J. Hlaváčová (Ed.): ITAT 2017 Proceedings, pp. 159–166
CEUR Workshop Proceedings Vol. 1885, ISSN 1613-0073, © 2017 P. Vidnerová, R. Neruda

Evolution Strategies for Deep Neural Network Models Design

Petra Vidnerová, Roman Neruda

Institute of Computer Science, The Czech Academy of Sciences
petra@cs.cas.cz

Abstract: Deep neural networks have become the state-of-art methods in many fields of machine learning recently. Still, there is no easy way how to choose a network architecture which can significantly influence the network performance.

This work is a step towards an automatic architecture design. We propose an algorithm for an optimization of a network architecture based on evolution strategies. The algorithm is inspired by and designed directly for the Keras library [3] which is one of the most common implementations of deep neural networks.

The proposed algorithm is tested on MNIST data set and the prediction of air pollution based on sensor measurements, and it is compared to several fixed architectures and support vector regression.

1 Introduction

Deep neural networks (DNN) have become the state-of-art methods in many fields of machine learning in recent years. They have been applied to various problems, including image recognition, speech recognition, and natural language processing [8, 10].

Deep neural networks are feed-forward neural networks with multiple hidden layers between the input and output layer. The layers typically have different units depending on the task at hand. Among the units, there are traditional perceptrons, where each unit (neuron) realizes a nonlinear function, such as the *sigmoid* function, or the rectified linear unit (*ReLU*).

While the learning of weights of the deep neural network is done by algorithms based on the stochastic gradient descent, the choice of architecture, including a number and sizes of layers, and a type of activation function, is done manually by the user. However, the choice of architecture has an important impact on the performance of the DNN. Some kind of expertise is needed, and usually a trial and error method is used in practice.

In this work we exploit a fully automatic design of deep neural networks. We investigate the use of evolution strategies for evolution of a DNN architecture. There are not many studies on evolution of DNN since such approach has very high computational requirements. To keep the search space as small as possible, we simplify our model focusing on implementation of DNN in the Keras library [3] that is a widely used tool for practical applications of DNNs.

The proposed algorithm is evaluated both on benchmark and real-life data sets. As the benchmark data we use the MNIST data set that is classification of handwritten digits. The real data set is from the area of sensor networks for air pollution monitoring. The data came from De Vito et al [21, 5] and are described in detail in Section 5.1.

The paper is organized as follows. Section 2 brings an overview of related work. Section 3 briefly describes the main ideas of our approach. In Section 4 our algorithm based on evolution strategies is described. Section 5 summarizes the results of our experiments. Finally, Section 6 brings conclusion.

2 Related Work

Neuroevolution techniques have been applied successfully for various machine learning problems [6]. In classical neuroevolution, no gradient descent is involved, both architecture and weights undergo the evolutionary process. However, because of large computational requirements the applications are limited to small networks.

There were quite many attempts on architecture optimization via evolutionary process (e.g. [19, 1]) in previous decades. Successful evolutionary techniques evolving the structure of feed-forward and recurrent neural networks include NEAT [18], HyperNEAT [17] and CoSyNE [7] algorithms.

On the other hand, studies dealing with evolution of deep neural networks and convolutional networks started to emerge only very recently. The training of one DNN usually requires hours or days of computing time, quite often utilizing GPU processors for speedup. Naturally, the evolutionary techniques requiring thousands of training trials were not considered a feasible choice. Nevertheless, there are several approaches to reduce the overall complexity of neuroevolution for DNN. Still due to limited computational resources, the studies usually focus only on parts of network design.

For example, in [12] CMA-ES is used to optimize hyperparameters of DNNs. In [9] the unsupervised convolutional networks for vision-based reinforcement learning are studied, the structure of CNN is held fixed and only a small recurrent controller is evolved. However, the recent paper [16] presents a simple distributed evolutionary strategy that is used to train relatively large recurrent network with competitive results on reinforcement learning tasks.

In [14] automated method for optimizing deep learning architectures through evolution is proposed, extending existing neuroevolution methods. Authors of [4] sketch a genetic approach for evolving a deep autoencoder network enhancing the sparsity of the synapses by means of special operators. Finally, the paper [13] presents two version of an evolutionary and co-evolutionary algorithm for design of DNN with various transfer functions.

