EvoFlow: A Python library for evolving deep neural network architectures in tensorflow

Unai Garciarena*, Roberto Santana†, and Alexander Mendiburu‡
Intelligent Systems Group, University of the Basque Country (UPV/EHU)
Donostia-San Sebastian, Spain
Email: *unai.garciarena@ehu.eus, †roberto.santana@ehu.eus, ‡alexander.mendiburu@ehu.eus

***Abstract*—Neuroevolutionary algorithms are one of most effective and extensively applied methods for neural architecture search. While several neuroevolutionary approaches have been proposed, the availability of software that allows a fast development of code to solve problems and test research questions is limited. In this paper we introduce EvoFlow, a Python library for evolving shallow and deep neural network (DNN) architectures. EvoFlow optimizes network structures for DNNs implemented in tensorflow. Single and multi-component DNN architectures are represented by means of descriptors, and the instantiation of the network occurs in the evaluation of the architecture. Genetic operators work by modifying the descriptors. We show how EvoFlow allows efficient architecture optimization of single-component DNNs, such as deep multi-layer perceptrons, but also of multi-component DNNs, such as generative adversarial nets.**

***Index Terms*—neuroevolution, deep neural networks, genetic algorithms, GANs**

I. Introduction

The Neuroevolution (NE) research field has been one of the most active ones in the last few years in the neural structure search area, producing a large amount of works [1], [2], [3], [4], [5], [6], [7], [8]. These methods have proven to be competitive when compared to expert human knowledge at designing DNN architectures. However, a large part of the effort produced by the research community is not efficiently presented for other peers to take advantage of. Only exceptionally, published works release accompanying software for the benefit of the research area [2], [9], [1], and authors are therefore forced to re-implement methods, which is a waste of effort, or directly abandon or change promising ideas in order to save time. Additionally, most of the developed methods are highly dependent on the problem-type they were developed for. For example, a method that automatically designs networks for image classification (commonly employing convolutional operations) would probably not be able to handle temporal data (which generally require recurrent connections).

In addition to all of this, as the boundaries of DNN complexities are expanded with different objectives (e.g., seeking improved performance, or adapting DNN models to paradigms other than the classic regression or classification), models which encapsulate more than a single DNN have started to appear. These multi-network models do not need to be homogeneous in terms of the architecture they utilize, this is, while one network can be shaped as a convolutional DNN (CNN), another network in the same model could be designed as a multi-layer perceptron (MLP).

As stated earlier, the main architectures of existing and available evolutionary approaches are heavily problem-bounded [2], [9], [1]. This lack of flexibility is even more apparent when, instead of requiring a single neural network for solving a problem, multiple heterogeneous networks are needed [10].

Finally, and this is one of the main reasons that motivates our work, there are key questions in the design and implementation of neuroevolutionary software that can have a *direct impact* on relevant research directions for neuroevolution as they have had in other fields of evolutionary computation [11], [12], [13], [14], [15]. For instance, the simplifications and assumptions made in the software when deciding the representation of the neural architecture will strongly influence the type and scope of the practical problems that could be solved. They also bias the type of hypothesis that can be tested by the algorithms and therefore delimit, to some extent, the research that can be carried out. Therefore, we stress the importance of discussing and going deeper into the analysis of neuroevolutionary software, and this paper takes a step in this direction.

This paper presents a software tool aimed at giving response to the problems introduced above. EvoFlow is an open-source tool that does not impose any kind of restriction over the heterogeneity of the networks involved in a model or in the architecture of the networks. It is designed in such a way that the user can be involved at different depth levels regarding the design of the evolutionary algorithm. From evolving a simple MLP for a regression or a classification problem, to customizing any aspect of the evolutionary process, EvoFlow is able to serve a wide range of users with different levels of knowledge on deep learning or evolutionary algorithm matters.

This work has been supported by the TIN2016-78365-R (Spanish Ministry of Economy, Industry and Competitiveness), PID2019-104966GB-I00 (Spanish Ministry of Science and Innovation), the IT-1244-19 (Basque Government), and 3KIA KK-2020/00049 (SPRI-Basque Government through ELKARTEK) programs. Unai Garciarena also holds a predoctoral grant (ref. PIF16/238) from the University of the Basque Country. We also gratefully acknowledge the support of NVIDIA Corporation with the donation of a Titan X Pascal GPU used to accelerate the process of training the models used in this work.

978-1-7281-2547-3/20/$31.00 ©2020 IEEE

The paper is organized as follows. In the next section we review related work on implementation of neuroevolutionary approaches. Section III describes the main principles behind the design of EvoFlow. In Section IV we present the evolutionary component of EvoFlow and discuss its characteristics. Section V describes the different involvement levels the user can embrace when using EvoFlow. Section VI illustrates the application of EvoFlow with two examples. The conclusions of the paper, and some directions for future work are presented in Section VIII and Section IX.

