana, R., and Mendiburu, A. (2020b). EvoFlow: A Python library for evolving deep neural network architectures in tensorflow. In *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 2288–2295.

[45] Gers, F. A., Schmidhuber, J., and Cummins, F. (1999). Learning to forget: Continual prediction with LSTM. *Technical report*.

[46] Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In Teh, Y. W. and Titterington, M., editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy. PMLR.

[47] Gong, X., Chang, S., Jiang, Y., and Wang, Z. (2019). AutoGAN: Neural Architecture Search for Generative Adversarial Networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 3224–3234.

[48] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press.

[49] Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative Adversarial Nets. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, pages 2672–2680, Cambridge, MA, USA. MIT Press. event-place: Montreal, Canada.

[50] Gretton, A., Borgwardt, K. M., Rasch, M. J., Schölkopf, B., and Smola, A. (2012). A Kernel Two-Sample Test. *Journal of Machine Learning Research*, 13(25):723–773.

[51] Harvey, I. (2009). The microbial genetic algorithm. In *European Conference on Artificial Life*, pages 126–133. Springer.

[52] He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778.

[53] Henrion, M. (1988). Propagating uncertainty in Bayesian networks by probabilistic logic sampling. In Lemmer, J. F. and Kanal, L. N., editors, *Proceedings of the Second Annual Conference on Uncertainty in Artificial Intelligence*, pages 149–164. Elsevier.

[54] Heusel, M., Ramsauer, H., Unterthiner, T., Nessler, B., and Hochreiter, S. (2017). GANs trained by a two time-scale update rule converge to a local Nash equilibrium. In *Advances in Neural Information Processing Systems*, pages 6626–6637.

[55] Hinton, G. E. and Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507.

[56] Hoang, Q., Nguyen, T. D., Le, T., and Phung, D. (2018). MGAN: Training Generative Adversarial Nets with Multiple Generators. In *International Conference on Learning Representations*.

[57] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

[58] Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366. Publisher: Elsevier.

[59] Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. (2017). Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*.

[60] Im, D. J., Ma, H., Kim, C. D., and Taylor, G. (2016). Generative adversarial parallelization. *arXiv preprint arXiv:1612.04021*.

[61] Isola, P., Zhu, J.-Y., Zhou, T., and Efros, A. A. (2017). Image-to-image translation with conditional adversarial networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1125–1134.

[62] Jarrett, K., Kavukcuoglu, K., LeCun, Y., and others (2009). What is the best multi-stage architecture for object recognition? In *2009 IEEE 12th International Conference on Computer Vision (ICCV)*, pages 2146–2153. IEEE.

[63] Jianlin, S. (2017). A baseline of Fashion MNIST (MobileNet 95%). Blog entry: https://kexue.fm/archives/4556.

[64] Karras, T., Aila, T., Laine, S., and Lehtinen, J. (2018). Progressive Growing of GANs for Improved Quality, Stability, and Variation. In *International Conference on Learning Representations*.

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

[66] Kingma, D. P. and Welling, M. (2013). Auto-encoding variational Bayes. *arXiv preprint arXiv:1312.6114*.

[67] Kornblith, S., Shlens, J., and Le, Q. V. (2019). Do better imagenet models transfer better? In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2661–2671.

[68] Krizhevsky, A. (2009). Learning multiple layers of features from tiny images. Master's thesis, Department of Computer Science, University of Toronto.

[69] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105.

[70] Kruskal, W. H. and Wallis, W. A. (1952). Use of ranks in one-criterion variance analysis. *Journal of the American Statistical Association*, 47(260):583–621.

[71] Kullback, S. and Leibler, R. A. (1951). On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86.

[72] Larrañaga, P. and Lozano, J. A. (2001). *Estimation of distribution algorithms: A new tool for evolutionary computation*, volume 2. Springer Science & Business Media.

[73] Lauritzen, S. L. (1996). *Graphical Models*. Oxford Clarendon Press.

[74] LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521(7553):436–444.

[75] LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551.

[76] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

[77] LeCun, Y., Cortes, C., and Burges, C. (2010). MNIST handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2.