3 Our Approach

In our approach we use evolution strategies to search for optimal architecture of DNN, while the weights are learned by gradient based technique.

The main idea of our approach is to keep the search space as small as possible, therefore the architecture specification is simplified. It directly follows the implementation of DNN in Keras library, where networks are defined layer by layer, each layer fully connected with the next layer. A layer is specified by number of neurons, type of an activation function (all neurons in one layer have the same type of an activation function), and type of regularization (such as dropout).

In this paper, we work only with fully connected feedforward neural networks, but the approach can be further modified to include also convolutional layers. Then the architecture specification would also contain type of layer (dense or convolutional) and in case of convolutional layer size of the filter.

4 Evolution Strategies for DNN Design

Evolution strategies (ES) were proposed for work with real-valued vectors representing parameters of complex optimization problems [2]. In the illustration algorithm bellow we can see a simple ES working with n individuals in a population and generating m offspring by means of Gaussian mutation. The environmental selection has two traditional forms for evolution strategies. The so called $(n+m)$-ES generates new generation by deterministically choosing n best individuals from the set of $(n+m)$ parents and offspring. The so called (n,m)-ES generates new generation by selecting from m new offspring (typically, $m > n$). The latter approach is considered more robust against local optima premature convergence.

Currently used evolution strategies may carry more meta-parameters of the problem in the individual than just a vector of mutation variances. A successful version of evolution strategies, the so-called *covariance matrix adaptation ES* (CMA-ES) [12] uses a clever strategy to approximate the full $N \times N$ covariance matrix, thus representing a general N-dimensional normal distribution. Crossover operator is usually used within evolution strategies.

In our implementation (n,m)-ES (see Alg. 1) is used. Offspring are generated using both mutation and crossover operators. Since our individuals are describing network topology, they are not vectors of real numbers. So our operators slightly differ from classical ES. The more detail description follows.

Algorithm 1 (n,m)-Evolution strategy optimizing real-valued vector and utilizing adaptive variance for each parameter

```
procedure $(n,m)$-ES
    $t \leftarrow 0$
    Initialize population $P_t$ $n$ by randomly generated vectors $\vec{x^t} = (x_1^t, \dots, x_N^t, \sigma_1^t, \dots, \sigma_N^t)$
    Evaluate individuals in $P_t$
    while not terminating criterion do
        for $i \leftarrow 1, \dots, m$ do
            choose randomly a parent $\vec{x_i^t}$,
            generate an offspring $\vec{y_i^t}$
            by Gaussian mutation:
            for $j \leftarrow 1, \dots, N$ do
                $\sigma_j' \leftarrow \sigma_j \cdot (1 + \alpha \cdot N(0,1))$
                $x_j' \leftarrow x_j + \sigma_j' \cdot N(0,1)$
            end for
            insert $\vec{y_i^t}$ to offspring candidate population $P_t'$
        end for
        Deterministically choose $P_{t+1}$ as $n$ best individuals from $P_t'$
        Discard $P_t$ and $P_t'$
        $t \leftarrow t + 1$
    end while
end procedure
```

4.1 Individuals

Individuals are coding feed-forward neural networks implemented as Keras model *Sequential*. The model implemented as *Sequential* is built layer by layer, similarly an individual consists of blocks representing individual layers.

$$\begin{aligned} I = (\quad & [size_1, drop_1, act_1, \sigma_1^{size}, \sigma_1^{drop}]_1, \dots, \\ & [size_H, drop_H, act_H, \sigma_H^{size}, \sigma_H^{drop}]_H \quad), \end{aligned}$$

where H is the number of hidden layers, $size_i$ is the number of neurons in corresponding layer that is dense (fully connected) layer, $drop_i$ is the dropout rate (zero value represents no dropout), $act_i \in \{\texttt{relu}, \texttt{tanh}, \texttt{sigmoid}, \texttt{hardsigmoid}, \texttt{linear}\}$ stands for activation function, and σ_i^{size} and σ_i^{drop} are strategy coefficients corresponding to size and dropout.

