Automatic model selection for neural networks

David Laredo, Jian-Qiao Sun, Yulin Qin

***Abstract*—Neural networks and deep learning are changing the way that artificial intelligence is being done. Efficiently choosing a suitable model (including hyperparameters) for a specific problem is a time-consuming task. Choosing among the many different combinations of neural networks available gives rise to a staggering number of possible alternatives overall. Here we address this problem by proposing a fully automated framework for efficiently selecting a neural network model given a specific problem (whether it is classification or regression). Our proposal focuses on a distributed decision-making algorithm for keeping the most promising models among a pool of possible models for one of the major neural network architectures, Multi-Layer Perceptrons (MLPs). This work develops Automatic Model Selection (AMS), a modified micro genetic algorithm that automatically and efficiently finds the most suitable neural network model for a given dataset. Our contributions include: a simple list based encoding for neural networks as genotypes in an evolutionary algorithm, new crossover and mutation operators, the introduction of a fitness function that considers both, the accuracy of the model and its complexity and a method to measure the similarity between two neural networks. Our evaluation on two different datasets show that AMS effectively finds suitable neural network models while being efficient in terms of the computational burden, our results are compared against other state of the art methods such as Auto-Keras and AutoML.**

***Index Terms*—artificial neural networks, model selection, hyperparameter tuning, distributed computing, evolutionary algorithms.**

I. Introduction

MACHINE learning studies algorithms that improve themselves through experience. Given the large amounts of data currently available in many fields such as engineering, biomedical, finance, etc, and the increasingly computing power available machine learning is now practiced by people with very diverse backgrounds. Increasingly, users of machine learning tools are non-experts who require off-the-shelf solutions. Automated Machine Learning (AutoML) is the field of machine learning devoted to developing algorithms and solutions to enable people with limited machine learning background knowledge to use use machine learning models easily. Tools like WEKA [1], PyBrain [2] or MLLib [3] follow this paradigm. Nevertheless, the user still needs to make some choices which not may be obvious or intuitive (selecting a learning algorithm, hyperparameters, features, etc) thus leading to the selection of non optimal models.

Recently, deep learning models (CNN, RNN, Deep NN) have gained a lot of attention due their improved efficiency on complex learning problems and their flexibility and generality for solving a large number of problems such as: regression, classification, natural language processing, recommendation systems, etc. Furthermore, there are a lot software libraries which makes their implementation easier. TensorFlow [4], Keras [5], Caffe [6] and CNTK [7] are some examples of such libraries. Despite the availability of such libraries and tools, the task of picking the right neural network model and its hyperparameters is usually complex and iterative in nature specially among non computer scientist.

Usually, the process of selecting a suitable machine learning model for a particular problem is done in an iterative manner. First, an input dataset must be transformed from a domain specific format to features which are predictive of the field of interest, once features have been engineered users must pick a learning setting appropriate to their problem, e.g. regression, classification or recommendation. Next users must pick an appropriate model, such as support vector machines (SVM), logistic regression or any flavor of neural networks (NNs). Each model family has a number of hyperparameters, such as regularization degree, learning rate, number of neurons, etc, and each of these must be tuned to achieve optimal results. Finally, users must pick a software package that can train their model, configure one or more machines to execute the training and evaluate the model's quality. It can be challenging to make the right choice when faced with so many degrees of freedom, leaving many users to select a model based on intuition or randomness and/or leave hyperparameters set to default, this approach will usually yield suboptimal results.

This suggests a natural challenge for machine learning: given a dataset, to automatically and simultaneously chose a learning algorithm and set its hyperparameters to optimize performance. As mentioned in [1] the combined space of learning algorithm and hyperparemeters is very challenging to search: the response function is noisy and the space is high dimensional involving both, categorical and continuous choices and containing hierarchical dependencies (e.g. hyperparameters of the algorithm are only meaningful if that algorithm is chosen). Thus, identifying a high quality model is typically costly (in the sense that entails a lot of computational effort) and time consuming.

To address this challenges we propose Automatic Model Selection (AMS) a flexible and scalable method to automate the process of selecting artificial neural network models. The key contributions of this paper include: 1) a simple, list based encoding of neural networks as genotypes for evolutionary computation algorithms, 2) new crossover and mutation operators to generate valid neural networks models from an evolutionary algorithm, 3) the introduction of a fitness function that considers both, the accuracy of the model and its complexity and 3) a method for measuring the similarity between two neural networks. All these components together form a new method based on an evolutionary algorithm, which

All the authors are affiliated with the Department of Mechanical Engineering, University of California, Merced, CA, 95340 USA e-mail: dlaredorazo@ucmerced.edu.

we call AMS, and that can be used to find an optimal neural network architecture for a given dataset.

The remainder of this paper is organized as follows: Section II formally introduces the model selection problem. The related work is briefly reviewed in Section III. The AMS algorithm and all of its components are described in detail in Section IV, experiments to test the algorithm and comparison against other state of the art methods are presented in Section V. Conclusions and future work are discussed in Section VI.

II. Problem Statement

In this section we mathematically define the general neural network architecture search problem. Consider a dataset $\mathcal{D}$ made up of training points $d_i = (\mathbf{x_i}, \mathbf{y_i}) \in \mathcal{X} \times \mathcal{Y}$ where $\mathcal{X}$ is the set of data points and $\mathcal{Y}$ is the set of labels. Furthermore, lets split the dataset into training set D_t, cross validation set D_v, and test set D_e. Given a neural network architecture search space $\mathcal{F}$ the performance of a neural network architecture $f \in \mathcal{F}$ on trained on $\mathcal{D}_t$ and validated with $\mathcal{D}_v$ is defined as:

$$p = Per(f(D_t); D_v), \tag{1}$$

where $Perf(.)$ is a measurement of the error attained by the learning algorithm. Common error indicators are: accuracy error, precision error and mean squared error, their definitions along with some other common error indicators are presented in Table XV.

Finding a neural network $f^* \in \mathcal{F}$ that achieves a good performance has been explored in [8], [9] among others. While this task alone is challenging, usually the efficiency of f^* is not measured. Indeed, it turns out that there can be several candidate models that can attain similar performance with improved efficiency. By efficiency we mean, in practical terms, how fast is to train f^* as compared to other possible solutions.

In this paper we aim to achive neural network models f^* that not only exhibit good performance in $\mathcal{D}$ as measured by p but also achieves such performance using a simple structure, which directly translates to improved efficiency of the model. To measure the complexity of the architecture we make use of the number of trainable parameters $w(f)$ of the neural network, we will also refer to $w(f)$ as the "size" of the neural network.

The problem of finding an neural network f that achieves both a good performance on the dataset $\mathcal{D}$ with a simple model can be mathematically stated as the following multiobjective optimization problem:

$$f^* = \underset{f \in \mathcal{F}}{\operatorname{argmin}} \quad (p(f), w(f)) \tag{2}$$

In this paper we develop an algorithm to efficiently solve Eq. 2.

III. Related Work

Automated machine learning has been of research interest since the uprising of deep learning. This is no surprise since selecting an effective combination of algorithm and hyperparameter values is currently a challenging task requiring both deep machine learning knowledge and repeated trials. This is not only beyond the capability of layman users with limited computing expertise, but also often a non-trivial task even for machine learning experts [10].

Until recently most state-of-the-art image classifier architectures have been manually designed by human experts. To make the the process easier and faster, researchers have looked into automated methods. These methods' goal is to find, withind a pre-specified resource limit (usually specified in terms of time, number of algorithms and/or combintations of hyperparameter values), an effective algorithm and/or combination of hyperparameter values that maximize the accuracy measure on the given machine learning problem and data set. Using an automated machine learning, the machine learning practitioner can skip the manual and iterative process of selecting and efficient combination of of hyperparameter values and neural network model, which is labor intensive and requires a high skill set in machine learning.

In the context of deep learning , neural architecture search (NAS) which aims to search for the best neural network architecture for the given learning task and dataset, has become an effective computational tool in AutoML. Unfortunately, existing NAS algorithms are usually computationally expensive where its time complexity can easily scale to $O(nt)$ where n is the number of neural architectures evaluated, and t is the average time consumption for each of the n neural networks. Many NAS approaches such as deep reinforcement learning [11], [12], [13] and evolutionary algorithms [14], [15], [16], [17] require a large n to reach good performance. Other approaches include Bayesian Optimization [18], [19] and Sequential Model Based Optimization (SMBO) [20], [21], nevertheless often times this approaches are as expensive as NAS while being more limited as to what kind of models they can explore.

In the recent years a number of tools have been made available for users to automate the model selection and/or hyperparameter tuning, in the following we present a brief description of some of them that share similarities with out method and will serve as comparisson.