[78] Li, H. and Zhang, Q. (2008). Multiobjective Optimization Problems with Complicated Pareto Sets, MOEA/D and NSGA-II. *IEEE Transactions on Evolutionary Computation*, 13(2):284–302.

[79] Li, W., Zhu, X., and Gong, S. (2017). Person re-identification by deep joint learning of multi-loss classification. *arXiv preprint arXiv:1705.04724*.

[80] Liang, J., Meyerson, E., and Miikkulainen, R. (2018). Evolutionary Architecture Search for Deep Multitask Networks. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 466–473. ACM.

[81] Lima, R. H., Pozo, A., Mendiburu, A., and Santana, R. (2020). A Symmetric grammar approach for designing segmentation models. In *2020 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE.

[82] Linden, A. and Kindermann, J. (1989). Inversion of multilayer nets. In *Proc. Int. Joint Conf. Neural Networks*, volume 2, pages 425–430.

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

[84] Liu, T., Tao, D., Song, M., and Maybank, S. J. (2016). Algorithm-dependent generalization bounds for multi-task learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(2):227–241.

[85] Loshchilov, I. and Hutter, F. (2016). CMA-ES for hyperparameter optimization of deep neural networks. *arXiv preprint arXiv:1604.07269*.

[86] Lu, J., Ma, W., and Faltings, B. (2018a). CompNet: Neural networks growing via the compact network morphism. *arXiv preprint arXiv:1804.10316*.

[87] Lu, Z., Whalen, I., Boddeti, V., Dhebar, Y., Deb, K., Goodman, E., and Banzhaf, W. (2018b). NSGA-NET: a multi-objective genetic algorithm for neural architecture search. *arXiv preprint arXiv:1810.03522*.

[88] Marcus, M., Santorini, B., and Marcinkiewicz, M. A. (1993). Building a Large Annotated Corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330.

[89] Margolin, A. A., Nemenman, I., Basso, K., Wiggins, C., Stolovitzky, G., Dalla Favera, R., and Califano, A. (2006). ARACNE: an algorithm for the reconstruction of gene regulatory networks in a mammalian cellular context. In *BMC bioinformatics*, volume 7, page S7. Springer.

[90] Marti, L., Garcia, J., Berlanga, A., and Molina, J. M. (2009). An approach to stopping criteria for multi-objective optimization evolutionary algorithms: The

MGBM criterion. In *2009 IEEE Congress on Evolutionary Computation*, pages 1263–1270.

[91] Maul, T., Bargiela, A., Chong, S. Y., and Adamu, A. S. (2014). Towards Evolutionary Deep Neural Networks. In *European Conference on Modelling and Simulation*, pages 319–325.

[92] Mehta, K., Kobti, Z., Pfaff, K., and Fox, S. (2019). Data Augmentation using CA Evolved GANs. In *2019 IEEE Symposium on Computers and Communications*, pages 1087–1092.

[93] Metz, L., Poole, B., Pfau, D., and Sohl-Dickstein, J. (2016). Unrolled generative adversarial networks. *arXiv preprint arXiv:1611.02163*.

[94] Meyerson, E. and Miikkulainen, R. (2017). Beyond Shared Hierarchies: Deep Multitask Learning through Soft Layer Ordering. *arXiv preprint arXiv:1711.00108*.

[95] Miikkulainen, R., Liang, J., Meyerson, E., Rawal, A., Fink, D., Francon, O., Raju, B., Shahrzad, H., Navruzyan, A., Duffy, N., and others (2019). Evolving deep neural networks. In *Artificial Intelligence in the Age of Neural Networks and Brain Computing*, pages 293–312. Elsevier.

[96] Miller, J. F. and Thomson, P. (2000). Cartesian genetic programming. In *European Conference on Genetic Programming*, pages 121–132. Springer.

[97] Mitchell, M. (1998). *An Introduction to Genetic Algorithms*. MIT Press.

[98] Mitchell, M. (2006). Coevolutionary learning with spatially distributed populations. In Fogel, D. B., editor, *Computational intelligence: principles and practice*. IEEE Computational Intelligence Society.