So far, we work only with dense layers, but the individual can be further generalized to work with convolutional layers as well. Also other types of regularization can be considered, we are limited to dropout for the first experiments.

4.2 Crossover

The operator *crossover* combines two parent individuals and produces two offspring individuals. It is implemented as one-point crossover, where the cross-point is on the border of a block.

Let two parents be

$$I_{p1} = (B_1^{p1}, B_2^{p1}, \dots, B_k^{p1})$$
$$I_{p2} = (B_1^{p2}, B_2^{p2}, \dots, B_l^{p2}),$$

then the crossover produces offspring

$$I_{o1} = (B_1^{p1}, \dots, B_{cp1}^{p1}, B_{cp2+1}^{p2}, \dots, B_l^{p2})$$
$$I_{o1} = (B_1^{p2}, \dots, B_{cp2}^{p2}, B_{cp1+1}^{p1}, \dots, B_k^{p1}),$$

where $cp_1 \in \{1, \dots, k-1\}$ and $cp_2 \in \{1, \dots, l-1\}$.

4.3 Mutation

The operator *mutation* brings random changes to an individual. Each time an individual is mutated, one of the following mutation operators is randomly chosen:

- mutateLayer - introduces random changes to one randomly selected layer. One of the following operators is randomly chosen:
  - changeLayerSize - the number of neurons is changed. Gaussian mutation is used, adapting strategy parameters σ^{size}, the final number is rounded (since size has to be integer).
  - changeDropOut - the dropout rate is changed using Gaussian mutation adapting strategy parameters σ^{drop}.
  - changeActivation - the activation function is changed, randomly chosen from the list of available activations.
- addLayer - one randomly generated block is inserted at random position.
- delLayer - one randomly selected block is deleted.

Note, that the ES like mutation comes in play only when size of layer or dropout parameter is changed. Otherwise the strategy parameters are ignored.

4.4 Fitness

Fitness function should reflect a quality of the network represented by an individual. To assess the generalization ability of the network represented by the individual we use a crossvalidation error. The lower the crossvalidation error, the higher the fitness of the individual.

Classical k-fold crossvalidation is used, i.e. the training set is split into k-folds and each time one fold is used for testing and the rest for training. The mean error on the testing set over k run is evaluated.

The mean squared error is used as an error function:

$$E = 100 \frac{1}{N} \sum_{t=1}^{N} (f(x^t) - y^t)^2,$$

where $T = (x_1, y_1), \dots, (x_N, y_N)$ is the actual testing set and f is the function represented by the learned network.

4.5 Selection

The tournament selection is used, i.e. each turn of the tournament k individuals are selected at random and the one with the highest fitness, in our case the one with the lowest crossvalidation error, is selected.

Our implementation of the proposed algorithm is available at [20].

5 Experiments

5.1 Data Set

For the first experiment we used real-world data from the application area of sensor networks for air pollution monitoring [21, 5], for the second experiment the well known MNIST data set [11].

The sensor data contain tens of thousands measurements of gas multi-sensor MOX array devices recording concentrations of several gas pollutants collocated with a conventional air pollution monitoring station that provides labels for the data. The data are recorded in 1 hour intervals, and there is quite a large number of gaps due to sensor malfunctions. For our experiments we have chosen data from the interval of March 10, 2004 to April 4, 2005, taking into account each hour where records with missing values were omitted. There are altogether 5 sensors as inputs and 5 target output values representing concentrations of CO, NO_2, NOx, $C6H6$, and $NMHC$.

The whole time period is divided into five intervals. Then, only one interval is used for training, the rest is utilized for testing. We considered five different choices of the training part selection. This task may be quite difficult, since the prediction is performed also in different parts of the year than the learning, e.g. the model trained on data obtained during winter may perform worse during summer (as was suggested by experts in the application area).

Table 1 brings overview of data sets sizes. All tasks have 8 input values (five sensors, temperature, absolute and relative humidity) and 1 output (predicted value). All values are normalized between $\langle 0, 1 \rangle$.

Table 1: Overview of data sets sizes.

Task	train set	test set
CO	1469	5875
NO2	1479	5914
NOx	1480	5916
C6H6	1799	7192
NMHC	178	709

The MNIST data set contains 70 000 images of hand written digits, 28×28 pixel each (see Fig. 1). 60 000 are used for training, 10 000 for testing.

