# Automatic model selection for neural networks

David Laredo, Jian-Qiao Sun

***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 three of the major neural network architectures, namely Convolutional Neural Networks (CNNs), Multi-Layer Perceptrons (MLPs) and Recurrent Neural Networks (RNNs). This work develops AutoNN, a new micro genetic algorithm (along with a new representation for the neural network as a genotype and new crossover and mutation operator) that automatically and efficiently finds the most suitable neural network model for a problem specified by the user. Our evaluation on four different datasets show that the AutoNN 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-Weka and SkLearn.**

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

## I. Introduction

MACHINE learning studies automatic 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. The machine learning community has aided these users by making available a variety of easy to use learning algorithms and feature selection methods such as WEKA [1], PyBrain [2] or MLLib [3]. 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 neural networks have gained a lot of attention due to the newer models (CNN, RNN, Deep Learning, etc.) and their flexibility and generality for solving a large number of problems: regression, classification, natural language processing, recommendation systems, just to mention a few. Furthermore, there are a lot software libraries which makes their implementations easy to use (TensorFlow [4], Keras [5], Caffe [6], CNTK [7], etc). Nevertheless, the task of picking the right neural network model (hyperparameters included) can be even more complicated and time consuming than that of other algorithms. Given the popularity of neural networks, specially among non computer scientist we will restrict our efforts in this study to them and leave other machine learning algorithms for future work.

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.

Distributed and cloud computing provide a compelling way to accelerate this process, but also present additional challenges. Though parallel storage and processing techniques enable users to train models on massive datasets and accelerate the search process by training multiple models at once, the distributed setting forces several more decisions upon users: what parallel execution strategy to use, how big a cluster to provision, how to efficiently distribute computation across it, and what machine learning framework to use. These decisions are onerous, particularly for users who are experts in their own field but inexperienced in machine learning and distributed systems.

To address this challenges we propose AutoNN a flexible

D. Laredo and JQ Sun are with the Department of Mechanical Engineering, University of California, Merced, CA, 95340 USA e-mail: dlaredorazo@ucmerced.edu.

and scalable system to automate the process of selecting artificial neural network models. The key contributions of this paper include: 1) a new way to encode 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) defining a way of measuring the similarity between two neural networks, 4) all these components together make a new evolutionary algorithm, which we call AutoNN, which can be used to find an optimal neural network architecture for a given problem.

The remainder of this paper is organized as follows: Section II formally introduces the model selection problem (also referred as CASH problem). The related work is briefly reviewed in Section III. Next the AutoNN 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. Finally conclusions and future work are discussed in Section VI.

## II. The CASH problem

In this section we introduce and formally describe the model selection problem, our description borrows the definitions given in [8]. For the sake of simplicity for this work we only consider supervised learning problems, i.e. learning a function $f : \mathcal{X} \mapsto \mathcal{Y}$ with finite $\mathcal{Y}$ labels. A learning algorithm $A$ maps a set $\{d_1, \ldots, d_n\}$ of training data points $d_i = (\mathbf{x_i}, \mathbf{y_i}) \in \mathcal{X} \times \mathcal{Y}$ to such a function $f$. Most learning algorithms $A$ further expose hyperparameters $\lambda \in \mathbf{\Lambda}$, which change the way the learning algorithm $A_\lambda$ works. One example of hyperparameters is the number of neurons in a hidden layer of an ANN, another common example is the learning rate $\alpha$ of a neural network. These hyperparameters are typically optmized in an "outer loop" that evaluates the performance of each hyperparameter configuration using cross-validation [9].