II. Related work: Implementations of neuroevolutionary approaches

The literature on Neuroevolution is extensive and actively expanding [16]. Furthermore, neural architecture search (NAS) [17] approaches increasingly propose hybrid algorithms that incorporate elements or concepts from evolutionary computation. Therefore, we focus our review of related work on two directions of research: 1) Main paradigms for representing and evolving neural networks, and 2) Available implementations. Note that the goal of this review is not to be comprehensive, but to provide an overview of some of the most popular neuroevolutionary algorithms and the current state regarding their available implementations.

A. Main approaches to the evolution of neural networks

One of the first popular algorithms in the NE research field is NEAT [18]. The NEAT algorithm consists of a genetic algorithm (GA) that evolves neural networks represented as graphs. The key aspect of this algorithm is that it sets a population of very simple networks as a starting point and, benefiting from the usage of ad-hoc mutation procedures, evolves the network complexity to a more appropriate level. The algorithm was tested on problems with a wide range of difficulties, from approximating the XOR function to reinforcement learning problems such as pole balancing.

This work set the ground for a number of extensions of the NEAT method, such as DeepNEAT [1], and CoDeepNEAT [2]. This last proposal consists of evolving two subpopulations; one of *empty* DNN structures and another one of small *cells* which are developed for filling the gaps in the structure. This approach was tested for evolving complex networks, such as CNNs or RNNs.

CPPN [19], another popular model based on a grid layout for forming the values of the weights, was also further developed to form DPPN [20], and combined with NEAT to assemble HyperNEAT [21].

CoSyNe [22] evolves networks by operating at the lowest possible level, as it randomly generates a sub-population of weights for each parameter in the network. Values from each sub-population are combined to form functional networks, which are evaluated. Then, the fittest individuals are improved via mutation and crossover operators in an iterative evolutionary algorithm.

Preliminary versions of EvoFlow have been successfully applied to the evolution of GANs [23] and its predefined classes and operators have been used to carry out local searches in other multi-network DNNs [10].

B. Available software

As aforementioned, currently, several implementations of some NE algorithms are publicly available. Table I summarizes some of them.

TABLE I
Some of the most popular Neuroevolutionary algorithms which have source code available for the general public. Ja: Java, Py: Python, MA: Matlab

Identifier	Reference	Language
NEAT	[24]	Ja, Py, C++, MA
CPPN	[19]	Ja, Py
HyperNEAT	[25]	Ja, Py, C#, C++
DeepNEAT	[1]	Py
CoDeepNEAT	[2]	Py
CoSyNe	[22]	C++

Most of these methods, as explained in the previous sections, are heavily domain dependent. For example, CPPN has the grid layout limitation, whereas CoDeepNEAT is mainly intended for evolving single CNNs or RNNs. What is more, some of these implementations are coded in high-abstraction libraries such as Keras or Torch. This adds another layer of restrictions to the programs.

In contrast, EvoFlow allows high customization of every aspect in the evolution. The usage of *low-level* tensorflow allows deep characterization of the different layers in a network (e.g., straightforward manipulation of the application of gradients, restriction of the values weights can take, definition of new recurrent architectures, etc.). The evolutionary aspect is also highly customizable, as the selection, evolutionary, and/or evaluation operators can be designed by the user.

Nevertheless, EvoFlow comes with enough pre-implemented modules that the end user is not required to implement all the aforementioned aspects. Instead, they can simply select the ones to be used and, if necessary, implement the ones specific to their application.

III. EvoFlow: Principles of design

There are a number of general principles that guide the design of EvoFlow. In this section, we first present these principles and then provide an overview of the way these principles are translated to the software architecture.

A. General principles

The general principles that guide the conception of this software are:

- *Modular design*: Neuroevolution is represented as a process where multiple well-defined components interact for the final goal. For example, the evolutionary component interacts with the neural network fitness evaluation component. Within each component, different elements also interact in a modular way.

- *Multiple module implementations*: A similar objective can be implemented in different ways. Alternative implementations of the similar objectives are grouped within the same module.
- *Hierarchical design*: Some of the components, most notably the neural network representations, follow a hierarchical design to promote polymorphism.
- *Predefined models and operators*: A number of models and operators are already implemented and can be used out of the box.
- *Open to the addition of new methods*: The modular design allows the addition of new methods by the user which can be combined with predefined components.
- *Avoid re-implementation*: In order to avoid useless and expensive re-implementation of functionalities, EvoFlow heavily depends on well tested libraries; `tensorflow` and `DEAP` [13].