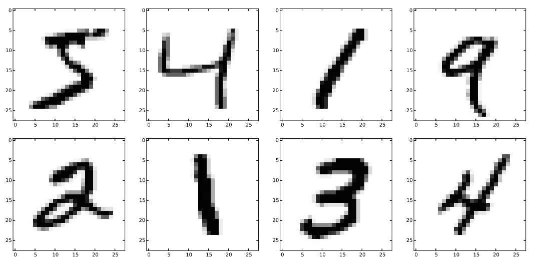

Figure 1: Example of MNIST data set samples.

5.2 Setup

For the sensor data the proposed algorithm was run for 100 generations for each data set, with $n = 10$ and $m = 30$. During fitness function evaluation the network weights are trained by RMSprop (one of the standard algorithms) for 500 epochs. Besides the ES classical GA was implemented and run on sensor data with same fitness function.

For the MNIST data set, the algorithm was run for 30 generations, with $n = 5$ and $m = 10$, for fitness evaluation the RMSprop was run for 20 epochs.

When the best individual is obtained, the corresponding network is built and trained on the whole training set and evaluated on the test set.

5.3 Results

The resulting testing errors obtained by GA and ES in the first experiment are listed in Table 3. There are average, standard deviation, minimum and maximum errors over 10 computations. The performance of ES over GA is slightly better, the ES achieved lower errors in 15 cases, GA in 11 cases.

Table 4 compares ES testing errors to results obtained by support vector regression (SVR) with linear, RBF, polynomial, and sigmoid kernel function. SVR was trained using Scikit-learn library [15], hyperparameters were found using grid search and crossvalidation.

The ES outperforms the SVR, it found best results in 17 cases.

Finally, Table 5 compares the testing error of evolved network to error of three fixed architectures (for example 30-10-1 stands for 2 hidden layers of 30 and 10 neurons, one neuron in output layers, ReLU activation is used and dropout 0.2). The evolved network achieved the most (10) best results.

Since this task does not have much training samples, also the networks evolved are quite small. The typical evolved network had one hidden layer of about 70 neurons, dropout rate 0.3 and ReLU activation function.

The second experiment was the classification of MNIST letters. As a baseline architecture was taken the one from Keras examples, i.e. network with two hidden layers of 512 ReLU units each, both with dropout 0.2. This network has a fairly good performance. It was trained 10 times and the results are listed in Table 2, together with results obtained by the evolved network.

Table 2: Test accuracies on the MNIST data set.

model	avg	std	min	max
baseline	98.34	0.13	98.18	98.55
evolved by ES	98.64	0.05	98.55	98.73

The evolved network had also two hidden layers, first with 736 ReLU units and dropout parameter 0.09, the second with 471 hard sigmoid units and dropout 0.2. The ES found a competitive result, the evolved network achieved better accuracy than the baseline model.

6 Conclusion

We have proposed an algorithm for automatic design of DNNs based on evolution strategies. The algorithm was tested in experiments on the real-life sensor data set and MNIST dataset of handwritten digits. On sensor data set, the solutions found by our algorithm outperforms SVR and selected fixed architectures. The activation function dominating in solutions is the ReLU function. For the MNIST data set, the network with ReLU and hard sigmoid units was found, outperforming the baseline solution. We have shown that our algorithm is able to found competitive solutions.

The main limitation of the algorithm is the time complexity. One direction of our future work is to try to lower the number of fitness evaluations using surrogate modeling or to use asynchronous evolution.

Also we plan to extend the algorithm to work also with convolutional networks and to include more parameters, such as other types of regularization, the type of optimization algorithm, etc.

The gradient based optimization algorithm depends significantly on the random initialization of weights. One way to overcome this is to combine the evolution of weights and gradient based local search that is another possibility of future work.

Acknowledgment

This work was partially supported by the Czech Grant Agency grant 15-18108S and institutional support of the Institute of Computer Science RVO 67985807.