A. Auto-Keras

Auto-Keras [22] constructs an edit-distance kernel for neural network which measures the distances between neural network architectures capable. Using this kernel makes searching the model structures in a tree-structured space (constructed for network morphism) into possible. Hence, a reasonable acquisition function can be proposed and Bayesian optimization can be used to guide the model selection. Auto-Keras combines Bayesian optimization and network morphism, and achieves good prediction accuracies. However, it is still very complicated and time consuming model. It takes several hours to converge on MNIST data set. A new model may be trained much faster as a trade of the accuracy rate. Also, the Auto-keras will not use the weighted sum of several good candidates as the final results. Hence, the results of Auto-Keras may be as robustness as the models who do so.

B. AutoML Vision

AutoML [23] uses evolutionary algorithms to perform image classification. Without any restriction on the search space(network depth, skip connections, etc.), the algorithm starts from a simple model without convolutions and evolves the complex network. A massively-parallel and lock-free infrastructure is designed, many computers are used to search the large space using a shared file system, which saves the population. Unlike the Auto-Keras, without any guide during the evolutionary process, AutoML requires extreme high computational power. Also, starting from the simple poor performance models may make the results not restricted to some well known structures, rather then using some warm start structures. But it will have significantly negative influence on the training time.

C. Auto-sklearn

Auto-sklearn [21] is a system designed to help machine learning users by automatically searching through the joint space of sklearn's learning algorithms and their respective hyperparameter settings to maximize performance using a state-of-the-art Bayesian optimization method. Auto-sklearn addresses the model selection problem by treating all of sklearn algorithms as a single, highly parametric machine learning framework, and using Bayesian optimization to find an optimal instance for a given dataset. Auto-sklearn also natively supports parallel runs (on a single machine) to find good configurations faster and save the n best configurations of each run instead of just the single best. Nevertheless, and to the best of our knowledge Auto-sklearn does not provide support for neural networks and it does not take into consideration the complexity of the proposed models for assesing their optimality.

IV. An evolutionary algorithm for the neural network model selection problem

While there are a number of methods for automatic model selection and hyperparameter tuning, many of them do not provide support for some of the most sophisticated deep learning architectures. For instance, Auto-sklearn it does not provide good support for large machine learning problems, nor provides good support for distributed computing. Furthermore, Auto-sklearn lacks support for neural networks. Auto-keras and AutoML vision, which provide support for neural networks obtaining models with a high accuracy degree, do not consider the complexity (size) of the neural network when assessing the performance of the models. Furthermore, none of them provide support for regression problems.

We propose an efficient method that, given a dataset $\mathcal{D}$, will automatically find a neural network model that attains high performance while being computationally efficient. The proposed model is capable of performing inference tasks for both, classification and regression problems. Furthermore, the proposed system is scalable and easy to use in distributed computing environments, allowing it to be usable for large datasets and complex models, for such task we make use of Ray [24], a distributed framework designed with large scale distributed machine learning in mind.

Our method provides support for three of the major neural networks architectures, namely multilayer percpetrons (MLPs) [25], convolutional neural networks (CNNs) [26] and recurrent neural networks (RNNs) [27]. Our method can construct models of any of these architectures by stacking together a *valid* combination of any of the four following layers: fully connected layers, recurrent layers, convolutional layers and pooling layers. Our method does not only build neural networks for the aforementioned architectures, but also tunes some of its hyperparameters such as the number of neurons at each layer, the activation function to use or the dropout rates for each layer. Support for skip connections is left for future work.

We say that a neural network architecture is *valid* if it complies with the following set of rules, which we derived empirically from our practice in the field:

- A fully connected layer can only be followed by another fully connected layer
- A convolutional layer can be followed by a pooling layer, a recurrent layer, a fully connected layer or another convolutional layer
- A recurrent layer can be followed by another recurrent layer or a fully connected layer.
- The first layer is user defined according to the type of architecture chosen (MLP, CNN or RNN)
- The last layer is always a fully connected layer with either a softmax activation function for classification or a linear activation function for regression problems

A. Automatic Model Selection (AMS)

The key idea of our method is to develop an evolutionary algorithm (EA) capable of evolving different neural network architectures to find a suitable model for a given dataset $\mathcal{D}$, while being computationally efficient. EAs were chosen for this work since, contrary to the more classical optimization techniques, they do not make any assumptions about the problem, treating it as a black box that merely provides a measure of quality given a candidate solution. Furthermore, they do not require the gradient, which is impossible to obtain for a neural network f, when searching for optimal solutions.

In the following we describe the very basics of evolutionary algorithms as an introduction for the reader, further reading can be found in [28], [29], [30].

Every evolutionary algorithm consists of a population of individuals where each individual in the population (neural network model) is indeed a potential solution to the optimization problem (see equation 5). Every individual has a specific genotype (encoding), in the evolutionary algorithm domain, that represents a solution to the given problem while the actual representation of the individual, in the specific application domain, is often referred as the phenotype. In particular, for this application the fenotype represents the neural network architecture while the genotype is represented by a list of lists. To assess the quality of an individual EAs make use of a so-called fitness function, which indicates how every

individual in the population performs with respect to a certain performance indicator, establishing thus an absolute order among the various solutions and a way of fairly comparing them against each other.

New generations of solutions are created iteratively by using crossover and mutation operators. The crossover operator is an evolutionary operator used to combine the information of two parents to generate a new offspring while the mutation operator is used to maintain genetic diversity from one generation of the population to the next.

The basic template for an evolutionary algorithm is the following:

Algorithm 1 Basic Evolutionary Algorithm

Let $t = 0$ be the generation counter
Create and initialize an n_x-dimensional population, $\mathcal{C}(0)$, to consist of n_s individuals
while stopping condition not true **do**
  Evaluate the fitness, $f(\mathbf{x}_i(t))$, of each individual, $\mathbf{x}_i(t)$
  Perform reproduction to create offspring
  Select the new population, $\mathcal{C}(t+1)$
  Advance to the new generation, i.e. $t = t + 1$
end while

One of the major drawbacks of EAs is the time penalty involved in evaluating the fitness function. If the computation of the fitness function is computationally expensive, as in this case, then using any flavor of EA may be very computationally expensive and in some instances unfeasible. Micro-genetic algorithms [31] are one variant of GAs whose main advantage is the use of small populations (less than 10 individuals per population) in contrast to some other EAs like the genetic algorithms (GAs), evolutionary strategies (ES) and genetic programming (GP) [25]. Since computational efficiency is one of our main concerns for this work we will follow the general principles of micro-GA in order to reduce the computational burden of our method.

The pseudocode for our proposed method is described in Algorithm 2. Let C_p and M_p be the crossover and mutation probabilities respectively, let also G be the maximum number of allowed generations and E the maximum number of repetitions for the micro-GA, finally let $\mathcal{B}$ be an archive for storing the best architectures found at every run of the micro-GA. Our algorithm, which we call Automatic Model Selection (AMS), is as follows:

In the following sections we describe in detail each one of the major components of the AMS algorithm.

B. The fitness function

To establish a ranking between the different tested architectures a suitable cost (fitness) function is required. While equation 2 can be used as the cost function, using it would give rise to a multi-objective optimization problem (MOP). We leave this approach for a future revision of this work and instead make use of scalarization to transform the MOP into a single- objective optimization problem (SOP). The scalarization approach taken here is the well known weighted sum method [32], thus equation 2 is restated as:

Algorithm 2 AMS

Let $t_e = 0$ be the experiments counter
while $t_e < E$ **do**
  Let $t_g = 0$ be the generation counter
  Create and initialize an initial population $\mathcal{C}(0)$, consisting of n_s individuals, where $n_s <= 10$. See section IV-D
  while $t_g < G$ or nominal convergence not reached **do**
    Check for nominal convergence in $\mathcal{C}(t)$. See section IV-H
    Evaluate the cost, $c(f)$ of each candidate model f. See section IV-B
    Identify best and worst models in $\mathcal{C}(t)$
    Replace worst model in $\mathcal{C}(t)$ with best from $\mathcal{C}(t-1)$
    Perform selection. See section IV-E
    Perform crossover of models in $\mathcal{C}(t)$ with $C_p = 1$. Let $\mathcal{O}(t)$ be the offspring population. See section IV-F
    For each model in $\mathcal{O}(t)$ perform mutation with M_p probability. See section IV-G
    Make $\mathcal{C}(t+1) = \mathcal{O}(t)$
    $t_g = t_g + 1$
  end while
  Append best solution from previous run to $\mathcal{B}$
  $t_e = t_e + 1$
end while
Normalize the cost for each model in the archive $\mathcal{B}$. See section IV-B
Final Solution is best existing solution in $\mathcal{B}$

$$f^* = \underset{f \in \mathcal{F}}{\operatorname{argmin}} \quad (1-\alpha)p(f) + \alpha w(f). \tag{3}$$

The cost function associated with equation 3 is

$$c(f) = (1-\alpha)p(f) + \alpha w(f), \tag{4}$$

where $\alpha \in [0, 1]$ is a scaling factor biasing the total cost towards the size of the network or its performance. Equation 3 measures the cost in terms of performance and size of a given neural network f.