B. Principles of neural network representation

There exists a fine balance between the level of realism or detail of a model and its ease of use. This balance is at the junction of the design of neuroevolutionary algorithms since choices have to be taken that define a compromise between the extent of the neural network components that are amenable for evolution, and the computational time that the representation, evolution and evaluation of these components require. EvoFlow representation of neural networks is based on the following decisions and assumptions:

- *Declarative representation of networks*: The neural network models are not represented in their final structure. Instead, they are specified as a set of parts that are assembled during the evaluation. More specifically, neural networks are represented as lists.
- *Weights are not evolved*: In the current implementation of EvoFlow, only the structure and some specific parameters are evolved. Weights can be optimized, or fixed once the network architecture has been assembled (see discussion in Section VIII).
- *Multiple components*: Models can include multi-component networks, e.g., generative adversarial networks (GAN) and variational autoencoders (VAE). The user can design their own multi-component networks.
- *Multiple objectives*: Each model can be evaluated according to different criteria.
- *Additional parameters*: The user can include a number of hyperparameters which are evolved by EvoFlow. The user must define the meaning of these parameters by adequately employing them in the training and/or evaluation functions. These parameters can be used for evolving the gradient optimizer, batch size, learning rate, or any other aspect relevant to a DNN which is not included in its structure.

C. Library design

EvoFlow is implemented in Python using an object-oriented design. Although the goal of this paper is not to present a technical description of the software, we discuss a number of questions in order to understand the general design.

The declarative representation principle stated above is implemented by splitting the representation of the DNN in two types of classes: The `NetworkDescriptor` class is used to represent an arbitrary network as a class. It contains the minimal information required to conform a model in tensorflow (TF). The `Network` class translates the information contained in the descriptor into a functional TF implementation.

This split allows an easier definition of the genetic operators that will interact only with the descriptors. Every network structure can be implemented as a child of `NetworkDescriptor` and the `Network` classes. EvoFlow contains the following three pre-implemented networks for their direct usage by the users.

- Class `MLPDescriptor` inherits from `NetworkDescriptor` and implements functionalities specific to an MLP (i.e., defines the number of neurons in a layer).
- Class `MLP` inherits from `Network` and uses a `MLPDescriptor` to create a functional MLP implementation in TF.
- Class `ConvDescriptor` inherits from `NetworkDescriptor` and implements functionalities specific to a CNN (e.g., defines the number of filters).
- Class `CNN` inherits from `Network` and uses a `ConvDescriptor` to create a functional CNN implementation in TF.
- Class `TConvDescriptor` inherits from `NetworkDescriptor` and implements functionalities specific to a transposed CNN (e.g., defines the number of filters).
- Class `TCNN` makes use of a `TConvDescriptor` to create a functional transposed CNN implementation in TF. It inherits from `Network`.

These already implemented classes contain the attributes and functions specific to each DNN type, and serve different purposes; they can be used by users with no extensive knowledge of the neural network technicalities, or those users that want to solve a target application without specific requirements about their architectures. The workflow of EvoFlow is graphically displayed in Figure 1.

The user can define new types of networks to be evolved by creating new classes that serve to describe the new model as a list, and to translate this list into a TF representation. These classes will respectively inherit from classes `NetworkDescriptor` and `Network`, or from the more specific classes included in EvoFlow (e.g., if a MLP with certain characteristics is required, the class could inherit from MLP directly instead of Network). The preimplemented classes can also serve as an example for users with specific needs on how to implement the customized networks. When creating multi-component models, the user can combine new components with others that already exist in the library.

To finish this section, we describe the attributes present in the main `NetworkDescriptor` class since they illustrate