### A. Model selection

Given a set of learning algorithms $\mathcal{A}$ and a limited amount of training data $\mathcal{D} = \{(\mathbf{x_1}, \mathbf{y_1}, \ldots \mathbf{x_n}, \mathbf{y_n})\}$, the goal of model selection is to determine the algorithm $A^* \in \mathcal{A}$ with optimal generalization performance. Generalization performance is estimated by splitting $\mathcal{D}$ into disjoint training and validation sets $\mathcal{D}_t$ and $\mathcal{D}_v$ respectively, learning function $f$ by applying $A^*$ to $\mathcal{D}_t$, and evaluating the predictive performance of this function on $\mathcal{D}_v$. Using $k$-fold validation, which splits the data into $k$ equal sized partitions $\mathcal{D}_v^1, \ldots, \mathcal{D}_v^k$ and sets $\mathcal{D}_t^i = D \backslash D_v^i$ for $i = 1, \ldots, k$ the model selection problem is written as:

$$A^* \in \underset{A \in \mathcal{A}}{\operatorname{argmin}} \frac{1}{k} \sum_{i=1}^{k} \mathcal{L}\left(A, \mathcal{D}_t^i, \mathcal{D}_v^i\right), \tag{1}$$

where $\mathcal{L}\left(A, \mathcal{D}_t^i, \mathcal{D}_v^i\right)$ is the loss achieved by $A$ when trained on $\mathcal{D}_t^i$ and evaluated on $\mathcal{D}_v^i$.

### B. Hyperparameter optimization

The problem of optimizing the hyperparameters $\lambda \in \mathbf{\Lambda}$ of a given learning algorithm $A$ is conceptually similar to that of model selection. Some key differences are that hyperparameters are often continuous, that hyperparameter spaces are often high dimensional, and that we can exploit correlation structure between different hyperparameter settings $\lambda_1, \lambda_2 \in \mathbf{\Lambda}$. Given $n$ hyperparameters $\lambda_1, \ldots, \lambda_n$ with domains $\Lambda_1, \ldots, \Lambda_n$ , the hyperparameter space $\mathbf{\Lambda}$ is a subset of the crossproduct of these domains: $\mathbf{\Lambda} \subset \Lambda_1 \times \ldots \times \Lambda_n$. This subset is often strict, such as when certain settings of one hyperparameter render other hyperparameters inactive. For example, the parameters determining the specifics of the third layer of a deep belief network are not relevant if the network depth is set to one or two. More formally, following [10], we say that a hyperparameter $\lambda_i$ is conditional on another hyperparameter $\lambda_j$, if $\lambda_i$ is only active if hyperparameter $\lambda_j$ takes values from a given set $V_i(j) \subsetneq \Lambda_j$; in this case we call $\lambda_j$ a parent of $\lambda_i$. Conditional hyperparameters can in turn be parents of other conditional hyperparameters, giving rise to a tree-structured space [11] or, in some cases, a directed acyclic graph (DAG) [10]. Given such a structured space $\mathbf{\Lambda}$, the (hierarchical) hyperparameter optimization problem can be written as:

$$\lambda^* \in \underset{\lambda \in \mathbf{\Lambda}}{\operatorname{argmin}} \frac{1}{k} \sum_{i=1}^{k} \mathcal{L}\left(A_\lambda, \mathcal{D}_t^i, \mathcal{D}_v^i\right), \tag{2}$$

In this study we consider the more general combined algorithm selection and hyperparameter optimization (CASH). That is we intend to optimize both problems at the same time. Given a set of algorithms $\mathbb{A} = \left\{A^{(1)}, \ldots, A^{(k)}\right\}$ with associated hyperparameter spaces $\mathbf{\Lambda}^{(1)}, \ldots \mathbf{\Lambda}^{(k)}$, the CASH problem is the defined as computing

$$A^*\lambda^* \in \underset{A^{(j)} \in \mathcal{A}, \lambda \in \mathbf{\Lambda}^{(j)}}{\operatorname{argmin}} \frac{1}{k} \sum_{i=1}^{k} \mathcal{L}\left(A_\lambda^{(j)}, \mathcal{D}_t^i, \mathcal{D}_v^i\right). \tag{3}$$

Note that this problem can be reformulated as a single combined hierarchical hyperparameter optimization problem with parameter space $\mathbf{\Lambda} = \mathbf{\Lambda}^{(1)} \cup \ldots \cup \mathbf{\Lambda}^{(k)} \cup \{\lambda_r\}$, where $\lambda_r$ is a new root-level hyperparameter that selects between algorithms $A^{(1)}, \ldots, A^{(k)}$. The root-level parameters of each subspace $\mathbf{\Lambda}^{(i)}$ are made conditional on $\lambda_r$ being instantiated to $A_i$.