Note that making an accurate assessment of the inference performance, $p(f)$, of a neural network involves training f for a large number of epochs. Since the training process usually involves thousands of computations training every candidate solution and then assessing its performance becomes unfeasible. Instead we relax the training process for each of the candidate models by using a *partial train* which, in short, is training the model for a very small number of epochs (only a few tenths of them). This approach has been successfully tested in [33] since eventhough the models are just partially trained, a clear trend in terms of whether a model is promising or not can be clearly observed.

Computing the fitness of individuals using the current definitions of $p(f)$ and $w(f)$ posses a big problem, namely that the performance indicator $p(f)$ and the number of trainable weights $w(f)$ are on entirely different scales. While $w(f)$ can range from a few hundreds up to several millions, the range

$Per(f)$	Range	Common range
Accuracy	$[0, 1]$	$[0, 1]$
Precision	$[0, 1]$	$[0, 1]$
Recall	$[0, 1]$	$[0, 1]$
MSE	$[0, +\infty]$	$[0, 10^4]$
RMSE	$[0, +\infty]$	$[0, 10^2]$

TABLE I: Common ranges for some neural network performance indicators.

of $p(f)$ depends on the type of scoring function used, Table I presents some common ranges for $p(f)$.

It is necessary for our application that the range of $p(f)$ is consistent no matter the type of score $Per(.)$, thus, we normalize the values of $p(f)$ to be within the range $[0, 1]$. The normalization process is quite straightforward: assuming a population $\mathcal{C}$ with n individuals, let $\mathcal{P} = [p_0(f), p_1(f), \ldots, p_n(f)]$ be the vector whose components are the scores p_i for the ith element in the population. Then $\mathcal{P}^* = \mathcal{P}/norm_2(\mathcal{P})$ is the vector with the normalized values of $\mathcal{P}$, hence $p_i^* \in [0, 1]$ for any score $Per(.)$.

Now we focus on $w(f)$. For the sake of simplicity lets just consider the case of the MLP class of neural networks since this is usually the model where $w(f)$ is larger. Let $\mathcal{L}$ be the maximum number of possible layers for any model, for this work we will limit $\mathcal{L} = 64$ since we consider this a pretty decent number for most of the mainstream deep learning models. From Table XVI we know that the maximum number of neurons at any layer is 1024, thus the maximum number of $w(f)$, for any given model is $\mathcal{W} = 2^{26}$. Furthermore, we want neural networks that are similar (by a few thousands of parameters) to have the same score $w(f)$, therefore we replace the last 3 digits of the $w(f)$ with $0's$, let $w^+(f)$ be this new size score. Finally, considering that $p^*(f) \in [0, 1]$, we make $w^*(f) = log(w^+(f))$, (therefore $\mathcal{W}^* \approx log(2) * 26$), hence $w^*(f) \in [0, 7.8]$ which is in the same order of magnitude as $p^*(f)$.

Thus, we redefine Equation 4 as:

$$c(f) = 10(1 - \alpha)p^*(f) + \alpha w^*(f), \tag{5}$$

where we multiply $p^*(f)$ by a factor of 10 to make the scaling even more similar as to that of $w^*(f)$. As can be observed, Equation 5 is now properly scaled, and therefore it is a suitable choice as the fitness function for assessing the performance of a neural network model while also considering its size.

C. Neural networks as lists

In order to perform the optimization of neural network architectures a suitable encoding for the neural networks is needed. A good encoding has to be flexible enough to represent neural network architectures of variable length while also making it easy to verify the *validity* of a the candidate neural network architecture.

Array based encodings are quite popular for numerical problems, nevertheless they often use a fixed-length genotype which is not suitable for representing neural network architectures. While it is possible to use an array based representation for encoding a neural network, this would require the use of very large arrays, furthermore verifying the validity of the encoded neural network is hard to achieve. Three-based representation as those used in genetic programming [25] enables more flexibility when it comes to the length of the genotype, nevertheless imposing constraints for building a valid neural network requires traversing the entire tree or making use of complex data structures every time a new layer is to be stacked in the model.

For this work we introduce a list-based encoding. In this new list-based encoding neural network models are represented as a list of arrays, where the length of the list can be arbitrary. Each array within the list represents the details of a neural network layer as described in Table XVI. A visual depiction of the array is presented in Figure 1.

Layer type	Number of neurons	Activation function	CNN filter size	CNN kernel size	CNN stride	Pooling size	Dropout rate

Fig. 1: Visual representation of a neural network layer as an array.

The proposed representations is capable of handling different types of neural network architectures, in principle this representation can handle multi-layer perceptrons (MLPs), convolutional neural networks (CNNs) and recurrent neural networks (RNNs). For any given neural network layer the array will only contain values for the entries that are applicable to the layer, values for other types of layers are set to 0.

Let us illustrate the proposed encoding with an example. The following example considers an MLP, thus only Layer type, Number of neurons, Activation function and Dropout rate entries are applicable, the rest will remain as 0. Consider S_e as a model made up of several stacked layers as those shown in Figure 1.

$$\begin{aligned} S_e = [&[1, 264, 2, 0, 0, 0, 0, 0], [5, 0, 0, 0, 0, 0, 0, 0, 0.65], \\ &[1, 464, 2, 0, 0, 0, 0, 0], [5, 0, 0, 0, 0, 0, 0, 0, 0.35], \\ &[1, 872, 2, 0, 0, 0, 0, 0], [1, 10, 3, 0, 0, 0, 0, 0]] \end{aligned}$$

The neural network representation of the presented model is shown in Table II.

Layer type	Neurons	Activation Function	Dropout Ratio
Fully connected	264	ReLU	n/a
Dropout	n/a	n/a	0.65
Fully Connected	464	ReLU	n/a
Dropout	n/a	n/a	0.35
Fully Connected	872	ReLU	n/a
Fully Connected	10	Softmax	n/a

TABLE II: Neural network model.

Encoding the neural network as a list of arrays presents two big advantages over other representations. First, the number of layers that can be stacked is, in principle, arbitrary. Second, the validity of an architecture can be verified in constant time every time a new layer is to be stacked to the model, this is due to the fact that in order to stack a layer in between the model

one just needs to check for compatibility between the previous and next layers. The ability of stacking layers dynamically and verifying its correctness as a new layer is stacked allows for a powerful representation that can build several kinds of neural networks such as fully connected, convolutional and recursive. The rules for stacking layers together are described in Table XVII.

D. Generating valid models

Having the rules for stacking layers together, generating valid models is straightforward. An initial layer type has to be specified by the user, the initial layer type can be Fully-Connected, Convolutional or Recurrent. Defining the initial layer type effectively defines the type of architectures that can be generated by the algorithm, i.e. if the user chooses FullyConected as the initial layer, all the generated models will be fully connected, if the user chooses Convolutional as initial layer the algorithm will generate Convolutional models only and so on.

Just as the initial layer type has to be user defined, the final/output layer is also user defined, in fact, all of the generated models share the same output layer. The output layer is always a FullyConnected layer, furthermore, it is generated based on the type of problem to solve (classification or regression). In the case of classification problems the number of neurons is defined by the number of classes in the problem and the softmax function is used as activation function. For regression problems the number of neurons is one and the activation function used is the linear function.

Having defined the architecture type and the output layer, generating an initial model is an iterative process of stacking new layers that comply with the rules in Table XVII. A user defined parameter n_p is used to stop inserting new layers, every time a new layer is stacked in the model a random number $n_r \in [0, 1]$ is generated using the following probability distribution

$$n_r = 1 - \sqrt{1 - U}, \tag{6}$$

where U is a number drawn from a uniform distribution. If $n_r < n_p$ and if the current layer is compatible with the last layer (according to Table XVII) then no more layers are inserted. Equation 6 is used to let the user choose the probability with which more layers are stacked to a neural network model, thus, if the user wants that a new layer is inserted with an 80% of probability the user must choose $n_p = 0.8$. With regards to layers that have an activation function, even though in principle any valid combination is possible, for this application we choose to keep all the activations for similar layers the same across the model since this is usually the common practice.

E. Selection

In order to generate n_s offsprings $2n_s$ parents are required. The parents are chosen using a selection mechanism which takes the population $\mathcal{C}(t)$ at the current generation and returns a list of parents for crossover. For our application, the selection mechanism used is based on the binary tournament selection [25], [31]. A description of the mechanism is given next:

- Select m_s parents at random where $m_s < n_s$.
- Compare the selected elements in a pair-wise manner and return the most fit individuals.
- Repeat the procedure until $2n_s$ parents are selected.

It is important to note in the above procedure that the larger m_s is, the more the probable that the best individual in the population is chosen as one of the parents, this is not a desirable behavior, thus we warn the users to keep m_s small. Also, recall from Algorithm 2 that our approach uses elitism, therefore the best individual of a current generation remains unchanged in the next generation.