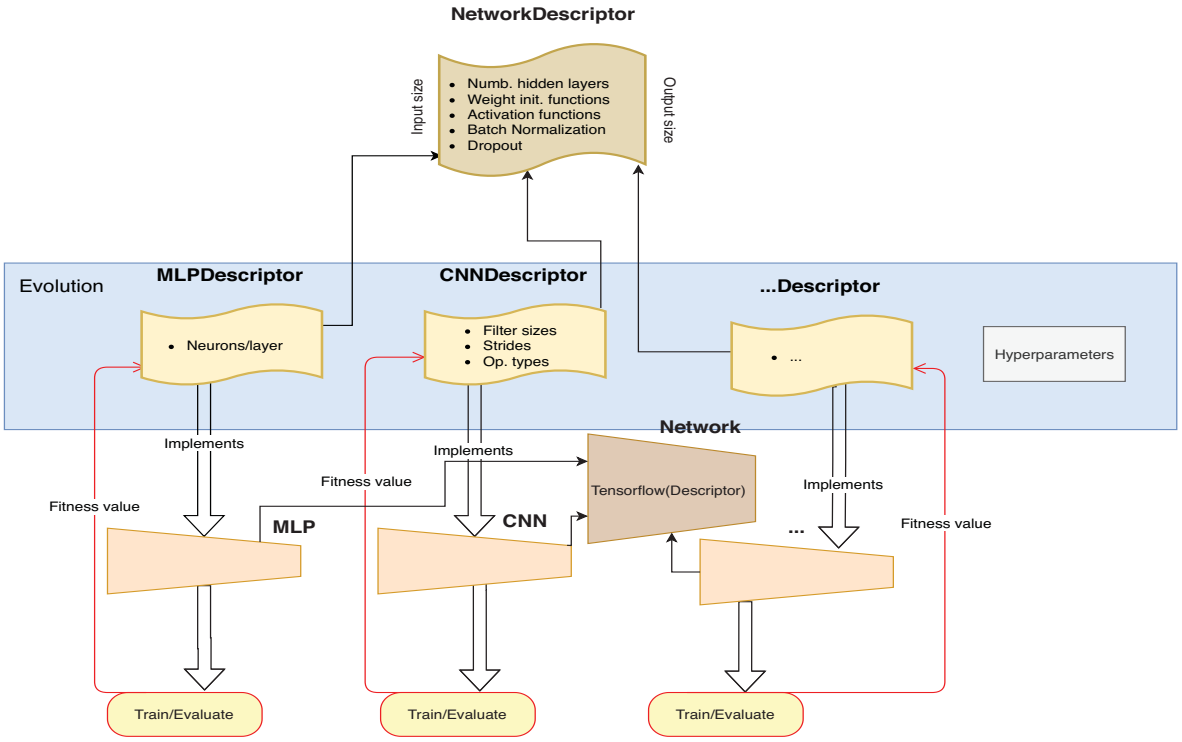

Fig. 1. Diagram showing how EvoFlow evolves network descriptors, and uses the performance of their tensorflow implementation to assess the quality of the specification.

the level of detail used for the representation of the neural networks. These attributes are:

- Number of hidden layers: A single integer.
- Input dimension: An iterable or an integer.
- Output dimension: Similar to the input, the output can have different shapes.
- Initialization functions: List of indices of functions used for initializing the random weights of the layers.
- Activation functions: List of indices of activation functions to be applied after each layer is computed. One index for each layer.
- Dropout: List of Boolean elements determining whether the application of dropout after the activation functions is applied.
- Batch normalization: List of Boolean elements indicating whether the network includes batch normalization before the activation functions.

Also, unlike in the MLP, in which only densely connected layers are viable, CNNs can use different kinds of layers. In this case, default EvoFlow implements 2-D convolutional layers, and max and mean pooling.

IV. The Evolutionary Component

The GA used to evolve DNNs uses a list-based encoding, i.e., a list of descriptors that specifies the network architecture and other parameters such as the loss function, weight initialization scheme, etc. Therefore, the representation is declarative, it contains the specification of the network but does not include the weights. In the solution evaluation step, the list of descriptors serves to instantiate a tensorflow object describing a neural network. Once instantiated, the network is trained on the available data, i.e., back-propagation is invoked and weights are learned at each architecture evaluation. For the evaluation process, one or more metrics (e.g., accuracy, learning time, etc) describing the DNN performance are computed. Solutions are selected based on the objective functions and different selection operators are available in EvoFlow through the `DEAP` library.

A distinguished characteristic of EvoFlow is that genetic operators (crossover and mutation) operate on the network descriptions. These operators can be defined by the user according to the particular network architectures. However, EvoFlow already contains a number of operators which have been already implemented.

As previously mentioned, one of the strengths of EvoFlow is that every component within it is customizable. In addition, EvoFlow features a well-rounded set of tools which can be used to perform evolution of a wide range of DNN architectures with little effort. We will now introduce the evolutionary-relevant methods which come preimplemented in EvoFlow.

A. Mutation operators

The operators used to mutate individuals are the following:

- **layer_change** is applied to a random layer in a network. It randomly reinitializes its description (e.g., the weight

initialization, and activation functions (common to all networks), and number of neurons for an MLP, or the number of kernels and their sizes for a CNN).

- **add_layer** is applied in a random position of a network, and introduces a new layer in that point. As in **layer_change**, the introduced parameters suit the network architecture.
- **del_layer** is also applied in a random position of a network. The layer at that position is deleted.
- **activ_change** can be applied to any layer in a network, it changes the employed activation function to another one.
- **weight_change** is similar to **activ_change**, but instead of changing the activation function applied after the computations, the function used to initialize the weights at the beginning of the learning procedure is changed.

Since these operators are modular and can produce different effects on the network architectures that can be combined in different ways. For instance, they could be used to implement incremental neuro-evolutionary strategies similar to the one used by the NEAT approach [18]. In the default mutation strategy implemented in EvoFlow, when a DNN is eligible for being mutated, one of these elementary operators is randomly chosen and applied. The pool of applicable mutations, however, does not always include all the operators. For a mutator to be applied to a DNN, the resulting network must hold the DNN characteristics. This means that, when a network with a single hidden layer is to be mutated, the **del_layer** operator cannot be applied. The same logic is applied when a network has reached a layer limit. In this case, **add_layer** operator can not be called.