Access to computing and storage facilities owned by parties and projects contributing to the National Grid Infrastructure MetaCentrum provided under the programme "Projects of Large Research, Development, and Innovations Infrastructures" (CESNET LM2015042), is greatly appreciated.

Table 3: Errors on test set for networks found by GA and ES. The average, standard deviation, minimum and maximum of 10 evaluations of the learning algorithm are listed.

	GA				**ES**			
	avg	std	min	max	avg	std	min	max
CO part1	**0.209**	0.014	0.188	0.236	0.229	0.026	0.195	0.267
CO part2	0.801	0.135	0.600	1.048	**0.657**	0.024	0.631	0.694
CO part3	0.266	0.029	0.222	0.309	**0.256**	0.045	0.199	0.349
CO part4	**0.404**	0.226	0.186	0.865	0.526	0.108	0.308	0.701
CO part5	0.246	0.024	0.207	0.286	**0.235**	0.025	0.199	0.277
NOx part1	2.201	0.131	1.994	2.506	**2.132**	0.086	2.021	2.284
NOx part2	1.705	0.284	1.239	2.282	**1.599**	0.077	1.444	1.685
NOx part3	**1.238**	0.163	0.982	1.533	1.339	0.242	1.106	1.955
NOx part4	**1.490**	0.173	1.174	1.835	1.610	0.164	1.435	2.041
NOx part5	**0.551**	0.052	0.456	0.642	0.622	0.075	0.521	0.726
NO2 part1	1.697	0.266	1.202	2.210	**1.506**	0.217	1.132	1.823
NO2 part2	2.009	0.415	1.326	2.944	**1.371**	0.048	1.242	1.415
NO2 part3	**0.593**	0.082	0.532	0.815	0.660	0.078	0.599	0.863
NO2 part4	**0.737**	0.023	0.706	0.776	0.782	0.043	0.711	0.856
NO2 part5	1.265	0.158	1.054	1.580	**0.730**	0.111	0.520	0.905
C6H6 part1	**0.013**	0.005	0.006	0.024	**0.013**	0.004	0.007	0.018
C6H6 part2	0.039	0.015	0.025	0.079	**0.034**	0.010	0.020	0.050
C6H6 part3	**0.019**	0.011	0.009	0.041	0.048	0.015	0.016	0.075
C6H6 part4	0.030	0.015	0.014	0.061	**0.020**	0.010	0.010	0.042
C6H6 part5	**0.017**	0.015	0.004	0.051	0.027	0.011	0.014	0.051
NMHC part1	1.719	0.168	1.412	2.000	**1.685**	0.256	1.448	2.378
NMHC part2	**0.623**	0.164	0.446	1.047	0.713	0.097	0.566	0.865
NMHC part3	1.144	0.181	0.912	1.472	**1.097**	0.270	0.775	1.560
NMHC part4	1.220	0.206	0.994	1.563	**1.099**	0.166	0.898	1.443
NMHC part5	1.222	0.126	1.055	1.447	**1.023**	0.050	0.963	1.116
	11				15			
	44%				60%			

Table 4: Test errors for evolved network and SVR with different kernel functions. For the evolved network the average, standard deviation, minimum and maximum of 10 evaluations of learning algorithm are listed.