[99] Morcos, A. S., Yu, H., Paganini, M., and Tian, Y. (2019). One ticket to win them all: generalizing lottery ticket initializations across datasets and optimizers. *arXiv preprint arXiv:1906.02773*.

[100] Morimoto, T. (1963). Markov processes and the H-theorem. *Journal of the Physical Society of Japan*, 18(3):328–331.

[101] Netzer, Y., Wang, T., Coates, A., Bissacco, A., Wu, B., and Ng, A. Y. (2011). Reading digits in natural images with unsupervised feature learning. In *NIPS workshop on deep learning and unsupervised feature learning*.

[102] Neyshabur, B., Bhojanapalli, S., and Chakrabarti, A. (2017). Stabilizing GAN training with multiple random projections. *arXiv preprint arXiv:1705.07831*.

[103] Nowozin, S., Cseke, B., and Tomioka, R. (2016). f-GAN: Training generative neural samplers using variational divergence minimization. In *Advances in Neural Information Processing Systems*, pages 271–279.

[104] Olson, R. S., Bartley, N., Urbanowicz, R. J., and Moore, J. H. (2016). Evaluation of a tree-based pipeline optimization tool for automating data science. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference*, pages 485–492. ACM.

[105] Papers With Code (2021a). *State-of-the-Art Image Classification on the Imagenet Benchmark*. Publication Title: https://paperswithcode.com.

[106] Papers With Code (2021b). *State-of-the-Art Language Modelling on the Penn Treebank Benchmark*. Publication Title: https://paperswithcode.com.

[107] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.

[108] Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, San Mateo, California.

[109] Pearl, J. (2000). *Causality: Models, Reasoning and Inference*. Cambridge University Press.

[110] Radford, A., Metz, L., and Chintala, S. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*.

[111] Real, E., Aggarwal, A., Huang, Y., and Le, Q. V. (2019). Regularized evolution for image classifier architecture search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 4780–4789.

[112] Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386.

[113] Salimans, T., Goodfellow, I., Zaremba, W., Cheung, V., Radford, A., and Chen, X. (2016). Improved techniques for training GANs. In *Advances in Neural Information Processing Systems*, pages 2234–2242.

[114] Schutze, O., Esquivel, X., Lara, A., and Coello, C. A. C. (2012). Using the averaged Hausdorff distance as a performance measure in evolutionary multiobjective optimization. *IEEE Transactions on Evolutionary Computation*, 16(4):504–522.

[115] Scutari, M. and Denis, J.-B. (2014). *Bayesian Networks: With Examples in R*. CRC press.

[116] Shwartz-Ziv, R. and Tishby, N. (2017). Opening the black box of deep neural networks via information. *arXiv preprint arXiv:1703.00810*.

[117] Smolensky, P. (1986). Information processing in dynamical systems: Foundations of harmony theory. Technical Report CU-CS-321-86, Colorado University at Boulder. Dept. of Computer Science.

[118] Srivastava, A., Valkov, L., Russell, C., Gutmann, M. U., and Sutton, C. (2017). Veegan: Reducing mode collapse in GANs using implicit variational learning. In *Advances in Neural Information Processing Systems*, pages 3308–3318.

[119] Stanley, K. O. (2007). Compositional pattern producing networks: A novel abstraction of development. *Genetic Programming and Evolvable Machines*, 8(2):131–162.

[120] Stanley, K. O. and Miikkulainen, R. (2002a). Efficient reinforcement learning through evolving neural network topologies. In *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, pages 569–577. Morgan Kaufmann Publishers Inc.

[121] Stanley, K. O. and Miikkulainen, R. (2002b). Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127.

[122] Suganuma, M., Shirakawa, S., and Nagao, T. (2017). A genetic programming approach to designing convolutional neural network architectures. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 497–504. ACM.

[123] Sun, Y., Cuesta-Infante, A., and Veeramachaneni, K. (2019). Learning vine copula models for synthetic data generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 5049–5057.