F. Crossover operator

Since the encoding chosen for this task is rather peculiar the existing operators are not suitable for it, indeed, we designed a new crossover operator. In this section we describe in detail the proposed crossover operator. Our operator is based on the two point crossover operator for genetic algorithms [34] in the sense that two points are selected for each parent, nevertheless our operator is more restrictive, as to which pairs of points may be selected, in order to ensure the generation of valid architectures.

The key concept behind our crossover operator is that of "compatibility between pairs of points". Consider two models S_1, S_2 that will serve as parents for one offspring, assume that the offspring will be generated by replacing some layers in S_1 from some layers in S_2, S_1 is thus the base parent. If we selected any two pairs of points (r_1, r_2) from S_1 and (r_3, r_4) from S_4 it may happen that such pairs of points can not be interchangeable (because layer r_3 can not be placed instead of layer r_1 or layer r_4 can not be placed instead of layer r_2). Therefore, the selection mechanism must ensure that the interchange points, $(r_1, r_2), (r_3, r_4)$, are compatible, i.e. that is that layer r_3 is compatible with the layer preceding r_1 and that the layer after r_2 is compatible with layer r_4, compatibility is defined in terms of the rules described in Table XVII. A selection mechanism that guarantees this compatibility between pairs of points is described in Algorithm 3, as mentioned before, this method assumed that the offspring will be generated by replacing some layers in S_1 from some layers in S_2 .

It is possible that the mechanism described Algorithm 3 requires more than one attempt to find valid interchange points, $(r_1, r_2), (r_3, r_4)$, for models S_1 and S_2. Based on our experience with the method and the obtained results, Algorithm 3 usually requires only one attempt to successfully generate a valid offspring, nevertheless to prevent the crossover mechanism from getting trapped in an infinite loop we limit the number of trials done to n_c, where $n_c = 3$ is the default but can be adjusted by the user to its needs. Let us illustrate Algorithm 3 with an example. Consider the following models:

Algorithm 3 Crossover Method

Let S_1, S_2 be the arrays containing the stacked layers of a neural network model in parents 1 and 2 respectively.
Take two random points (r_1, r_2) from S_1 where $r_1 <= r_2$
if $r_1 = r_2$ **then**
  $r_2 = len(S_1 - 1)$
else
  pass
end if
Find all the pairs of points $(r_3, r_4)_i$ in S_2 that are compatible with (r_1, r_2) where $r_3 < r_4$ and $(r_4 - r_3) - (r_2 - r_1) < \mathcal{L}$
Randomly pick any of the pairs $(r_3, r_4)_i$
Replace the layers in S_1 between r_1, r_2 inclusive with the layers in S_2 between r_3, r_4 inclusive. Label the new model as S_3
Rectify the activation functions of S_3 to match the activation functions of S_1

$$S_1 = [[1, 264, 2, 0, 0, 0, 0, 0], [5, 0, 0, 0, 0, 0, 0, 0.65], [1, 464, 2, 0, 0, 0, 0, 0], [5, 0, 0, 0, 0, 0, 0, 0.35], [1, 872, 2, 0, 0, 0, 0, 0], [1, 10, 3, 0, 0, 0, 0, 0]]$$

$$S_2 = [[1, 56, 0, 0, 0, 0, 0, 0], [5, 0, 0, 0, 0, 0, 0, 0.25], [1, 360, 0, 0, 0, 0, 0, 0], [1, 480, 0, 0, 0, 0, 0, 0] [1, 88, 0, 0, 0, 0, 0, 0], [5, 0, 0, 0, 0, 0, 0, 0.2], [1, 10, 3, 0, 0, 0, 0, 0]]$$

Lets take $r_1 = 1$ and $r_2 = 3$, since these points are going to be removed from the model we need to find the compatible layers with $S_1[r_1 - 1]$ and $S_1[r_2]$ according to the rules described in Table XVII. Note however that if $r_1 = 0$, i.e. the initial layer, then only a layer whose layer type is equal to the layer type of $S_1[0]$ is compatible. Thus, for this example the compatible pairs of points $(r_3, r_4)_i$ are:

$$[(0,0), (0,2), (0,4), (0,5), (1,2), (1,4), (1,5), (2,2), (2,4), (2,5), (4,4), (4,5), (5,5)]$$

Assume we pick at random the pair $(2, 4)$, thus the offspring, which we will call S_3, looks like:

$$S_3 = [[1, 264, 2, 0, 0, 0, 0, 0], [1, 360, 2, 0, 0, 0, 0, 0], [1, 480, 2, 0, 0, 0, 0, 0], [1, 88, 2, 0, 0, 0, 0, 0], [1, 872, 2, 0, 0, 0, 0, 0], [1, 10, 3, 0, 0, 0, 0, 0]]$$

which is a valid model. The reader is encouraged to check the actual neural network representations for each of the models in this example in Tables XIX to XXI. Notice though that all the activation functions of the same layer types are changed to match the activation functions of the first parent S_1. We call this process "activation function rectification" and it basically implies changing all the activation functions of the layers that share the same layer type between S_1 and S_3 to the activation functions used in S_1.

Finally, one important feature of this crossover operator is that it has the ability to generate neural network models of different sizes, i.e. it can shrink or increase the size of the base parent. This behavior mimics that of machine learning practitioners, which will often start with a base model and iteratively shrink or increase the size of the base model in order to find the one that has the best inference performance.

G. Mutation operator

The mutation operator is used to induce small changes to some of the models generated through the crossover mechanism. In the context of evolutionary computation these subtle changes tend to improve the exploration properties of the current population (genetic diversity) by injecting random noise to the current solutions. Although according to [31] mutation is not needed in the micro-GA, we believe some sort of mutation is needed in our application in order to get more diverse models which could potentially lead to better inference abilities, nevertheless, our mutation approach will be less aggressive in order to mitigate its effect. In the following we discuss the core ideas of our mutation mechanism.

As stated above our mutation approach is less aggressive than common mutation operators [25], this design follows two main reasons: First, is the fact that usually micro genetic algorithms don't make use of the mutation algorithm since the crossover operator has already induce significant genetic diversity in the population, thus we want to minimize its impact. The second reason is related to the way neural networks are usually built by human experts; commonly, experts try a number of models and then make subtle changes to each of them in order to try to improve the inference ability of them, such changes usually involve changing the parameters in a layer, adding or removing a layer, adding regularization or changing the activation functions.

Based on the principles described above, our mutation process randomly chooses one layer, using a probability m_p, of the model and then proceeds to make one of the following changes to the model:

- Change a parameter of the layer chosen for a value complying the values stated in Table XVI.
- Change the activation function of the layer. This would involve rectifying the entire model (described in section IV-F).
- Add a dropout layer if the chosen layer is compatible.

This operations together provide a rich set of possibilities for performing efficient mutation while still keeping valid models after mutation is performed. Furthermore, the original model is barely changed.

H. Determining nominal convergence

Nominal convergence is one of the criteria used for early stopping of our evolutionary algorithm. Some literature defines the convergence in terms of the fitness of the individuals

[25], while in [31] the convergence is defined in terms of the genotype or phenotype of the individuals. Although convergence based on the actual fitness of the individuals may be easier to asses given that the fitness is already calculated, we believe that an assessment of convergence based on the actual genotype of the individuals suits our needs better.

Since neural networks are stochastic in nature, we expect some variations in the fitness of the individuals at every different run, furthermore since we are running the training process for only few epochs (in order to avoid a high computational burden) the final performance of the networks can be quite different and would not be a reliable indicator of convergence. Instead, to assess convergence we look at the genotype (the actual neural network architecture) and compute the layer-wise distance between the different individuals in population $\mathcal{C}$.

Let S_i, S_j, where $len(S_j) >= len(S_i)$, be the genotypes representing any two different models, also let $S_i[j]$ be the vector representation of the j-th layer of model S_i. The method for computing the distance, $d(S_i, S_j)$, between models S_i and S_j is defined in Algorithm 4.

Algorithm 4 Layer-wise distance, $d(S_i, S_j)$ between model genotypes

Let $d \in \mathbb{R}$ be the distance between S_i and S_j models. Make $d = 0$
for Each layer i in S_i except last layer **do**
  $d = d + norm_2(S_j[i] - S_i[i])$
end for
for Each remaining layer i in S_2 except last layer **do**
  $d = d + norm_2(S_j[i])$
end for
Return d

This method is computationally inexpensive since the size of the population is small. Furthermore, it helps accurately establish the similarity between any two neural network models. Given two neural network models S_i and S_j if $d(S_j, S_i) = 0$ then $S_i = S_j$. We say that AMS (Algorithm 2) has reached nominal convergence if at least m_c pairs of models have $d(S_i, S_j) \leq d_t$, where both m_c and d_t are user defined parameters.