Although the CASH problem can be solved using Bayesian Optimization [12] and Sequential Model Based Optimization (SMBO) [13] the studies presented so far are very restricted when it comes to neural networks. Instead, in this work we aim to solve the CASH problem for neural networks using evolutionary algorithms.

## III. Literature Review

Automatic model selection 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 [14].

To make machine learning accessible to non-expert users, researchers have proposed various automatic selection methods for machine learning algorithms and/or hyperparameter values for a given supervised machine learning problem. 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 automatic selection method method, 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 high labor intensive and requires a high skill set in machine learning.

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 survey of the most popular methods.

### A. AutoWEKA

Auto-WEKA [15] is a system designed to help machine learning users by automatically searching through the joint space of WEKA's learning algorithms and their respective hyperparameter settings to maximize performance using a state-of-the-art Bayesian optimization method. AutoWEKA addresses the CASH problem by treating all of WEKA as a single, highly parametric machine learning framework, and using Bayesian optimization to find a strong instantiation for a given dataset. AutoWEKA 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. AutoWEKA is tightly integrated with WEKA and does provide support for Multilayer Perceptrons (MLP).

### B. Auto-sklearn

Auto-sklearn [16] is Auto-WEKA's sister package, it uses the same Bayesian optimizer but comprises a smaller space of models and hyperparameters, however it includes additional meta-learning techniques.

### C. TuPAQ

TuPAQ [14] is a system designed to efficiently and scalably automate the process of training predictive models. One of its main features is a planning algorithm which decides on an efficient parallel execution strategy during model training while identifying new hyperparameter configurations and proactively eliminating models which are unlikely to provide good results. TuPAQ is aimed at large scale machine learning, it builds on top of the well know Apache Spark. TuPaq only focuses on classification problems and considers only three model families (Support Vector Machines, Logistic Regression and nonlinear SVMs), each with several hyperparameters. TuPAQ performs batching to train multiple models simultaneously and deploys bandit resource allocation to allocate more resources to the most promising models. TuPAQ does not provide support for neural networks.

## IV. An evolutionary algorithm for the CASH problem

While there is a number of methods for automatic model selection and hyperparameter tuning, the most popular ones do not provide support for some of the most sophisticated deep learning architectures. In the case of AutoWEKA and Auto-sklearn they do not provide good support for large machine learning problems, nor provide support for distributed computing. TuPAQ on the other hand, provides wide support for distributed computing, maximizing the use of computational resources through the use of sophisticated optimizations, nevertheless its restricted to only classification problems and does not provide support for neural networks.

We propose to implement a system for automatically selecting the most fitting neural network architecture for a given problem, whether it is classification or regression. Furthermore, that proposed system is scalable and easy to use in distributed computing environments, allowing it to be usable for large datasets and complex models. To achieve the latter we make use of Ray [17], a distributed framework designed with large scale distributed machine learning in mind.

For this work we consider three major neural networks architectures, namely multilayer percpetrons (MLPs) [18], convolutional neural networks (CNNs) [19] and recurrent neural networks (RNNs) [20]. Each one of these architectures can be built by stacking together a *valid* combination of any of the four following layers: fully connected layers, recurrent layers, convolutional layers and pooling layers.

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. AutoNN

The main part of this work consists of developing an evolutionary algorithm (EA) capable of trying and evolving different neural network architectures and, in the end, find a suitable model for the given problem 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 when searching for optimal solutions making them very suitable for applications such as the one at hand.

In the following we describe the very basics of evolutionary algorithms as an introduction for the reader.

Every evolutionary algorithm consists of a population of individuals (sometimes EAs are also referred as population based algorithms) where each individual in the population (in this case neural network model) is indeed a potential solution to the optimization problem (in this case the CASH problem). Every individual has a specific 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 does every individual in the population performs with respect to a certain metric, establishing thus an absolute order among the various solutions and a way of fairly comparing them against each other.