[124] Szegedy, C., Ioffe, S., Vanhoucke, V., and Alemi, A. A. (2017). Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI'17, pages 4278–4284. AAAI Press. event-place: San Francisco, California, USA.

[125] Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., and Fergus, R. (2013). Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*.

[126] Thornton, C., Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2013). Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 847–855, New York, NY, USA. ACM.

[127] Tolstikhin, I. O., Gelly, S., Bousquet, O., Simon-Gabriel, C.-J., and Schölkopf, B. (2017). Adagan: Boosting generative models. In *Advances in Neural Information Processing Systems*, pages 5424–5433.

[128] Toutouh, J., Hemberg, E., and O'Reilly, U.-M. (2019). Spatial Evolutionary Generative Adversarial Networks. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 472–480, New York, NY, USA. Association for Computing Machinery. event-place: Prague, Czech Republic.

[129] Uriot, T. and Izzo, D. (2020). Safe crossover of neural networks through neuron alignment. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, pages 435–443.

[130] Wang, C., Xu, C., Yao, X., and Tao, D. (2019). Evolutionary Generative Adversarial Networks. *IEEE Transactions on Evolutionary Computation*, 23(6):921–934.

[131] Wang, K., Gou, C., Duan, Y., Lin, Y., Zheng, X., and Wang, F.-Y. (2017). Generative adversarial networks: introduction and outlook. *IEEE/CAA Journal of Automatica Sinica*, 4(4):588–598.

[132] Wang, R., Cheng, M., Chen, X., Tang, X., and Hsieh, C.-J. (2021). Rethinking Architecture Selection in Differentiable NAS. In *International Conference on Learning Representations*.

[133] Wang, Y., Zhang, L., and van de Weijer, J. (2016). Ensembles of generative adversarial networks. *arXiv preprint arXiv:1612.00991*.

[134] Wei, T., Wang, C., Rui, Y., and Chen, C. W. (2016). Network morphism. In *International Conference on Machine Learning*, pages 564–572.

[135] Werbos, P. J. (1994). *The roots of backpropagation: from ordered derivatives to neural networks and political forecasting*, volume 1. John Wiley & Sons.

[136] White, C., Nolen, S., and Savani, Y. (2020). Local Search is State of the Art for NAS Benchmarks. *arXiv preprint arXiv:2005.02960*.

[137] Wilcoxon, F. (1945). Individual comparisons by ranking methods. *Biometrics bulletin*, 1(6):80–83.

[138] Xiao, H., Rasul, K., and Vollgraf, R. (2017). Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. arXiv: cs.LG/1708.07747.

[139] Yan, X., Yang, J., Sohn, K., and Lee, H. (2016). Attribute2image: Conditional image generation from visual attributes. In *European Conference on Computer Vision*, pages 776–791. Springer.

[140] Zhang, X., Zhang, X., and Liu, H. (2015). Smart multitask Bregman clustering and multitask kernel clustering. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 10(1):8.

[141] Zhang, Y. and Yang, Q. (2017). A survey on multi-task learning. *arXiv preprint arXiv:1707.08114*.

[142] Zhang, Y. and Yeung, D.-Y. (2014). A regularization approach to learning task relationships in multitask learning. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 8(3):12.

[143] Zheng, Z., Yang, X., Yu, Z., Zheng, L., Yang, Y., and Kautz, J. (2019). Joint discriminative and generative learning for person re-identification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2138–2147.

[144] Zitzler, E. and Thiele, L. (1999). Multiobjective evolutionary algorithms: A comparative case study and the strength pareto approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257–271.

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

[146] Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. (2018). Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8697–8710.

Appendices

A

Annex for Chapter 3

As a complement to Figure 3.8, we present Figure A.1; consisting of scatter plots showing information related to the remaining 6 functions: $F1$, $F2$, $F3$, $F4$, $F5$, and $F9$.

In these figures it can observed how evolution has been able to produce much more robust GANs. This is exceptionally clear in $F1$, $F3$, $F4$, where only the evolved GANs are able to reach the lower values in all three metrics. For $F2$, $F5$, and $F9$, some random GANs have been able to sparsely produce low results too. The evolved models, however, are much more consistent, as they hardly ever produce high values in any metric, unlike random GANs.