I. Implementation

AMS is implemented in about 700 lines of code in Python. The code can be found in https://github.com/dlaredo/automatic_model_selection. We took a functional programming approach for its implementation with some helper objects. The models S_i generated by the algorithm are fetched to Keras [5] and then evaluated, nevertheless the models can be fetched and evaluated in any other framework such as TensorFlow or Pytorch and even using different programming languages such as C++ for performance reasons.

In order to boost performance of AMS we make use of Ray [24] which is a distributed computing framework tailored for AI applications. In order to distribute workloads in Ray developers only have to define Remote Functions by making use of Python annotations, Ray will then distribute these Remote Functions across the different nodes in the cluster. There are three different types of nodes in Ray: Drivers, Workers and Actors. A Driver is the process executing the user program, a Worker is a stateless process that executes remote functions invoked by a driver or another worker, finally, an Actor is a statefull process that executes, when invoked, the methods it exposes.

For our implementation we code the individual fetching to Keras and its fitness evaluation as Ray Remote Functions (Workers), while the rest of our algorithm is implemented within the Driver, therefore the partial train of each neural network within the current population is performed in a distributed way, highly increasing the performance of our algorithm. Furthermore since the only messages being sent over the cluster are arrays (the neural network representation S_i) and the fitness f of the neural network model there is little chance that the data interchanged causes a bottleneck or increases latency in the system. Nevertheless and also for performance reasons, each node in the cluster has to maintain a local copy of the dataset.

V. Evaluation

We evaluate AMS with two different datasets, each of which implies a different type of inference problem, we also compare our results with state-of-the-art neural network models for each problem.

For the experiments in this section each model generated by AMS was trained using the following parameters:

Dataset	Epochs	Learning Rate	Optimizer	Loss function	Metrics
MNIST	5	0.001	Adam	Categorical crossenthropy	Categorical accuracy
CMAPSS	5	0.01	Adam	MSE	MSE

TABLE III: Training parameters for each of the used datasets.

For CMAPSS dataset we used a larger learning rate since we are evaluating the model using very few epochs for this complex problem, therefore, in order to get a clearer idea of which individuals within the population may be promising we make the learning process more aggresive during the first iterations of the algorithm.

A. MNIST Dataset - A classification problem

We first test our algorithm on the MNIST Dataset [35]. The MNIST Dataset of handwritten digits is one of the most commonly used datasets for evaluating the performance of neural networks. It has a training set of 60,000 examples and a test set of 10,000 examples. The digits have been size-normalized and centered in a fixed-size image, the size of each image is of 28x28 pixels. Ass part of the data pre-processing step we normalized all the pixels in each image to be in the range of $[0, 1]$ and unrolled the 28x28 image into a vector with 784 components.

We use MNIST dataset as a baseline for measuring the performance of our approach, furthermore, we use MNIST to analyze each one of the major components of AutoNN. Given the popularity of MNIST, several neural networks with

varying degrees of accuracy have been proposed, therefore it is easy to find good models to compare with.

We start by running AutoNN to find a suitable FullyConnected model for classification for the MNIST dataset. The details for the parameters used in this test are described in Table IV. Each of the experiments carried out by AMS took about 4 minutes in our test rig.

Parameter	AutoNN Value
Problem Type	1
Architecture Type	FullyConnected
Input Shape	$(784, M)$
Output Shape	$(10, M)$
Cross Validation Ratio	$c_v = 0.2$
Mutation Ratio	$m_p = 0.4$
More Layers Probability	$n_r = 0.4$
Network Size Scaling Factor	$\alpha = 0.5$
Population Size	$n_s = 10$
Tournament Size	$n_t = 4$
Max Similar Models	$s = 3$
Training epochs	$t_e = 5$
Total Experiments	$E_{max} = 5$

TABLE IV: AMS Parameters for MNIST dataset.

We first take a look at the generated initial population. For the sake of space we will only discuss the sizes of the models, furthermore we make a small change in our notation for describing neural network models. For the remainder of this section we denote a layer of a neural network as (n_e, a_f) where n_e denotes the number of neurons in case the layer is fully connected or the dropout rate in case the layer is a dropout layer and a_f denotes the activation function of the later if the layer is a fully connected layer. The initial five generated individuals are presented next. Fitness, accuracy and raw size of the models in the initial population are presented in Table V.

$$
\begin{aligned}
S_1 &= [(64, 0), (0.4), (10, 3)] \\
S_2 &= [(760, 2), (0.5), (608, 2), (0.65), (10, 3)] \\
S_3 &= [(864, 0), (0.15), (536, 0), (928, 0), (10, 3)] \\
S_4 &= [(40, 0), (0.45), (912, 0), (10, 3)] \\
S_5 &= [(968, 1), (976, 1), (32, 1), (0.15), (808, 1), \\
&\quad (10, 3), (832, 2)]
\end{aligned}
$$

Model	Score (Accuracy)	Trainable Parameters	Fitness
S_1	91.7%	50890	2.7688
S_2	98.7%	1065378	3.0812
S_3	94.6%	1649506	3.3791
S_4	92.1%	77922	2.8427
S_5	96.6%	1771642	3.2867

TABLE V: Scores for the initial population found by AMS for MNIST. $\alpha = 0.5$

Observe that the sizess of the models in the initial population are diverse with some models having as few as 1 hidden layer and some having more than 5 hidden layers. The number of layers of the models in the initial population is defined by the parameter n_r, we set this value to be small on purpose, since MNIST is a dataset thats easy even for simple neural network models. Also note that the initial population exhibits models with different activation functions (with some models using sigmoid, some using tanh and some using relu), this is beneficial for the search process as some activation functions may yield better results than others.

Finally, we note that some models in the initial population already yield decent accuracy (about 90%), nevertheless they also exhibit a large number of trainable parameters. It is clear that, in the case of MNIST dataset, the task for AMS is to find a model with an accuracy higher than 95% and a small number of trainable parameters.

For a value of $\alpha = 0.5$, after 5 experiments and 10 generations for each experiment, AMS converged to the following five models. As a side note, for all of the experiments in this section we denote the best model found at experiment $i \in \{1, \cdots, n\}$ as S_i^* and the best model out of n experiments for a given α as s_i^+.

$$
\begin{aligned}
S_1^* &= [(56, 2), (10, 3)] \\
S_2^* &= [(168, 2), (10, 3)] \\
S_3^* &= [(40, 2), (0.2), (10, 3)] \\
S_4^+ &= [(48, 2), (48, 2), (10, 3)] \\
S_5^* &= [(312, 2), (10, 3)]
\end{aligned}
$$

The fitness, accuracy and raw size of the models are presented in Table VI.

Model	Score (Accuracy)	Trainable Parameters	Fitness
S_1^*	93.9%	44530	2.6287
S_2^*	95.8%	133570	2.7736
S_3^*	93.4%	31810	2.5826
S_4^+	94.7%	40522	2.5693
S_5^*	95.0%	248050	2.9431

TABLE VI: Scores for the best models found by AMS for MNIST. $\alpha = 0.5$

Table VI shows a clear preference for small models, furthermore there seems to be a preference for tanh activation functions.It can also be observed that for $\alpha = 0.5$, a good balance between size of the network and performance is obtained. In the following we perform tests with $\alpha = 0.7$ and $\alpha = 0.3$ to further analyze the behavior of the algorithm with respect to the preference of the user. A smaller α value will prioritize a better performing network while a larger value of α instructs AMS to focus on light-weight models.

The best models for each experiment with $\alpha = 0.3$, are listed next, the fitness and raw size of the models are described in Table VII.

$$S_1^* = [(32, 0), (10, 3)]$$
$$S_2^* = [(32, 1), (32, 1), (10, 3)]$$
$$S_3^+ = [(64, 1), (64, 1), (64, 1), (64, 1), (10, 3)]$$
$$S_4^* = [(56, 1), (10, 3)]$$
$$S_5^* = [(40, 1), (10, 3)]$$

Model	Score (Accuracy)	Trainable Parameters	Fitness
S_1^*	91.5%	25450	1.9126
S_2^*	95.2%	26506	1.6677
S_3^+	97.2%	63370	1.6358
S_4^*	94.7%	44530	1.7640
S_5^*	94.4%	31810	1.7412

TABLE VII: Scores for the best models found by AutoNN for MNIST, $\alpha = 0.3$.

The models presented in Table VII exhibit, in general, better performance than those depicted in Table VI. This is due to the fact that $\alpha = 0.3$ prioritizes model accuracy over model size, surprisingly, the sizes of the models obtained when $\alpha = 0.3$ are on the same order of magnitude as those obtained when $\alpha = 0.5$, a discussion on this behavior is provided later in this section.