New generation of solutions are generated iteratively by using crossover and mutation operators. Crossover operator is an evolutionary operator used to combine the information of two parents to generate 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 [21] are one variant of GAs whose main difference is the use of small populations (less than 10 individuals per population) in contrast to some well established EAs like the genetic algorithms (GAs), evolutionary strategies (ES) and genetic programming (GP) [18]. Since computational efficiency is one of our main concerns for this work we will follow general principles of micro-GA in order to reduce the computational burden of the algorithm.

Specifically speaking and taking inspiration from the micro-GA the pseudocode for our proposed algorithm is described in Algorithm 2. Let $C_p$ and $M_p$ be the crossover and mutation probabilities respectively, let also $G_{max}$ be the maximum number of allowed generations and $E_{max}$ 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 is as follows

**Algorithm 2** Neural Network Evolution

Let $t_e = 0$ be the experiments counter
**while** $t_e < E_{max}$ **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** $t_g < G_{max}$ or nominal convergence not reached **do**
    Check for nominal convergence in $\mathcal{C}(t)$. See section IV-H
    Evaluate the fitness, $f(\mathbf{x}_i(t))$, of each individual, $\mathbf{x}_i(t)$. See section IV-B
    Identify best and worst individuals of $\mathcal{C}(t)$
    Replace worst individual in $\mathcal{C}(t)$ with best from $\mathcal{C}(t-1)$
    Perform selection. See section IV-E
    Perform crossover of individuals in $\mathcal{C}(t)$ with $C_p = 1$. Let $\mathcal{O}(t)$ be the offspring population. See section IV-F
    For each individual 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**
Final Solution is best existing solution in $\mathcal{B}$

### B. The fitness function

In order to steer the search in the most promising directions, a carefully designed fitness function is required. The algorithm's goals are to generate a neural network architecture with good inference performance for the class of problem at hand while keeping the complexity of the network as low as possible. Measuring the inference power of the network is straightforward; having a valid neural network we can assess its inference power by training it on the set $\mathcal{D}_t$ and then evaluating the predictions using the set $\mathcal{D}_v$. A more robust approach would be performing a $k$-fold cross-validation which splits the data into $k$ equal sized partitions $\mathcal{D}_v^1, \ldots, \mathcal{D}_v^k$ and sets $\mathcal{D}_t^i = D \backslash D_v^i$ for $i = 1, \ldots, k$. Note that making an accurate assessment of the inference performance of a neural network involves training it for a large number of epochs. Since the training process usually involves extensive computations our algorithm can not make use of it entirely, since this would make the entire process very computationally expensive in most cases and unfeasible in some others, instead we opt to relax the training process for our evaluation of the model and use instead a *partial train* which follows the same process as training but uses a very limited number of epochs (only a few tens of them). As stated in [] this does not affect the searching process of the micro-GA since even after training

for few epochs the models start showing a clear trend in terms of whether a model is promising or not.

Let $A \in \mathcal{A}$ be a certain neural network architecture trained on set $\mathcal{D}_t$, let also $\mathcal{P}_A(\mathcal{D}_v)$ represent the performance of the neural network $A$ when tested using validation set $\mathcal{D}_v$ and the user-defined metric $m$, where $m$ is usually any of the metrics listed in Table VII . Using $k$-fold cross-validation the average performance of the algorithm can be written as

$$p = \frac{1}{k}\sum_{i=1}^{k}\mathcal{P}_A(\mathcal{D}_v^i) \tag{4}$$

For measuring the complexity of the architecture we consider the number of trainable weights $w$ of the neural network which is a good indicator of how complex the architecture is.

Using $p$ and $w$ we propose the following fitness function

$$f = p + \alpha w, \tag{5}$$

where $\alpha \in [0, 1]$ is a scaling factor that indicates how much does the number of trainable weights $w$ affects the overall fitness of the neural network. By setting $\alpha = 1$ the preference is given to very compact architectures, while $\alpha = 0$ will only care about architectures that find the best possible value for $p$ regardless of their complexity.