Figure A.2 shows the results regarding transferability of GANs evolved with $n = 784$ to other functions with the same problem dimension, as a complement to Figure 3.9. This way, the relative performance of the dimension transferability problem can be compared. We observe how, in some cases, the GANs evolved with $n = 10$ produce better outcomes when transferred to $n = 784$ and other functions, compared to those directly evolved with $n = 784$: $F7$, and $F8$. In the rest of the cases, the GANs evolved with $n = 784$ performed better, which could have been expected. Finally, $F4$ again stands out as the best function for evolving GANs, as these models outperform other 4 sets of GANs when sampling for a function they were evolved with. Additionally, GANs evolved for $F4$ were never outperformed when generating points for $F4$.

Figure A.3 is provided to complement the information shown in Figure 3.11. The obtainable conclusions do not differ from those extracted from Figure 3.11, as a similar overlap can be visualized, yet the evolved GANs produce considerably lower values in both mean MMD and variance. It is remarkable, however, the isolated point with low values in both mean and variance, a top consistency not found with $n = 10$.

Figs. A.4-A.9 are presented as complementary to Figure 3.12. This set of barplots show the same data as Figure 3.12; the appearance frequency of different network components for all functions.

Observing Figure A.4 (which shows the appearance frequency of the different initialization functions), we can notice that for $n = 10$, weights are hardly ever initialized with a normal distribution. In this aspect, the presence of Xavier and uniform

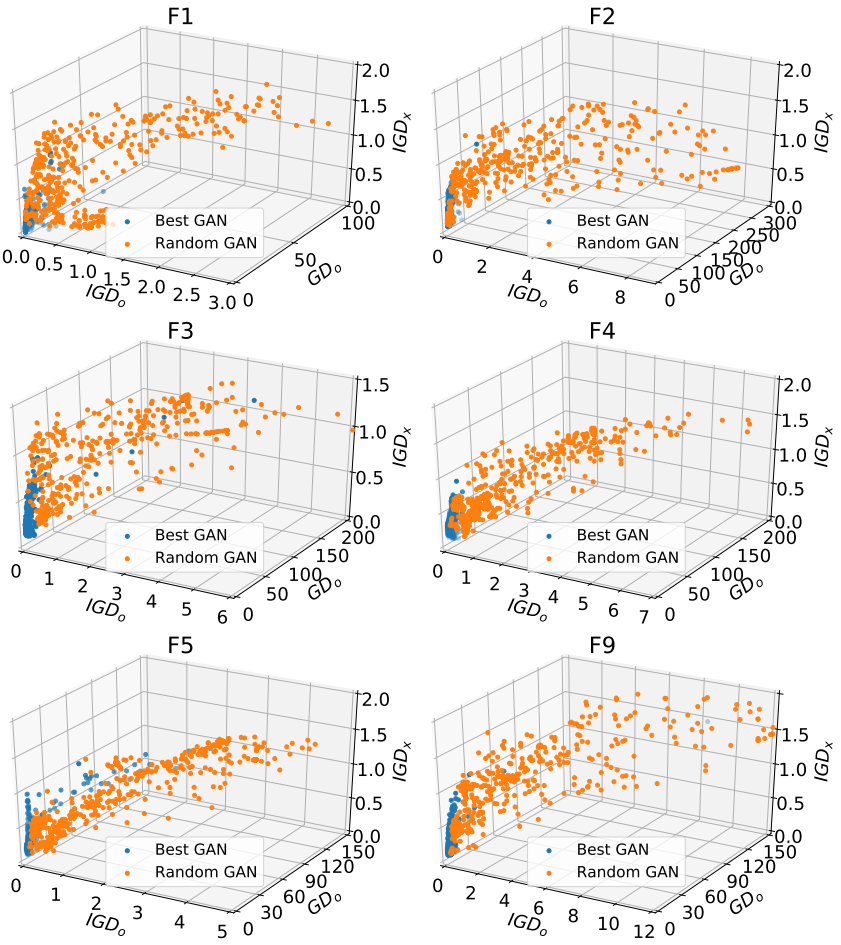


Fig. A.1: Relation of IGD_o, IGD_x, and GD_o for functions $F1$, $F2$, $F3$, $F4$, $F5$, and $F9$.

initialization is much more prominent. For $n = 784$, Xavier initialization is the more prominent one in most cases.