Task	Evolved network				SVR			
	avg	std	min	max	linear	RBF	Poly.	Sigmoid
CO_part1	**0.229**	0.026	0.195	0.267	0.340	0.280	0.285	1.533
CO_part2	0.657	0.024	0.631	0.694	0.614	**0.412**	0.621	1.753
CO_part3	**0.256**	0.045	0.199	0.349	0.314	0.408	0.377	1.427
CO_part4	**0.526**	0.108	0.308	0.701	1.127	0.692	0.535	1.375
CO_part5	0.235	0.025	0.199	0.277	0.348	0.207	**0.198**	1.568
NOx_part1	2.132	0.086	2.021	2.284	**1.062**	1.447	1.202	2.537
NOx_part2	1.599	0.077	1.444	1.685	2.162	1.838	**1.387**	2.428
NOx_part3	1.339	0.242	1.106	1.955	**0.594**	0.674	0.665	2.705
NOx_part4	1.610	0.164	1.435	2.041	0.864	0.903	**0.778**	2.462
NOx_part5	**0.622**	0.075	0.521	0.726	1.632	0.730	1.446	2.761
NO2_part1	**1.506**	0.217	1.132	1.823	2.464	2.404	2.401	2.636
NO2_part2	**1.371**	0.048	1.242	1.415	2.118	2.250	2.409	2.648
NO2_part3	**0.660**	0.078	0.599	0.863	1.308	1.195	1.213	1.984
NO2_part4	**0.782**	0.043	0.711	0.856	1.978	2.565	1.912	2.531
NO2_part5	**0.730**	0.111	0.520	0.905	1.0773	1.047	0.967	2.129
C6H6_part1	**0.013**	0.004	0.007	0.018	0.300	0.511	0.219	1.398
C6H6_part2	**0.034**	0.010	0.020	0.050	0.378	0.489	0.369	1.478
C6H6_part3	**0.048**	0.015	0.016	0.075	0.520	0.663	0.538	1.317
C6H6_part4	**0.020**	0.010	0.010	0.042	0.217	0.459	0.123	1.279
C6H6_part5	**0.027**	0.011	0.014	0.051	0.215	0.297	0.188	1.526
NMHC_part1	1.685	0.256	1.448	2.378	1.718	1.666	**1.621**	3.861
NMHC_part2	**0.713**	0.097	0.566	0.865	0.934	0.978	0.839	3.651
NMHC_part3	**1.097**	0.270	0.775	1.560	1.580	1.280	1.438	2.830
NMHC_part4	**1.099**	0.166	0.898	1.443	1.720	1.565	1.917	2.715
NMHC_part5	1.023	0.050	0.963	1.116	1.238	**0.944**	1.407	2.960
	17				2	2	4	
	68%				8%	8%	16%	

Table 5: Test errors for evolved network and three selected fixed architectures.

Task	Evolved network		50-1		30-10-1		30-10-30-1	
	avg	std	avg	std	avg	std	avg	std
CO_part1	**0.229**	0.026	0.230	0.032	0.250	0.023	0.377	0.103
CO_part2	**0.657**	0.024	0.861	0.136	0.744	0.142	0.858	0.173
CO_part3	**0.256**	0.045	0.261	0.040	0.305	0.043	0.302	0.046
CO_part4	0.526	0.108	0.621	0.279	0.638	0.213	**0.454**	0.158
CO_part5	**0.235**	0.025	0.283	0.072	0.270	0.032	0.309	0.032
NOx_part1	2.132	0.086	2.158	0.203	**2.095**	0.131	2.307	0.196
NOx_part2	**1.599**	0.077	1.799	0.313	1.891	0.199	2.083	0.172
NOx_part3	1.339	0.242	1.077	0.125	1.092	0.178	**0.806**	0.185
NOx_part4	1.610	0.164	**1.303**	0.208	1.797	0.461	1.600	0.643
NOx_part5	**0.622**	0.075	0.644	0.075	0.677	0.055	0.778	0.054
NO2_part1	1.506	0.217	1.659	0.250	**1.368**	0.135	1.677	0.233
NO2_part2	**1.371**	0.048	1.762	0.237	1.687	0.202	1.827	0.264
NO2_part3	0.660	0.078	0.682	0.148	**0.576**	0.044	0.603	0.069
NO2_part4	0.782	0.043	1.109	0.923	**0.757**	0.059	0.802	0.076
NO2_part5	0.730	0.111	**0.646**	0.064	0.734	0.107	0.748	0.123
C6H6_part1	0.013	0.004	**0.012**	0.006	0.081	0.030	0.190	0.060
C6H6_part2	**0.034**	0.010	0.039	0.012	0.101	0.015	0.211	0.071
C6H6_part3	0.048	0.015	**0.024**	0.007	0.091	0.047	0.115	0.031
C6H6_part4	**0.020**	0.010	0.026	0.010	0.051	0.026	0.096	0.020
C6H6_part5	0.027	0.011	**0.025**	0.008	0.113	0.025	0.176	0.058
NMHC_part1	**1.685**	0.256	1.738	0.144	1.889	0.119	2.378	0.208
NMHC_part2	0.713	0.097	**0.553**	0.045	0.650	0.078	0.799	0.096
NMHC_part3	1.097	0.270	1.128	0.089	0.901	0.124	**0.789**	0.184
NMHC_part4	1.099	0.166	1.116	0.119	0.918	0.119	**0.751**	0.096
NMHC_part5	1.023	0.050	0.970	0.094	0.889	0.085	**0.856**	0.074
	10		6		4		5	
	40%		24%		16%		20%	