We repeat the experiment with $\alpha = 0.7$, the obtained models are listed next and their fitness, raw size and accuracy are depicted in Table VIII.

$$S_1^* = [(24, 2), (10, 3)]$$
$$S_2^* = [(104, 2), (0.2), (10, 3)]$$
$$S_3^+ = [(16, 2), (24, 2), (10, 3)]$$
$$S_4^* = [(56, 2), (10, 3)]$$
$$S_5^* = [(208, 2), (10, 3)]$$

Model	Score (Accuracy)	Trainable Parameters	Fitness
S_1^*	92.2%	19090	3.2284
S_2^*	95.2%	82690	3.5856
S_3^+	92.1%	13218	3.1158
S_4^*	94.5%	44530	3.4200
S_5^*	95.8%	165370	3.7762

TABLE VIII: Scores for the best models found by AutoNN for MNIST, $\alpha = 0.7$.

The results in Table VIII show that when the number of trainable parameters has a large impact on the overall fitness of the individuals, the algorithm tends to prefer smaller networks, this is specially useful for cases where the computational power is limited, such as embedded systems. This brings an obvious drawback, namely that neural networks that exhibit a lower performance as compared to larger neural networks may be preferred. Nevertheless, this tradeoff can be controlled by the user by varying the α parameter.

As a side note please notice that the difference in the fitness exhibited among the models of the three different experiments is due to the fact that the size of the neural network was scaled (See section IV-B) and thus there is no fair way of comparing the fitness of the models shown in Tables VI, VII and VIII against each other.

Figure 3 shows the results obtained by all of the found models for $\alpha \in \{0.3, 0.5, 0.7\}$, which were trained for 100 epochs using 5-fold cross-validation to assess their accuracy. It is observed that models cluster around a model size less than 50000 and an error less than 0.15. As expected, models obtained with $\alpha = 0.7$ yield, in general, the smallest sizes. Outliers are mostly due to the fact that for such experiments the algorithm was unable to find a smaller model, this is likely due to a bad initial population for such experiments, nevertheless it could also be attributable to the scaling done to the network size (see Equation 5). Since AMS uses a logaritmic scale to measure the size of the networks a few thousands of weights are unlikely to make a big difference in the fitness of a model.

Finally, we compare the best models for each value of α against each other. A 10-fold cross-validation process, with a training of 50 epochs per fold, was carried out for each one of the best models in order to obtain the mean accuracy for each model. The three models were feed the same training data and validated using the same folds for the cross-validation data. We also measure the accuracy of each of the models using a test set that was never used during the training or hyper-parameter tuning processes of the models. The accuracy averages and size of the networks are summarized in Table IX.

Model	5-fold Avg. Score	Test Accuracy	Network size
$S_{0.3}^+$	97.3%	97.4%	63370
$S_{0.5}^+$	96.8%	97.0%	40522
$S_{0.7}^+$	95.0%	95.4%	13218

TABLE IX: Accuracy obtained by each of the top 3 models for MNIST dataset.

As expected, the model obtained for $\alpha = 0.7$ yields the smallest neural network (about 4 times smaller than the model obtained when $\alpha = 0.3$), nevertheless its accuracy is the worst of the three models (though by a small margin). On the opposite, the resulting model for $\alpha = 0.3$ gets the best performance in terms of accuracy but also attains the largest neural network model of the three. Finally, when $\alpha = 0.5$ AMS yields a model with a good balance between model size and performance. It is important to highligh that to obtain each of the models presented in Table IX at most 50 different models were tried. Nevertheless, since each model was trained for only 5 epochs the total time spent by the algorithm to find each of the models in Table IX was less than 4 minutes in our test rig.

Figure 2 plots the size of the model against the error yielded by it. As can be observed, the resulting models for different α sizes form a so called Pareto front [36]; i.e., none of the resulting models is better than any of the others. Furthermore, the tradeoff between size and performance of the model can be clearly seen in the figure. Although this is a nice property exhibited by MNIST dataset, this need not always hold, specially since it is known that the performance

of a neural networks does not monotonically increase with the size of the model.

Table X shows three of the top hand-crafted models along with their obtained accuracies for the MNIST dataset, it can be observed that the accuracy obtained by AMS is close to that obtained by fine tuned models.

Method	Test accuracy	Size
3 Layer NN, 300+100 hidden units [37]	96.95%	266610
2 Layer NN, 800 hidden units [38]	98.4%	636010
6-layer NN (elastic distortions) [39]	99.65%	11972510

TABLE X: Top results for MNIST dataset.

By comparing the models obtained by AMS (Table X) against the models in Table X we can observe that, eventhough AMS models don't attain the highest accuracy they exhibit good inference capabilities with a much lesser number of trainable parameters (at least one order of magnitude smaller). This shows that AMS models show a good balance between the inference power of the model and its overall size, furthermore the score-size tradeoff can be controlled by means of the α parameter, where a value closer to $\alpha = 0$ makes AMS prefer networks with higher scores and a value closer to $\alpha = 1$ makes AMS prefer networks with smaller sizes.

B. CMAPSS Dataset - A regression problem

Here we analyze the performance of AMS when dealing with regression problems. For testing regression we use the C-MAPSS dataset [40]. The C-MAPSS dataset contains simulated data produced using a model based simulation program developed by NASA. The dataset is further divided into 4 subsets composed of multi-variate temporal data obtained from 21 sensors, nevertheless for our test we will only make use of the first subset of data. Training and separate test sets are provided. The training set includes run-to-failure sensor records of multiple aero-engines collected under different operational conditions and fault modes.

The data is arranged in an $n \times 26$ matrix where n is the number of data points in each subset. The first two variables represent the engine and cycle numbers, respectively. The following three variables are operational settings which correspond to the conditions in Table XI and have a substantial effect on the engine performance. The remaining variables represent the 21 sensor readings that contain the information about the engine degradation over time.

Train trajectories	Test trajectories	Operating conditions	Fault modes
100	100	1	1

TABLE XI: CMAPSS dataset details.

Each trajectory within the training and test sets represents the life cycles of the engine. Each engine is simulated with different initial health conditions, i.e. no initial faults. For each trajectory of an engine the last data entry corresponds to the cycle at which the engine is found faulty. On the other hand, the trajectories of the test sets terminate at some point prior to failure, hence the need to predict the remaining useful life (RUL). The aim of the MLP model is to predict the RUL of each engine in the test set via regression. The actual RUL values of test trajectories are also included in the dataset for verification. Further discussions of the dataset and details on how the data is generated can be found in [41].

For this test we follow the data pre-processing described in [33], in short only 14 out of the total 21 sensors are used as the input data. Furthermore we also use a strided time-window, with window size of 24, a stride of 1 and earlyRUL of 129, to form the feature vectors for the MLP as in [33]. Further details of the time-window approach can be found in [33] and [42].

We run AMS to find a suitable MLP for regression using the CMAPSS dataset, the parameters used by the algorithm are described in Table XII.

Parameter	AutoNN Value
Problem Type	1
Architecture Type	FullyConnected
Input Shape	$(336, M)$
Output Shape	$(1, M)$
Cross Validation Ratio	$c_v = 0.2$
Mutation Ratio	$m_p = 0.4$
More Layers Probability	$n_r = 0.7$
Network Size Scaling Factor	$\alpha = 0.8$
Population Size	$n_s = 10$
Tournament Size	$n_t = 4$
Max Similar Models	$s = 3$
Training epochs	$t_e = 5$
Total Experiments	$E_{max} = 5$

TABLE XII: Parameters for CMAPSS for each method.

As with MNIST dataset we performed experiments for $\alpha \in \{0.3, \cdots, 0.7\}$ with $\Delta_\alpha = 0.1$. For the sake of space we only discuss in the detail the results obtained when $\alpha \in \{0.4, 0.5, 0.6\}$ (which were the α values that yielded the best results). The best models (best out of the five experiments) obtained for each of the α values by AMS are shown in Table XIII along with their scores (RMSE) and sizes. As with MNIST dataset, AMS partially evaluated at most 50 different models for each α value. The experiments for each α took about 2 minutes in our test rig.

$$S^+_{0.4} = [(104, 1), (824, 1), (1, 4)]$$
$$S^+_{0.5} = [(264, 2), (1, 4)]$$
$$S^+_{0.6} = [(80, 1), (80, 1), (1, 4)]$$

Model	5-fold Avg. Score	Test RMSE	Network size
$S^+_{0.4}$	14.87	14.99	122393
$S^+_{0.5}$	15.27	15.90	89233
$S^+_{0.6}$	14.78	15.74	33521

TABLE XIII: RMSE obtained by each of the top 3 models for CMAPSS dataset.

The results presented in Table XIII further demonstrate the impact of the α parameter. As can be observed the size of the networks grow as α is relaxed (smaller). It can also be observed that the results obtained by the three proposed

models in the Cross-Validation sets are very close to each other. Again, small networks are usually preferred.