Figure A.5 (showing data related to activation functions), confirms that the NE algorithm chose to keep most layers in all networks linearly activated. Beyond that, we again perceive much more pronounced differences with $n = 10$ as opposed to $n = 784$. For $n = 10$, Softsign, and Tanh (and Elu, to a lesser extent) are the most present functions. For $n = 784$, the pattern is not that clear, but suggests the usage of the same activation functions.

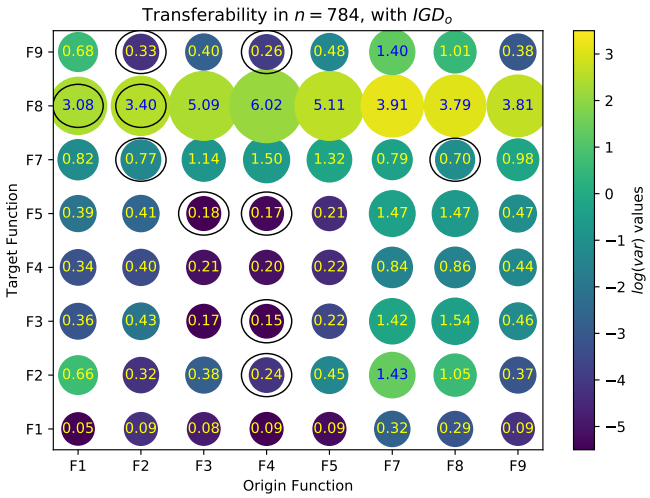


Fig. A.2: Transferability of the best GANs evolved with $n = 784$ across different functions.

In terms of network depth, very reduced networks are used in all cases, judging by Figure A.6, which shows the number of layers in the different networks. This can be seen for all function and problem sizes.

Figure A.7 (showing the appearance frequency of prior distributions and loss functions) shows that the random uniform is the preferred distribution as the prior in all cases (upper part of the figure). Regarding the loss functions (lower part of the figure), it is ratified that the KL and Wasserstein loss functions are the preferred choices, especially for $n = 10$. For $n = 784$, the Pearson's χ^2 loss function is also commonly found, but the Wasserstein function can again be found in most functions.

Regarding the layer size distribution displayed in Figure A.8, we can observe how generators tend to be narrower than discriminators, except for $F8$. Again, the effect of the complexity-penalizing objective becomes apparent in the evolved network structures. This effect is also visible in Figure A.9, where the loop parameter for generators and discriminators is shown. Apparently, in most cases, updating the networks once with each batch is enough in most cases.