The default implementation of EvoFlow integrates the following options for the evolvable components:

- For random weight initialization, values from a `normal` or `uniform` distribution are drawn. Additionally, when the layer is densely connected, `xavier` [26] initialization can also be applied.
- As activation functions to be applied after computation, one of the following can be chosen:
  - `Identity`
  - `ReLU`
  - `eLU`
  - `Softplus`
  - `Softsign`
  - `Sigmoid`
  - `Hyperbolic Tangent`

B. Crossover operator

While the mutation operator can be applied to any evolutionary scenario, this is not the case with crossover. This operator conducts individual reproduction at a higher level as opposed to mutation, as it does not modify the components. Rather, it performs one-point crossover, the point being randomly chosen in the [1, num_comp-1] range. That is, it exchanges the specifications corresponding to the same component between the two parents. For example, when evolving a multi-network model comprising as components one CNN and one MLP, crossover would change the two specifications of the MLP of two parents with each other to form two offspring. Crossover is not limited to network exchanging, as the set of additional hyperparameters, if present, is also a component susceptible of being interchanged.

C. Single-objective vs Multi-objective

EvoFlow is not limited to the evolution of a single objective, e.g., the loss function of a given network, or any other metric. Multi-objective evolution of structures is fully supported by the library. This can be specially useful when, for example, the user is not only interested in models which can achieve accurate results, but those which can do so in an efficient manner. To that end, the user could set a *secondary* objective which penalizes, for instance, excessively complex models.

D. Selection method

Due to the evolutionary processes of EvoFlow being inherited from DEAP, EvoFlow offers every selection method available in that library. By default, for multi-objective evolutionary procedures, EvoFlow employs the NSGA-II [11], whereas other options, such as tournament or truncation selection, are available for single-objective optimization.

Algorithm 1 contains a pseudocode of how the genetic algorithm implemented in EvoFlow works.

Algorithm 1: Genetic Algorithm (GA) for evolving DNN.

```
Set t ⇐ 0. Create a population D_0 by generating N random DNN descriptions;
while halting condition is not met do
    Evaluate D_t using the fitness function;
    From D_t, select a population D_t^S of K ≤ N solutions according to a selection method;
    Create a mating pool from the selected solutions;
    Create the offspring set O_t by applying genetic crossover with probability p_x;
    Apply mutation with probability p_m = 1 − p_x. Choice of the mutation operator is made uniformly at random;
    Create D_{t+1} by using the selection method over {D_t, O_t};
    t ⇐ t + 1;
end
```

V. EvoFlow specification levels

As previously mentioned, EvoFlow allows different levels of involvement of the user with the library. The degree of involvement depends on the knowledge and the requirements of the user. In this section we present four typical scenarios that correspond to four different types of uses given to the library.

A. Simple case

The simplest use case is that in which the end user needs to apply a basic neural network: a simple MLP, to solve a regression or classification task. The user wants to find an architecture of the network that optimizes the performance for the task.

This type of usage of EvoFlow requires no deep knowledge about the components of DNNs and the way they work, or about evolutionary algorithms. In this case, the user simply needs to specify the loss function to be used for training the model, and the evaluation procedure which will output a fitness value. Default EvoFlow integrates simple loss and evaluation functions, such as the cross-entropy and accuracy error for classification problems, and MSE for regression problems.

B. Basic-Customized case

The basic customization level requires the participation of the user and some knowledge of the internal mechanisms of neural networks. The user can specify the loss and evaluation functions employed for training and evaluating models. For example, in case the user is interested in using the cross-entropy for evaluating the model (aside from training), it is possible to implement a customized function. As in the simple case, the model being evolved cannot be formed of more than one network, or any network which is not an MLP.

C. Fully-Customized case

In the fully-customized case, the user is in charge of the implementation of the same two functions as in the basic-customization case. However, in this instance, the user needs to evolve either a multi-DNN model, or a set of hyperparameters, or both at the same time. In this case, the user should specify the way in which the networks interact with the hyperparameters and between themselves. This is done at the time of constructing the neural network model from the solution representation. For example, let us suppose the user is using EvoFlow for evolving GANs and includes the learning rate as a parameter to be evolved. In this case, the user needs to specify that the discriminator will be built from the output of the generator and the data source. Additionally, the user should implement the pipeline in which the gradient descent optimizer receives the learning rate, which is an evolved parameter contained in the representation, and uses it to train the network.

D. Model customization case

As previously mentioned, EvoFlow includes ready-to-use implementations of methods for evolving MLP, CNN (even with skip connections), and transposed CNN models. Users that intend to address machine learning tasks with other types of data, e.g., temporal data, may find this repertoire of models insufficient. For these scenarios, EvoFlow allows the creation of new `Descriptors` that encapsulate a new functionality, and their corresponding models of arbitrary complexity to be implemented in tensorflow. Following the temporal example, it would be suitable to implement `RNNDecriptor` and `RNN` classes, which would specify recurrent structures. On top of the design of these classes, the user would have to define fitting the mutation operators.