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

$$p \in \begin{cases} [0, \inf], \text{if } m \text{ is } R^2 \\ [0, 1], \text{otherwise.} \end{cases} \tag{6}$$

As can be observed, even $p$ can be in different ranges, it is necessary for our application that the range of $p$ is consistent no matter the type of metric $m$, thus, in the case $m$ is the $R^2$ metric we normalize the values of $m$ to be within the range $[0, 1]$. The normalization process is quite straightforward: assuming a population $\mathcal{C}$ with $n_s$ individuals, take the maximum value $m_{max}$ for $m$ at the population, then divide every individual in $\mathcal{C}$ by $m_{max}$. Following this process $p \in [0, 1]$ for any of the metrics $m$.

Now we focus on $w$. For the sake of simplicity lets just consider the case of the MLP class of neural networks since this is usually the model where $w$ is larger. Let $l_{max}$ be the maximum number of possible layers for any model, for this work we will limit $l_{max} = 64$ since we consider this a pretty decent number for most of the mainstream deep learning models. From Table VIII we know that the maximum number of neurons at any layer is 1024, thus the maximum number of $w$, which we will call $w_{max}$ for any given model is $w_{max} = 1024^2 \times 64 = 2^{26}$. Furthermore, we want neural networks that are similar (by a few thousands of weights) to have the same score $w$, therefore we discard the last 3 digits of the neural network and replace them by 0s, therefore let $w^+$ be this new weight with the last 3 digits discarded. Let $w^* = log(w^+)$, therefore $w^*_{max} \approx log(2) * 26$, hence we assume $w^* \in [0, 7.8]$.

Thus, we redefine Equation 5 as

$$f = 5p + \alpha w^*. \tag{7}$$

As can be observed, Equation 7 is now properly scaled, and therefore it is a suitable choice as the fitness function for assessing the performance of a neural network model.

### C. Encoding neural networks as genotypes

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 proposed 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 [18] 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 new list-based encoding, that is, the genotype is 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 given layer as described in Table VIII. A visual depiction of the array is presented in Figure 1.

| Layer type | Neuron number | 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.

Let us illustrate the proposed encoding with an example, let $S_e$ be a model composed 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 model just presented is described in Table I.

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 verify the previous layer and the

| 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 I: Neural network model.

layer ahead to check for compatibility. The rules for stacking layers together are described in Table IX. As can be observed, 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.

### D. Generating valid models

Generating valid models is straightforward. An initial layer type has to be specified by the user, the initial layer type can be FullyConnected, Convolutional or Recurrent. As it can be seen, 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 defined in advance, the final/output layer is also defined in advance, 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 IX. A user defined parameter $m_l$ 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, if $n_r < m_l$ and if the current layer is compatible with the last layer (according to Table IX) then no more layers are inserted. 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 [18], [21]. A description of the mechanism is given next:

- Select $n_p$ parents at random where $n_p < n_s$.
- 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 $n_p$ 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 $n_p$ small. Also, recall from Algorithm 2 that our approach uses elitism, therefore the best individual of a current generation goes 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 our encoding. In this section we describe in detail the used crossover operator. Our operator is based on the two point crossover operator for genetic algorithms [22] in the sense that two points are selected for each parent, nevertheless our operator is more restrictive in order to ensure the generation of valid architectures. The selection mechanism is described in Algorithm 3. The following algorithm will be executed for $n_s$ times, where $n_s$ is a user defined parameter, at most or until a valid offspring is generated. Nevertheless, based on our experience with the algorithm it usually takes only 1 attempt to successfully generate a valid offspring. Finally we would like to note that although this is the implementation we used, it may not be the only one to achieve the expected results.

**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 compatible 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 $r_4 - r_3 < l_{max}$
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 $S_1$

In Algorithm 3 when we mean compatibility between two points we mean that such two points can be interchanged and still comply with the building rules stated in Table IX. Let us illustrate Algorithm 3 with an example. Consider the following models

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

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

Lets take $r_1 = 1$ and $r_2 = 3$, since these points are going to be removed from the model then 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 IX. 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:

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

Now assume we pick at random the pair $(2, 4)$, thus the offspring which we will for simplicity 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 indeed a valid model, the reader can check the actual model representations for each of the models in this example in Tables XI to XIII. Notice though that all the activation functions of the same layer types are changed to match the activation functions of the first parent $S_1$, this is what we call activation function rectification, which basically means 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 $S_1$.

