S. Krajči (ed.): ITAT 2018 Proceedings, pp. 80–85
CEUR Workshop Proceedings Vol. 2203, ISSN 1613-0073, © 2018 Petra Vidnerová and Roman Neruda

Asynchronous Evolution of Convolutional Networks

Petra Vidnerová, Roman Neruda

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

Abstract: Due to many successful practical applications, deep neural networks and convolutional networks have become the state-of-art machine learning methods recently. The choice of network architecture for the task at hand is typically made by trial and error.

This work deals with an automatic data-dependent architecture design. We propose an algorithm for optimization of architecture of convolutional network based on asynchronous evolution. The algorithm is inspired by and designed directly for the Keras library which is one of the most common implementations of deep neural networks. The proposed algorithm is successfully tested on MNIST and Fashion-MNIST data sets.

1 Introduction

Various architectures of 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 [7, 11].

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*).

Convolutional networks (CNN) are family of DNN. They typically have three types of layers – convolutional layers, max-pooling layers, and dense (i.e. fully-connected) layers. For the most common case of image processing, convolutional layers perform convolution of an input image to abstract high-level features. They are defined by a set of learnable filters. Max-pooling layers reduce the size of representation, their function is fixed and they are not learnable. Dense layers are usually used as the last layers of the network to perform the actual classification.

While the learning of weights (including filters) of the CNN is done by algorithms based on the stochastic gradient descent, the choice of architecture, including a number and sizes of layers, number and size of convolutional filters, size of pools in pooling-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 CNN. 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 CNNs. We investigate the use of evolutionary algorithms for evolution of a CNN architecture. There are not many studies on evolution of CNN 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 CNN in the Keras library [3] that is a widely used tool for practical applications of DNNs and CNNs.

The approach described in this paper extends our previous results for evolving DNNs limited to networks with dense layers only [25, 24]. The proposed algorithm is evaluated on the MNIST and Fashion-MNIST data sets that are both classification tasks of small gray-scale images.

The paper is organized as follows. Next Section reviews related work. Section 3 describes the main ideas of our approach. Section 4 explains the main ideas of asynchronous evolution. Section 5 summarises the results of our experiments, and finally Section 6 brings conclusion.

2 Related Work

Neuroevolution represents an attempt to train a neural network by means of evolutionary techniques [5]. In traditional neuroevolution, no gradient descent is usually involved, and both architecture and weights of the network 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. [22, 1]) in previous decades. Successful evolutionary techniques evolving the structure of feed-forward and recurrent neural networks include NEAT [20], HyperNEAT [19] and CoSyNE [6] 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 [14] CMA-ES is used to optimize hyperparameters of DNNs. In [10] 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 [17] presents a simple distributed evolutionary strategy that is used to train relatively large recurrent network with competitive results on reinforcement learning tasks.

In [16] 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. The paper [15] presents two versions of an evolutionary and co-evolutionary algorithm for design of DNN with various transfer functions. Finally, [21] proposes genetic programming to evolve CNNs.

3 Our Approach

In our approach we use asynchronous evolution to search for optimal architecture of CNN, 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 CNN in Keras library, where networks are defined layer by layer. Layer is specified by its type – convolutional, max-pooling, dense. Dense layers are defined by a number of neurons, type of an activation function (all neurons in one layer have the same type of their activation function), and the type of regularization (such as dropout). Convolutional layers are defined by number of filters, size of the filter, and possibly the type of an activation function and type of regularization. Max-pooling layers are defined by the size of pool.

In this paper, we limit to networks that can be split into two parts.The first part is a preprocessing part, it contains only convolutional and max-pooling layers, and it is responsible for the preprocessing of the input and abstracting high-level features. The second part is a classifier and it consists of dense layers. Such architecture corresponds to the original proposal of the LeNet architecture [12] (Fig. 1).

3.1 Individuals

In order to apply genetic algorithm (GA) to the search for an optimal CNN architecture, we have to be able to encode the architecture by an individual of the GA.