References

[1] Jasmina Arifovic and Ramazan Gençay. Using genetic algorithms to select architecture of a feedforward artificial neural network. *Physica A: Statistical Mechanics and its Applications*, 289(3–4):574 – 594, 2001.

[2] H.-G. Beyer and H. P. Schwefel. Evolutionary strategies: A comprehensive introduction. *Natural Computing*, pages 3–52, 2002.

[3] François Chollet. Keras. `https://github.com/fchollet/keras`, 2015.

[4] Omid E. David and Iddo Greental. Genetic algorithms for evolving deep neural networks. In *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*, GECCO Comp '14, pages 1451–1452, New York, NY, USA, 2014. ACM.

[5] S. De Vito, G. Fattoruso, M. Pardo, F. Tortorella, and G. Di Francia. Semi-supervised learning techniques in artificial olfaction: A novel approach to classification problems and drift counteraction. *Sensors Journal, IEEE*, 12(11):3215–3224, Nov 2012.

[6] Dario Floreano, Peter Dürr, and Claudio Mattiussi. Neuroevolution: from architectures to learning. *Evolutionary Intelligence*, 1(1):47–62, 2008.

[7] Faustino Gomez, Juergen Schmidhuber, and Risto Miikkulainen. Accelerated neural evolution through cooperatively coevolved synapses. *Journal of Machine Learning Research*, pages 937–965, 2008.

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[9] Jan Koutník, Juergen Schmidhuber, and Faustino Gomez. Evolving deep unsupervised convolutional networks for vision-based reinforcement learning. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, GECCO '14, pages 541–548, New York, NY, USA, 2014. ACM.

[10] Yann Lecun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 5 2015.

[11] Yann LeCun and Corinna Cortes. The mnist database of handwritten digits, 2012.

[12] Ilya Loshchilov and Frank Hutter. CMA-ES for hyperparameter optimization of deep neural networks. *CoRR*, abs/1604.07269, 2016.

[13] Tomas H. Maul, Andrzej Bargiela, Siang-Yew Chong, and Abdullahi S. Adamu. Towards evolutionary deep neural networks. In Flaminio Squazzoni, Fabio Baronio, Claudia Archetti, and Marco Castellani, editors, *ECMS 2014 Proceedings*. European Council for Modeling and Simulation, 2014.

[14] Risto Miikkulainen, Jason Zhi Liang, Elliot Meyerson, Aditya Rawal, Dan Fink, Olivier Francon, Bala Raju, Hormoz Shahrzad, Arshak Navruzyan, Nigel Duffy, and Babak Hodjat. Evolving deep neural networks. *CoRR*, abs/1703.00548, 2017.

[15] F. Pedregosa et al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[16] T. Salimans, J. Ho, X. Chen, and I. Sutskever. Evolution Strategies as a Scalable Alternative to Reinforcement Learning. *ArXiv e-prints*, March 2017.

[17] Kenneth O. Stanley, David B. D'Ambrosio, and Jason Gauci. A hypercube-based encoding for evolving large-scale neural networks. *Artif. Life*, 15(2):185–212, April 2009.

[18] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

[19] B. u. Islam, Z. Baharudin, M. Q. Raza, and P. Nallagownden. Optimization of neural network architecture using genetic algorithm for load forecasting. In *2014 5th International Conference on Intelligent and Advanced Systems (ICIAS)*, pages 1–6, June 2014.

[20] Petra Vidnerová. GAKeras. `github.com/PetraVidnerova/GAKeras`, 2017.

[21] S. De Vito, E. Massera, M. Piga, L. Martinotto, and G. Di Francia. On field calibration of an electronic nose for benzene estimation in an urban pollution monitoring scenario. *Sensors and Actuators B: Chemical*, 129(2):750 – 757, 2008.