We would like to highlight one important feature of this crossover operator, namely that it has the ability to generate neural network models of different sizes, i.e. it can shrink or increase the size of one of the parents. This is a desirable behavior as in real life, machine learning experts will often try various sizes of neural network models when trying to find the one that has the best inference capabilities.

### G. Mutation operator

The mutation operator is used to induce small changes to some of the models generated through the crossover mechanism. In the realm 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 [21] 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 disruptive in order to mitigate its effect. Following the same ideas found on the literature we developed a mutation operator to handle neural network models.

As stated above our mutation approach is less disruptive than the common mutation operators [18], this decision 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. 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 operations:

- Change a parameter of the layer chosen for a value complying the values stated in Table VIII.
- 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.

### H. Determining nominal convergence

Nominal convergence is one of the criteria used for early stopping of the evolutionary procedure of our algorithm. Some literature defines the convergence in terms of the fitness of the individuals [18], while in [21] 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, this follows the following reasoning.

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 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 and compute the similarities between the different individuals.

To compute the similarities between different individuals we take the following approach. Let $S_1, S_2$ where $len(S_2) >= len(S_1)$ be the genotype representing two different models, let also $S_1 - S_2$ represent the layer-wise difference between each model and $S_i[j]$ be the array representation of the $j - th$ layer of model $i$, $S_2 - S_1$ is defined in Algorithm 4.

This method is computationally inexpensive since the size of the population is small. Furthermore, it helps accurately establish the similarity between any two neural networks,

**Algorithm 4** $S_2 - S_1$ Method

Let $d \in \mathbb{R}$ be the distance between the two models. Make $d = 0$
**for** Each layer $i$ in $S_1$ except last layer **do**
  $d = d + norm_2(S_2[i] - S_1[i])$
**end for**
**for** Each remaining layer $i$ in $S_2$ except last layer **do**
  $d = d + norm_2(S_2[i])$
**end for**
Return $d$

given two neural network models $S_1$ and $S_2$ is $S_1 = S_2$ then $S_2 - S_1 = 0$. For our purposes we stop the algorithm and launch the new experiment if the distance between any $s$ models is smaller than $c$ where both $s$ and $c$ are user defined parameters.

### I. Implementation

AutoNN 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. The models $S_i$ generated by the algorithm are fetched to Keras [5] and then evaluated, nevertheless the models can be 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 we make use of Ray [17] which is a distributed computing framework tailored for AI applications. In order to distribute workloads in Ray developers only have to define Remote Functions make 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 of the neural network model $f$ there is little chance that the data interchanged causes a bottleneck or increases latency in the system.

## V. Evaluation

We evaluate AutoNN with two different datasets, each of which is a different type of inference problem. Our results are compared against baseline implementations and state-of-the-art methods for automatic model selection. In particular we compare our results against AutoWeka [15], PyBrain [2] and MLLib [3].

### A. MNIST Dataset

We first test our algorithm with the MNIST Dataset [23]. 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.

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 II

| Parameter | AutoNN Value |
|---|---|
| Problem Type | 1 |
| Architecture Type | FullyConnected |
| Input Shape | $(784, M)$ |
| Output Shape | $(1, M)$ |
| Cross Validation Ratio | $c_v = 0.2$ |
| Mutation Ratio | $m_p = 0.4$ |
| More Layers Probability | $n_r = 0.8$ |
| Network Size Scaling Factor | $\alpha = 0.8$ |
| Population Size | $n_s = 10$ |
| Tournament Size | $n_t = 4$ |
| Max Similar Models | $s = 3$ |
| Total Experiments | $E_{max} = 5$ |

TABLE II: Parameters for MNIST for each method.