Table XIV presents some of the top results obtained by hand-crafted MLPs in the CMAPSS dataset, this time the performance of the models is measured in terms of the RMSE (Root Mean Squared Error) between the real RUL and the predicted RUL.

Method	Test RMSE	Network size
Time Window MLP [43]	15.16	6041
Time Window MLP with EA [33]	14.39	7161
Deep MLP ensemble [44]	15.04	n/a

TABLE XIV: Top results for CMAPSS dataset.

The obtained models are also competitive when compared against some of the latest MLPs used with CMAPSS dataset (XIV). In this case we compare against the score obtained in the test set for each of the models. It is shown that one of the three models obtained by AMS obtain a better score than two (Time Window MLP and Deep MLP ensemble) of the three compared models. Although the sizes of the neural networks obtained by AMS are one order of magnitude bigger than the hand-crafted models we can observe that AMS delivered compact models (having few layers with few neurons in each layer).

Finally, Figure 4 shows that all of the obtained models have a small model size (few hundred thousand parameters) while all of them attained a very good performance for this dataset (See [33] and [42] for further models and their scores). Once again, it can be observed that the model size decreases with larger α values. One important observation is that, for CMAPSS dataset, there doesnt seem to be a correlation between the model size and its performance. Indeed, the model size is just one simple indicator of a networks architecture, nevertheless, trying to characterize a network only by its size may leave out valuable information about it, information that could be used in the search of new individuals with better traits for the dataset. We leave out further analysis of this behavior for future work.

VI. Conclusions and Future Work

In this paper we presented AMS, a new evolutionary algorithm for efficiently finding suitable neural network models for classification and regression problems. Making use of efficient mutation and crossover operators AMS is able to generate valid and efficient neural networks, in terms of both the size of the network and its performance. Furthermore, AMS' design is highly parallelizable and distributable, exploiting this feature with the use of frameworks such as Ray [24] or Spark [45] the performance of the algorithm can be greatly boosted.

By allowing the user to control the trade-off between the total size of the network and its performance AMS is capable of finding small neural networks when necessary (applications with limited memory, mobiles or embedded systems may have constraints on the size of the models they can handle) or prioritize the model's performance. Although not every model found by AMS is a Pareto point, the found models yield a good balance between performance and size.

Furthermore, AMS is computationally efficient since it only needs to evaluate a few tenths of models to find suitable ones. As was demonstrated in this paper, even a medium tier computing rig (consisting of a modern, medium-range processor and a modern general purpose GPU) can find good models in less than 5 minutes (depending on the dataset). This is achieved mainly to the partial train strategy, which demonstrated to be an efficient method for assesing the fitness of a given model.

Overall, AMS provides an easy to use, efficient and robust algorithm for finding suitable neural network models given a dataset. We believe that the method can be easily used by somebody who has a basic knowledge of programming, making it possible for non-experts machine learning practitioners to obtain out-of-the-box solutions.

Future work will consider more complex neural network architecturs such as LSTM and CNNs, techniques for assembling entire neural network pipelines will also be explored in future as well as the inclusion of more hyperparameters in the tuning process. An analysis of what information can be used to better characterize a neural network (and not just the model size) is also left for future work. Finally a better way of measuring distance between two neural network architectures will be explored, since this last element is of high importance for applications such as visualization and evolutionary computation.

References

[1] M. Hall, E. Frank, G. Holmes, B. Pfahringer, and I. Reutemann, P.and Witten. The weka data mining software: an update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.

[2] T. Schaul, J. Bayer, D. Wierstra, Y. Sun, M. Felder, F. Sehnke, T. Rueckstiess, and J. Schmidhuber. Pybrain. *JMLR*, 11:743–746, 2010.

[3] X. Meng, J. Bradley, B. Yavuz, B. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, m. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. Mllib: Machine learning in apache spark. *Journal of Machine Learning Research*, 17(34):1–7, 2016.

[4] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and Z. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[5] C. Francois. Keras. https://github.com/fchollet/keras, 2015.

[6] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[7] F. Seide and A. Agarwal. Cntk: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 2135–2135, New York, NY, USA, 2016. ACM.

[8] H. Jin, Q. Song, and X. Hu. Efficient neural architecture search with network morphism. *CoRR*, abs/1806.10282, 2018.

[9] E. Real, A. Aggarwal, Y. Huang, and Q. Le. Regularized evolution for image classifier architecture search. *CoRR*, abs/1802.01548, 2018.

[10] E. Sparks, A. Talwalkar, V. Smith, J. Kottalam, X. Pan, and J. Gonzales. Automated model search for large scale machine learning. In *SoCC*, pages 368–380, 2015.

[11] B. Zoph and Q. Le. Neural architecture search with reinforcement learning. *CoRR*, abs/1611.01578, 2016.

[12] B. Baker, O. Gupta, N. Naik, and R Raskar. Designing neural network architectures using reinforcement learning. *CoRR*, abs/1611.02167, 2016.

[13] Z. Zhong, J. Yan, and L. Liu. Practical network blocks design with q-learning. *CoRR*, abs/1708.05552, 2017.

[14] C. Liu, B Zoph, J. Shlens, W. Hua, L. Li, L. Fei, A. Yuille, J. Huang, and K. Murphy. Progressive neural architecture search. *CoRR*, abs/1712.00559, 2017.

[15] R. Miikkulainen, J. Liang, E. Meyerson, R. Rawal, D. Fink, O. Francon, B. Raju, A. Shahrzad, A. Navruzyan, N. Duffy, and B. Hodjat. Evolving deep neural networks. *CoRR*, abs/1703.00548, 2017.

[16] P. Angeline, G. Saunders, and J. Pollack. An evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks*, 5(1):54–65, Jan 1994.

[17] M. Suganuma, S. Shirakawa, and T. Nagao. A genetic programming approach to designing convolutional neural network architectures. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '17, pages 497–504, New York, NY, USA, 2017. ACM.

[18] C. Thornton, F. Hutter, H. Hoos, K. Leyton-Brown, and L. Kotthoff. Auto-weka 2.0: Automatic model selection and hyperparameter optimization in weka. *JMLR*, 2016.

[19] E. Brochu, V. Cora, and N. de Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *CoRR*, abs/1012.2599, 2010.

[20] F. Hutter, H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proceedings of the 5th International Conference on Learning and Intelligent Optimization*, LION'05, pages 507–523, Berlin, Heidelberg, 2011. Springer-Verlag.

[21] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter. Efficient and robust automated machine learning. In *NIPS*, volume 17, pages 1–5, 2015.

[22] H. Jin, Q. Song, and X. Hu. Auto-keras: Efficient neural architecture search with network morphism. *CoRR*, abs/1806.10282v2, 2018.

[23] E. Real, S. Moore, A. Selle, S. Saxena, Y. Suematsu, J. Tan, Q. Le, and A. Kurakin. Large-scale evolution of image classifiers. *CoRR*, abs/1703.01041v2, 2017.

[24] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, W. Paul, M. Jordan, and I. Stoica. Ray: A distributed framework for emerging AI applications. *CoRR*, abs/1712.05889, 2017.

[25] D. Engelbrecht. *Computational Intelligence. An Introduction*. Wiley, 2007.

[26] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[27] Z. Lipton. A critical review of recurrent neural networks for sequence learning. *CoRR*, abs/1506.00019, 2015.

[28] D. Engelbrecht. *Computational Intelligence: An Introduction*. Willey, 2007.

[29] R. Ebehart and Y. Shi. *Computational Intelligence*. Morgan Kauffman, 2007.

[30] S. Sumathi and P. Surekha. *Computational Intelligence Paradigms. Theory and Applications using MATLAB*. CRC Press, 2010.

[31] K. Krishnakumar. Micro-genetic algorithms for stationary and non-stationary function optimization. In *SPIE Proceedings: Intelligent Control and Adaptive Systems*, pages 289–296, 1989.

[32] Claus Hillermeier. *Nonlinear Multiobjective Optimization*. Springer, 2001.

[33] D. Laredo, Z. Chen, O. Schuetze, and JQ. Sun. A neural network-evolutionary computational framework for remaining useful life estimation of mechanical systems. Submitted to Neural Networks.

[34] J. Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, 1992.

[35] Y. LeCun and C. Cortes. MNIST handwritten digit database. -, 2010.

[36] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, New York, NY, USA, second edition, 2006.

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

[38] P. Simard, D. Steinkraus, and J. Platt. Best practices for convolutional neural networks applied to visual document analysis. In *7th International Conference on Document Analysis and Recognition (ICDAR 2003), 2-Volume Set, 3-6 August 2003, Edinburgh, Scotland, UK*, pages 958–962, 2003.

[39] D. Ciresan, U. Meier, L. Gambardella, and J. Schmidhuber. Deep big simple neural nets excel on handwritten digit recognition. *CoRR*, abs/1003.0358, 2010.