VI. Examples of applications

To provide evidence of the usability and flexibility of the introduced library, in this section we discuss two different applications of EvoFlow. 1) The evolution of a simple MLP to solve a classification (or regression problem) and 2) The neuroevolutionary design of multi-components networks to solve a generative ML task[1]. When evolving a single DNN, the selected individuals have a mutation probability of 1, whereas in the multi-network case, it is divided into 0.5 and 0.5 for mutation and crossover. In all cases, Adam optimizer was used for training/evaluating the networks.

A. Evolving a simple MLP

For testing the potential of EvoFlow as a framework for fast prototyping of evolutionary procedures of simple models, we employ the Fashion-MNIST [27] classification task as an illustrative problem. This task consists of predicting the type of clothing from a small image (28×28). There are 10 types of clothes represented in the dataset. Therefore, the classification problem is multi-class.

Assuming that `x_train` and `y_train` contain the data and labels for training, and `x_test` and `y_test` the data and labels for testing, the following two lines of code show how to invoke EvoFlow for evolving an MLP architecture that solves the problem:

```
e = Evolving(loss="XEntropy",
desc_list=[MLPDescriptor],
x_trains=[x_train], y_trains=[y_train],
x_tests=[x_test], y_tests=[y_test],
evaluation="Accuracy_error",
n_inputs=[[28, 28]],  n_outputs=[[10]],
population=500, generations=100)
res = e.evolve()
```

In this case the algorithm outputs the variable res, which is a tuple with two components, a log with relevant statistics of the evolution, and a list of the best neural network descriptors found during evolution.

B. Evolving multi-network models

In this section, we show how to employ EvoFlow to evolve GANs specifically designed for the 2-D, 8-mode Gaussian mixture approximation problem [28]. This problem consists of a set of points x uniformly generated from 8 Gaussian distributions which share $\sigma = 0.05$, but differ in the μ: $x \sim \mathcal{N}(\mu, \sigma)$, where $\mu = \{\mu_0, \mu_1, \dots \mu_7\}$ is a vector of means and σ is the same variance parameter applied to each μ_n. This problem has already been addressed in the context of GANs [28], [29].

This problem is specially designed to expose mode collapsing models, which would tend to generate points from a

[1]The application examples along with EvoFlow itself can be found in https://github.com/unaigarciarena/EvoFlow

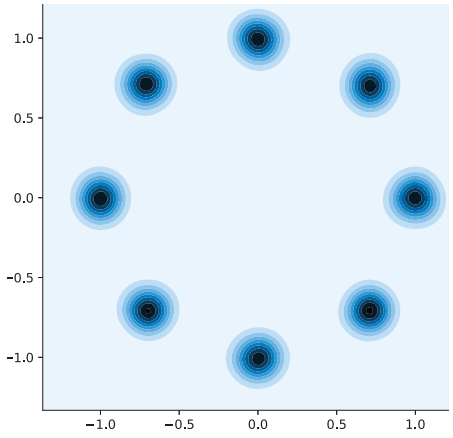

Fig. 2. Perfect distribution of the 2-D 8 Gaussian mixture problem.

subset of the 8 modes, instead of maintaining the uniform proportion, as in the original data. This is arguably one of the biggest flaws of the GAN model, as it is especially difficult to find GAN structures which do not have this characteristic. We have chosen the 2-D variant so that both the original data and the generations of the GANs can be easily visualized in a 2-D grid. Because of this, $\mu_n = (\mu_n^0, \mu_n^1)$. A visualization of the distribution of the perfect x is found in Figure 2. The 8 modes are distributed in a circle : $\mu_0 = (0, -1)$, $\mu_1 = (-\sqrt{2}, -\sqrt{2})$, $\mu_2 = (-1, 0)$, $\mu_3 = (-\sqrt{2}, \sqrt{2})$, $\mu_4 = (\sqrt{2}, -\sqrt{2})$, $\mu_5 = (1, 0)$, $\mu_6 = (\sqrt{2}, \sqrt{2})$, $\mu_7 = (0, 1)$.

In our approach using EvoFlow, GANs are encoded as two independent but related MLPs, the generator and the discriminator. To the evolution of the two networks, we have added three hyperparameters, one attached to each network, and a global one, all of which are integers. The parameters attached to each network determine the number of times each network has its weights updated with each batch. This is an important aspect of GAN training, as depending on the complexity of the problem and the characteristics of the two networks, the update rate can balance learning. The third parameter determines the gradient descent algorithm used to optimize the weights of both networks. This sets up a scenario in which two key characteristics of EvoFlow flexibility are tested; multi-network and hyperparameter evolution.