We first take a look at the generated initial population. Although we understand that for the sake of space we will only discuss the sizes of the models, furthermore we will make a small change in our notation for describing the 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.

$$S_1 = [(968, 0), (688, 0), (10, 3)]$$
$$S_2 = [(824, 2), (0.4), (168, 2), (808, 2), (528, 2), (10, 3)]$$
$$S_3 = [(136, 1), (984, 1), (144, 1), (680, 1), (10, 3)]$$
$$S_4 = [(712, 2), (0.3), (96, 2), (368, 2), (584, 2), (240, 2)$$
$$(256, 2), (0.65), (384, 2), (536, 2), (648, 2), (0.3), (88, 2), (10, 3)]$$
$$S_5 = [(448, 0), (0.3), (888, 0), (10, 3)]$$

First thing we can observer is that the sizes of the models are variable, with some models having as few as 1 hidden layer and some having up to 10 hidden layers, another thing

to note is the variability in terms of activation functions with some models using sigmoid, some using tanh and some using relu as activation functions.

In this particular case, after 5 generations, the population converged to $S_1^* = [(136, 1), (168, 1), (10, 3)]$, where * denotes the best individual in the population and the subindex denotes the experiment number.

After running the 5 experiments, we get the following models

$$S_1^* = [(136, 1), (168, 1), (10, 3)]$$
$$S_2^* = [(80, 1), (10, 3)]$$
$$S_3^* = [(152, 2), (152, 2), (0.35), (10, 3)]$$
$$S_4^* = [(248, 2), (10, 3)]$$
$$S_5^* = [(208, 1), (10, 3)]$$

Their fitness and raw size of the models are described in Table III. The best overall model is $S^* = S_2^*$.

| Model | Score (Accuracy) | Trainable Parameters | Fitness |
|---|---|---|---|
| $S_1^*$ | 0.959 | 131466 | 4.4963 |
| $S_2^*$ | 0.944 | 63610 | 4.3982 |
| $S_3^*$ | 0.962 | 144106 | 4.4991 |
| $S_4^*$ | 0.962 | 197170 | 4.6105 |
| $S_5^*$ | 0.956 | 165370 | 4.6114 |

TABLE III: Scores for the best models found by AutoNN for MNIST. $\alpha = 0.8$

As can be observed in Table III small models are preferred, regardless if some larger models yield better performance, this is due to the value of the size weighting factor $\alpha = 0.8$ which is still a pretty high value, thus giving a lot of weight to the actual size of the neural network in terms of the fitness of a model. Let us illustrate this by running the algorithm with $\alpha = 1$ and $\alpha = 0.6$.

The best models for each experiment with $\alpha = 0.6$, are listed next, their fitness and raw size of the models are described in Table IV. The best overall model is $S^* = S_2^*$.

$$S_1^* = [(72, 2), (0.1), (10, 3)]$$
$$S_2^* = [(64, 2), (64, 2), (56, 2), (56, 2)(10, 3)]$$
$$S_3^* = [(368, 1), (10, 3)]$$
$$S_4^* = [(120, 2), (10, 3)]$$
$$S_5^* = [(248, 2), (96, 2), (10, 3)]$$

| Model | Score (Accuracy) | Trainable Parameters | Fitness |
|---|---|---|---|
| $S_1^*$ | 0.947 | 57250 | 2.4306 |
| $S_2^*$ | 0.953 | 61802 | 2.3869 |
| $S_3^*$ | 0.957 | 292570 | 2.6075 |
| $S_4^*$ | 0.956 | 95410 | 2.4294 |
| $S_5^*$ | 0.968 | 219554 | 2.4519 |

TABLE IV: Scores for the best models found by AutoNN for MNIST, $\alpha = 0.6$.

Next we repeat the experiment with $\alpha = 1$, again the obtained models are listed next and their fitness and raw size of the models are described in Table V. The best overall model is $S^* = S_3^*$.

$$S_1^* = [(48, 1), (48, 1), (10, 3)]$$
$$S_2^* = [(64, 2), (10, 3)]$$
$$S_3^* = [(24, 1), (10, 3)]$$
$$S_4^* = [(72, 2), (10, 3)]$$
$$S_5^* = [(120, 1), (10, 3)]$$

| Model | Score (Accuracy) | Trainable Parameters | Fitness |
|---|---|---|---|
| $S_1^*$ | 0.917 | 40522 | 5.1869 |
| $S_2^*$ | 0.943 | 50890 | 5.2734 |
| $S_3^*$ | 0.917 | 19090 | 5.1037 |
| $S_4^*$ | 0.943 | 57250 | 5.316 |
| $S_5^*$ | 0.949 | 95410 | 5.479 |

TABLE V: Scores for the best models found by AutoNN for MNIST, $\alpha = 1$.