[40] A. Saxena and K. Goebel. Phm08 challenge data set. [Online] Available at: https://ti.arc.nasa.gov/tech/dash/groups/pcoe/prognostic-data-repository/.

[41] A. Saxena, K. Goebel, D. Simon, and N. Eklund. Damage propagation modeling for aircraft engine run-to-failure simulation. In IEEE, editor, *International Conference On Prognostics and Health Management*, pages 1–9, 2008.

[42] X. Li, Q. Ding, and J. Sun. Remaining useful life estimation in prognostics using deep convolution neural networks. *Reliability Engineering and System Safety*, 172:1–11, 2018.

[43] P. Lim, C. K. Goh, and K. C. Tan. A time window neural networks based framework for remaining useful life estimation. In *Proceedings International Joint Conference on Neural Networks*, pages 1746–1753, 2016.

[44] C. Zhang, P. Lim, A.K. Qin, and K.C. Tan. Multiobjective deep belief networks ensemble for remaining useful life estimation in prognostics. *IEEE Transactions on Neural Networks and Learning Systems*, 99:1–13, 2016.

[45] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, 2010.

Inidicator name	Application	Definition
Root Mean Squared Error	Regression	$x = y$
Accuracy	Classification	$x = y$
Precision	Classification	$x = y$
Recall	Classification	$x = y$
F1	Classification	$x = y$

TABLE XV: Common performance metrics for neural networks

Cell number	Cell name	Data Type	Represents	Applicable to	Values
0	Layer type	Integer	The type of layer. See table XVII	MLP/RNN/CNN	$x \in \{1, \ldots, 5\}$
1	Neuron number	Integer	Number of neurons/units in the layer	MLP/RNN	$8 * x$ where $x \in \{1, \ldots, 128\}$
2	Activation function	Integer	Type of activation function. See table XVIII	MLP/RNN/CNN	$x \in \{1, \ldots, 4\}$
3	Filter size	Integer	Number of filters generated by the layer	CNN	$8 * x$ where $x \in \{1, \ldots 64\}$
4	Kernel size	Integer	Size of the kernel used for convolutions	CNN	3^x where $x \in \{1, \ldots, 6\}$
5	Stride	Integer	Stride used for convolutions	CNN	$x \in \{1, \ldots, 6\}$
6	Pooling size	Integer	Size for the pooling operator	CNN	2^x where $x \in \{1, \ldots 6\}$
7	Dropout rate	Float	The dropout rate applied to the following layer	MLP/RNN/CNN	$x \in [0, 1]$

TABLE XVI: Details of the representation of a neural network layer as an array.

Layer type	Layer name	Can be followed by
1	Fully connected	$[1, 5]$
2	Convolutional	$[1, 2, 3, 5]$
3	Pooling	$[1, 2]$
4	Recurrent	$[1, 4]$
5	Dropout	$[1, 2, 4]$

TABLE XVII: Neural network stacking/building rules.

Index	Activation function
0	Sigmoid
1	Hyperbolic tangent
2	ReLU
3	Softmax
4	Linear

TABLE XVIII: Available activation functions.

Layer type	Neurons	Activation Function	Dropout Ratio
Fully connected	264	ReLU	n/a
Dropout	n/a	n/a	0.65
Fully Connected	464	ReLU	n/a
Dropout	n/a	n/a	0.35
Fully Connected	872	ReLU	n/a
Fully Connected	10	Softmax	n/a

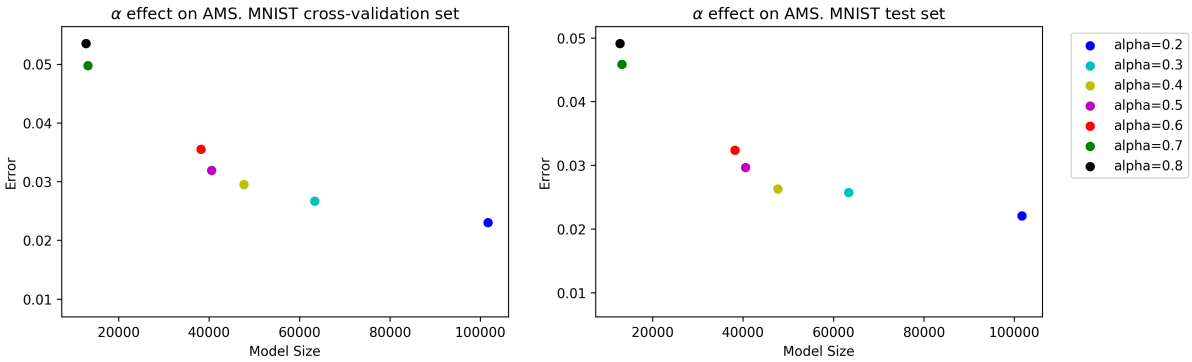
TABLE XIX: Neural network model corresponding to S_1.

Layer type	Neurons	Activation Function	Dropout Ratio
Fully connected	56	Sigmoid	n/a
Dropout	n/a	n/a	0.25
Fully Connected	360	Sigmoid	n/a
Fully Connected	480	Sigmoid	n/a
Fully Connected	80	Sigmoid	n/a
Dropout	n/a	n/a	0.20
Fully Connected	10	Softmax	n/a

TABLE XX: Neural network model corresponding to S_2.

Layer type	Neurons	Activation Function	Dropout Ratio
Fully connected	264	ReLU	n/a
Fully Connected	360	ReLU	n/a
Fully Connected	480	ReLU	n/a
Fully Connected	88	ReLU	n/a
Fully Connected	872	ReLU	n/a
Fully Connected	10	Softmax	n/a

TABLE XXI: Neural network model corresponding to S_3.



(a) Obtained models on cv set

(b) Obtained models on test set

Fig. 2: Influence of α on the model's size and error on MNIST dataset

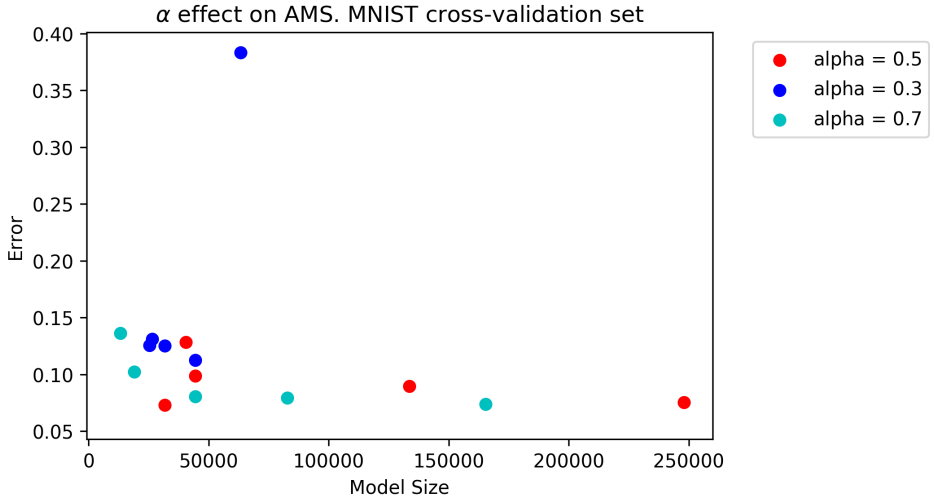
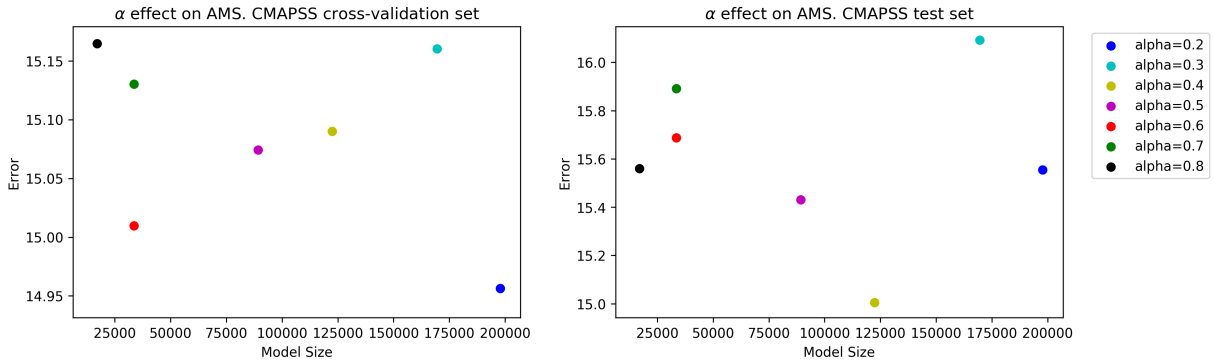


Fig. 3: Cluster formed by the found models for different α values for MNIST cross-validation set.



(a) Obtained models on cv set

(b) Obtained models on test set

Fig. 4: Influence of α on the model's size and error on CMAPSS dataset