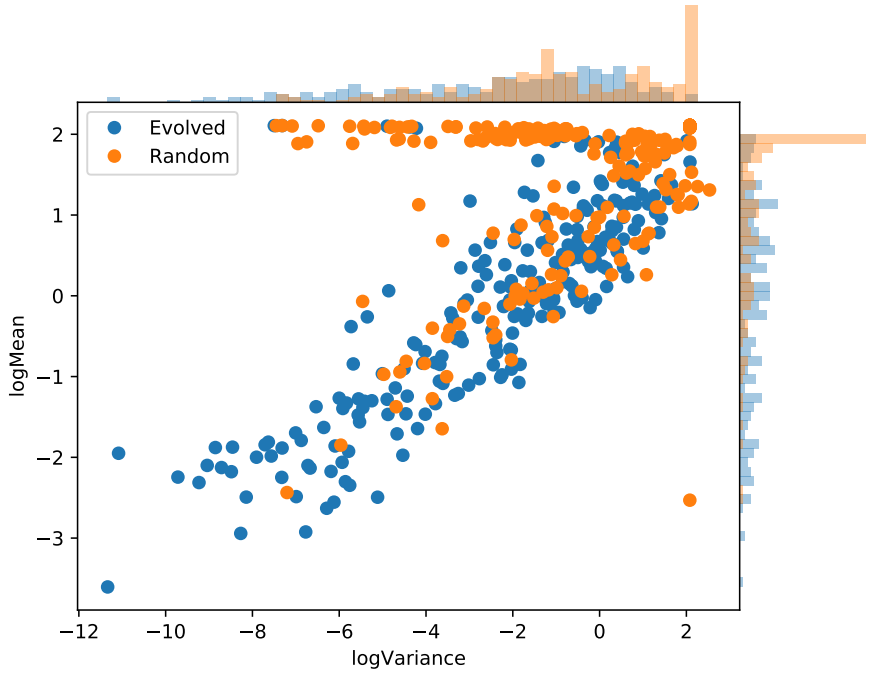


Fig. A.3: Transferability of GANs evolved with functions with $n = 784$.

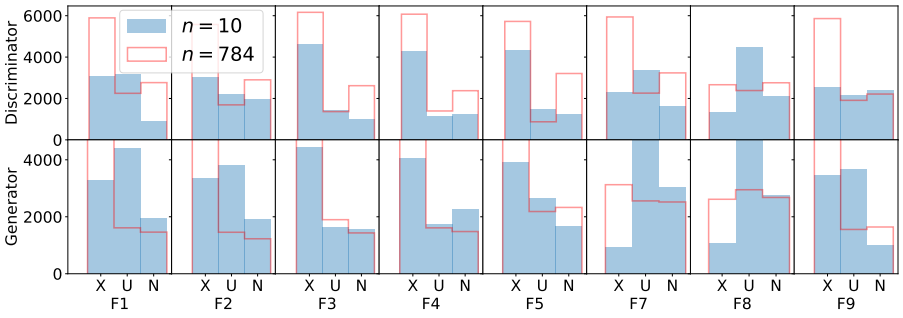


Fig. A.4: Initialization functions present in the two networks of fully evolved GANs, for all functions with both problem dimensions. X: Xavier initialization, U: random Uniform, N: random Normal.

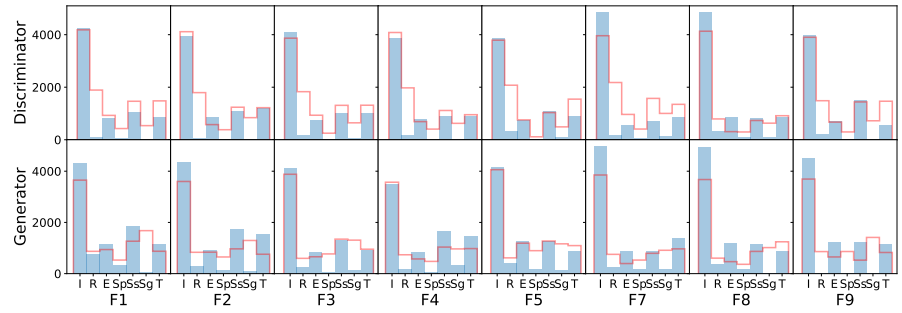


Fig. A.5: Activation functions present in the two networks of fully evolved GANs, for all functions with both problem dimensions. I: Identity function, R: Rectified linear unit, E: Exponential Linear Unit, Sp: Softplus function, Ss: Softsign function, Sg: Sigmoid function, T: Tanh function.

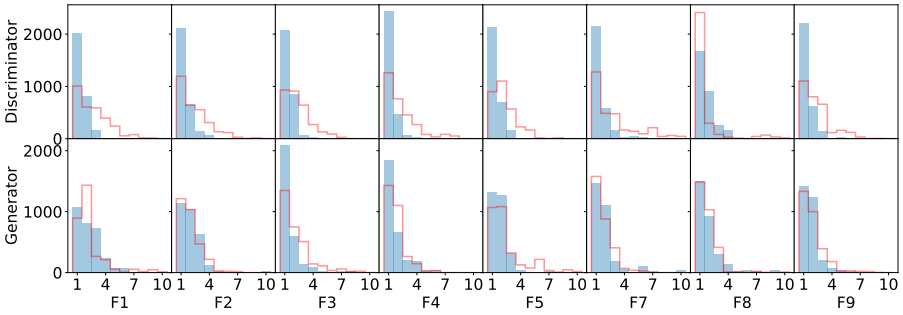


Fig. A.6: Number of layers present in the two networks of fully evolved GANs, for all functions with both problem dimensions.