As can be observed by the obtained results in Table V, when the number of trainable parameters has a high weight on the overall fitness of the individuals, the algorithm tends to prefer smaller neural networks, this is specially useful for cases where the computational power is limited, such as embedded systems. Nevertheless, this brings an obvious tradeoff, since more compact neural networks are preferred the algorithm may also choose neural networks that exhibit lower performance compared to larger neural networks. This tradeoff can be controlled by the user by varying the $\alpha$ parameter, as can be observed by the results in Table IV, when the impact of the size of the neural network is relaxed on the overall fitness of the individual, the algorithm may prefer bigger neural networks (whether in terms of layers or neurons in each layer) that provide better accuracy. Note that the change in fitness is due to the fact that the size of the neural network was scaled and thus there is no fair way of comparing the results of Table III against those in Table IV and so on. Nevertheless, it is also important to note that the accuracy of each of the models presented is already acceptable.

Finally, we train the 3 better models for 50 epochs each and using 10-fold cross-validation in order to obtain the mean accuracy of each of the models, we denote $S_1^*$ as the best model corresponding to $\alpha = 0.8$, $S_2^*$ corresponds to $\alpha = 0.6$ and $S_3^*$ corresponds to $\alpha = 1$. The accuracy results for the 10-fold cross-validation sets along with the accuracy obtained for the test set are summarized in Table VI.

| Model | 10-fold Min. | 10-fold Max. | 10-fold Avg. | Test Avg. |
|---|---|---|---|---|
| $S_1^*$ | 0.968 | 0.975 | 0.972 | 0.974 |
| $S_2^*$ | 0.970 | 0.976 | 0.974 | 0.975 |
| $S_3^*$ | 0.950 | 0.959 | 0.954 | 0.957 |

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

As predicted, model $S_2^*$ gets better overall performance (though marginal) because the $\alpha$ penalty for the size of the neural network was relaxed, on the contrary, model $S_3^*$ gets worst performance since the search was constrained to look for more compact models, finally $S_1^*$ lies in between obtaining a good tradeoff between inference power and compactness.

## VI. Conclusions and Future Work

In this paper we presented AutoNN, a new evolutionary algorithm for efficiently finding suitable neural network models for both classification and regression problems. Making use of efficient mutation and crossover operators the AutoNN is able to generate valid and efficient neural networks, in terms of both the size of the network and its performance. Furthermore, AutoNN is intrinsically distributable, exploiting this feature with the use of Ray [17] we were able to find efficient models for the MNIST dataset [23].

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. 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. Rucksties, 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] C. Thornton, F. Hutter, H. Hoos, and K. Leyton-Brown. Auto-weka: Combined selection and hyperparameter optimization of classification algorithms. In *KDD*, 2013.

[9] E. Alpaydin. *Introduction to Machine Learning*. The MIT Press, 2nd edition, 2010.

[10] F. Hutter, H. Hoos, K. Leyton-Brown, and T. Stutzle. Paramils: and automatic algorithm configuration framework. *JAIR*, 36(1):267–306, 2009.

[11] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kegl. Algorithms for hyper-parameter optimization. In *NIPS*, 2011.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

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

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

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

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

| Metric name | Definition |
|---|---|
| Root Mean Squared Error | Regression |
| Accuracy | Classification |
| Precision | Classification |
| Recall | Classification |
| F1 | Classification |

TABLE VII: 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 IX | 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 X | 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 VIII: 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 IX: Neural network stacking/building rules.

| Index | Activation function |
|---|---|
| 0 | Sigmoid |
| 1 | Hyperbolic tangent |
| 2 | ReLU |

TABLE X: 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 XI: 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 XII: 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 XIII: Neural network model corresponding to $S_3$.