In this case, a specific metric which measures the difference between the original points and the generation needs to be used. In this conjuncture, we resort to the Maximum Mean Discrepancy[2] [30] (MMD) to assess the quality of the GANs when generating samples

VII. Results

A. Simple case

In the experiments performed as an illustration of the capabilities of EvoFlow for the simple case, 100 individuals per generation throughout 100 generations were evaluated. The training consisted of 1.000 epochs.

[2]As implemented in https://github.com/tensorflow/models/blob/master/research/domain_adaptation/domain_separation/losses.py

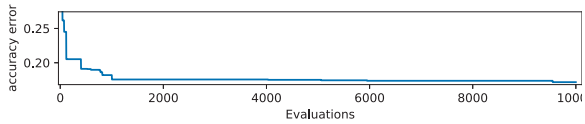

Fig. 3. Evolution of the logarithm of the accuracy error attained by the best MLPs found with EvoFlow for the Fashion-MNIST problem.

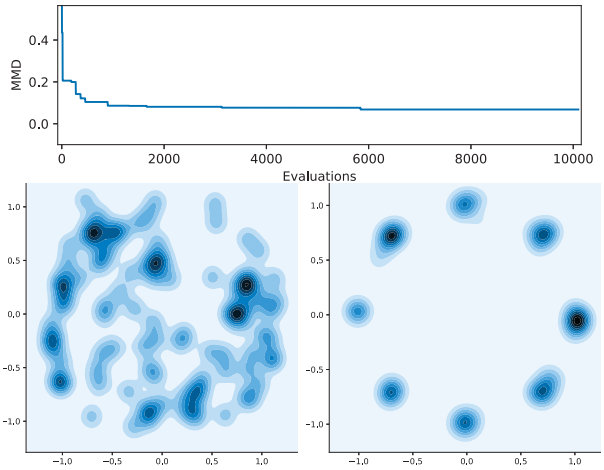

Fig. 4. Results of the 2-D 8-Gaussian mixture approximation problem.

Figure 3 shows the accuracy error achieved by the best individual found at each point of the evolution of the simple MLP for the Fashion-MNIST problem. The y axis represents the accuracy error, whereas the x axis represents the number of evaluations performed during evolution. It can be observed that the error decreases as the evolution progresses.

B. Fully-customized case

Figure 4 shows the results of the GAN evolution. The chart on the top shows, for each number of evaluations (in the x axis), the MMD generated by the best network configuration found (in the y axis). The figure at the bottom left shows an approximation performed by the best GAN found in the third generation, whereas the one on the right shows another approximation by a GAN in the last generation.

As can be deduced from comparing the results provided by a random GAN and the evolved counterpart, the evolution of the GANs is successful. Not only does the approximation made by the evolved GAN not generate points from areas from which it should not, but, more crucially, it also captures (although with different degrees) all eight modes in the original data.

VIII. Conclusions

In this paper, we have introduced EvoFlow, a flexible and powerful framework for the automatic design of heterogeneous DNN architectures. Two of the most relevant strengths of this tool are: 1) Its simplicity, which allows fast prototyping of

neuroevolutionary algorithms. 2) Its flexibility, in terms of the number of models it can evolve, the hyperparameters that can be involved in the evolutionary procedure, and the multiple ways in which the evolutionary procedure can be shaped.

We have presented and discussed the principles that guide the design of the library, and illustrated the way in which it can be used for solving practical applications. Finally, all the code, containing EvoFlow and the developed examples (among other use cases of the tool) have been made freely available.

IX. Further work

While the current implementation of EvoFlow does not support automatic weight evolution, an end user implementing their own specific evolutionary operators could integrate this feature into their evolutionary process. However, the final goal of EvoFlow in this aspect is to contain an implementation of weight evolution via appropriate generic evolutionary operators which would allow this feature, even for simple-case usages of the library.

The current version of EvoFlow supports the evolution of MLP, CNN, and transposed CNN models. However, another widely extended neural architecture has not been added to the pool of models EvoFlow can evolve; DNNs with a recurrent component. The addition of this constituent would make EvoFlow a well-rounded tool which covers all three main paradigms of deep learning. The application of the EvoFlow to different neuroevolutionary schemes which test the flexibility of the library is also an open challenge.

References

[1] R. Miikkulainen, J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy, and others, "Evolving deep neural networks," in *Artificial Intelligence in the Age of Neural Networks and Brain Computing*. Elsevier, 2019, pp. 293–312. [Online]. Available: https://doi.org/10.1016/B978-0-12-815480-9.00015-3

[2] J. Liang, E. Meyerson, and R. Miikkulainen, "Evolutionary Architecture Search for Deep Multitask Networks," in *Proceedings of the Genetic and Evolutionary Computation Conference*. New York, NY, USA: ACM, 2018, pp. 466–473.

[3] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. Le, and A. Kurakin, "Large-scale evolution of image classifiers," *arXiv preprint arXiv:1703.01041*, 2017.