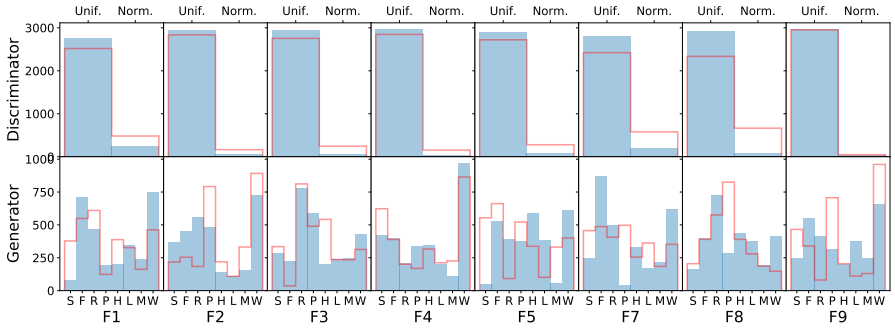


Fig. A.7: Loss functions and prior distributions used by the fully evolved GANs, for all functions with both problem dimensions. T: Total variation, S: Standard divergence, F: Forward KL, R: Reverse KL, P: Pearson χ^2, L: Least Squares.

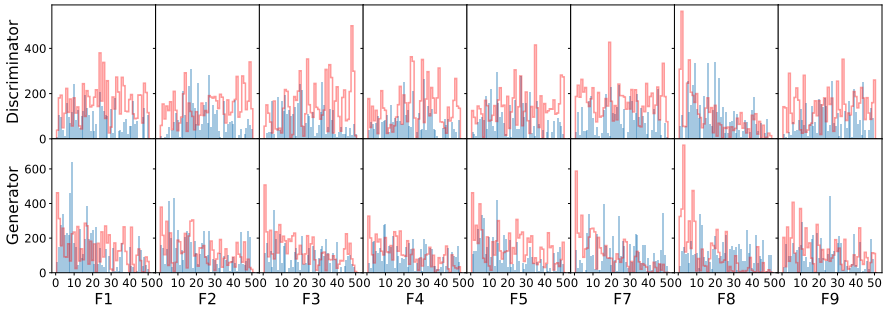


Fig. A.8: Layer sizes used by the fully evolved GANs, for all functions with both problem dimensions.

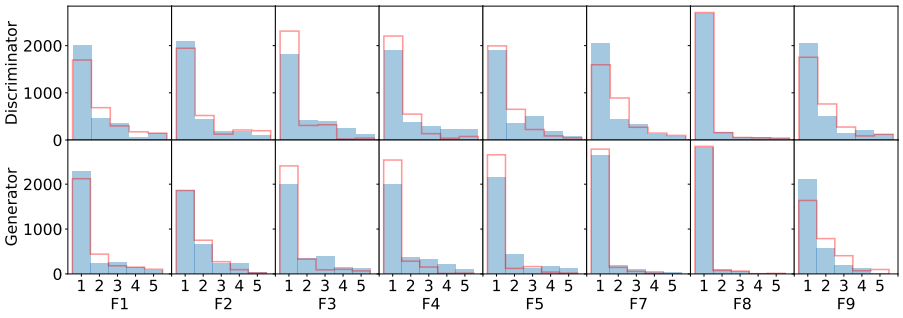


Fig. A.9: Loop parameters used by the fully evolved GANs, for all functions with both problem dimensions.

eman ta zabal zazu

Universidad del País Vasco

Euskal Herriko Unibertsitatea