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

[5] Y. Liu, J. Chen, and L. Deng, "An Unsupervised Learning Method Exploiting Sequential Output Statistics," *arXiv preprint arXiv:1702.07817*, 2017.

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

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

[8] U. Garciarena, A. Mendiburu, and R. Santana, "Analysis of the transferability and robustness of GANs evolved for Pareto set approximations," *Neural Networks*, pp. 281–296, 2020, publisher: Elsevier.

[9] K. Stanley, D. D'Ambrosio, and J. Gauci, "A hypercube-based encoding for evolving large-scale neural networks," *Artificial Life*, vol. 15, no. 2, pp. 185–212, 2009.

[10] U. Garciarena, A. Mendiburu, and R. Santana, "Automatic Structural Search for Multi-task Learning VALPs," in *International Conference on Optimization and Learning*. Cádiz, Spain: Springer, 2020, pp. 25–36.

[11] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.

[12] J. J. Durillo, A. J. Nebro, and E. Alba, "The jMetal Framework for Multi-Objective Optimization: Design and Architecture," in *Proceedings of the 2010 Congress on Evolutionary Computation CEC-2010*. Barcelone, Spain: IEEE Press, 2010, pp. 4138–4325.

[13] F.-A. Fortin, F.-M. D. Rainville, M.-A. Gardner, M. Parizeau, and C. Gagné, "DEAP: Evolutionary Algorithms Made Easy," *Journal of Machine Learning Research*, vol. 13, no. 70, pp. 2171–2175, 2012.

[14] J. Ocenasek, S. Kern, N. Hansen, and S. M. a. P. Koumoutsakos, "A Mixed Bayesian Optimization Algorithm with Variance Adaptation," *Lecture Notes in Computer Science*, pp. 352–361, 2004.

[15] M. Pelikan, "A Simple Implementation of Bayesian Optimization Algorithm in C++ (Version1. 0)," University of Illinois at Urbana-Champaign, Illinois Genetic Algorithms Laboratory, Urbana, IL, IlliGAL Report No. 99011, 1999.

[16] A. D. Martinez, J. Del Ser, E. Villar-Rodriguez, E. Osaba, J. Poyatos, S. Tabik, D. Molina, and F. Herrera, "Lights and Shadows in Evolutionary Deep Learning: Taxonomy, Critical Methodological Analysis, Cases of Study, Learned Lessons, Recommendations and Challenges," *CoRR*, vol. abs/2008.03620, 2020. [Online]. Available: http://arxiv.org/abs/2008.03620

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

[18] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary Computation*, vol. 10, no. 2, pp. 99–127, 2002.

[19] K. O. Stanley, "Compositional pattern producing networks: A novel abstraction of development," *Genetic Programming and Evolvable Machines*, vol. 8, no. 2, pp. 131–162, 2007.

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

[21] D. B. D'Ambrosio and K. O. Stanley, "A novel generative encoding for exploiting neural network sensor and output geometry," in *Proceedings of the 9th annual conference on Genetic and evolutionary computation*. ACM, 2007, pp. 974–981.

[22] F. Gomez, J. Schmidhuber, and R. Miikkulainen, "Accelerated neural evolution through cooperatively coevolved synapses," *Journal of Machine Learning Research*, vol. 9, no. May, pp. 937–965, 2008.

[23] U. Garciarena, R. Santana, and A. Mendiburu, "Evolved GANs for generating Pareto set approximations," in *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, 2018, pp. 434–441.

[24] K. O. Stanley and R. Miikkulainen, "Efficient reinforcement learning through evolving neural network topologies," in *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*. Morgan Kaufmann Publishers Inc., 2002, pp. 569–577.

[25] D. B. D'Ambrosio and K. O. Stanley, "Generative encoding for multiagent learning," in *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, 2008, pp. 819–826.

[26] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, Y. W. Teh and M. Titterington, Eds., vol. 9. Chia Laguna Resort, Sardinia, Italy: PMLR, May 2010, pp. 249–256. [Online]. Available: http://proceedings.mlr.press/v9/glorot10a.html

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

[28] L. Metz, B. Poole, D. Pfau, and J. Sohl-Dickstein, "Unrolled generative adversarial networks," *arXiv preprint arXiv:1611.02163*, vol. [cs,stat], 2016. [Online]. Available: https://arxiv.org/abs/1611.02163

[29] C. Wang, C. Xu, X. Yao, and D. Tao, "Evolutionary Generative Adversarial Networks," *IEEE Transactions on Evolutionary Computation*, vol. 23, no. 6, pp. 921–934, 2019.

[30] A. Gretton, K. M. Borgwardt, M. J. Rasch, B. Schölkopf, and A. Smola, "A Kernel Two-Sample Test," *Journal of Machine Learning Research*, vol. 13, no. 25, pp. 723–773, 2012. [Online]. Available: http://jmlr.org/papers/v13/gretton12a.html