Our proposal of encoding closely follows the CNN description and implementation in the Keras [3] model *Sequential*. The model implemented as *Sequential* is built layer by layer, similarly the GA individual consists of blocks representing individual layers.

$$I = (I_1, I_2),$$
$$I_1 = ([type, params]_1, \ldots, [type, params]_{H1})$$
$$I_2 = ([size, dropout, act]_1, \ldots, [size, dropout, act]_{H2})$$

where I_1 and I_2 are the convolutional and dense part, respectively, $H1, H2$ is the number of layers in convolutional and dense part, respectively. The blocks in convolutional part encode $type \in \{\texttt{convolutional}, \texttt{max}-\texttt{pooling}\}$ type of layer and *params* other parameters of the layer (for convolutional layer it is number of filters, size of filter, and activation function; for max-pooling layer it is only size of pool). The blocks in dense part code dense layers, so they consist of *size* the number of neurons, *drop* the dropout rate (zero value represents no dropout), $act \in \{\texttt{relu}, \texttt{tanh}, \texttt{sigmoid}, \texttt{hardsigmoid}, \texttt{linear}\}$ activation function.

3.2 Genetic Operators

To produce new individuals in genetic algorithm we use recombination operators *crossover* and *mutation*.

Crossover The *crossover* operator combines two parent individuals and produces two offspring individuals. It is implemented as one-point crossover, where the crossing point is determined at random, but on the border of a block only. The two parts of the individual are crossed over separately, so if parents are $I = (I_1, I_2)$ and $J = (J_1, J_2)$ we run $crossover(I_1, J_1)$ and $crossover(I_2, J_2)$.

Let the two parents be:

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

then, the crossover produces offspring:

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

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

Thus, only the whole layers are interchanged between individuals.

Mutation The *mutation* operator brings random changes to an individual. Each time an individual is mutated, one of the following mutation operators is randomly chosen (each of mutation operators has its own probability):

- mutateLayer - introduces random changes to one randomly selected layer.
- addLayer - one randomly generated block is inserted at random position. If it is inserted to the first part of the individual, its either convolutional layer or max-pooling layer; otherwise it is dense layer.

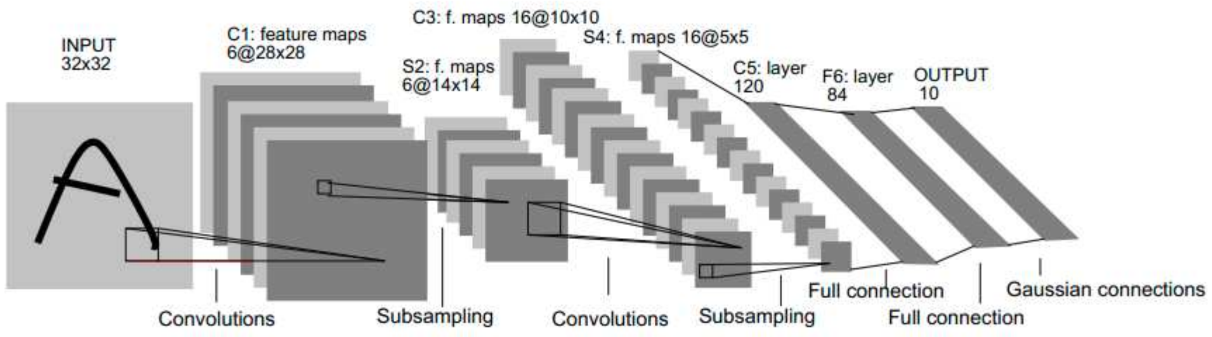

Figure 1: Convolutional neural network [12].

- delLayer - one randomly selected block is deleted.

When *mutateLayer* is performed, again one of the available operators is chosen. For dense layers they are:

- changeLayerSize - the number of neurons is changed. The Gaussian mutation is used, the final number is rounded (since size has to be an integer).
- changeDropOut - the dropout rate is changed using the Gaussian mutation.
- changeActivation - the activation function is changed, randomly chosen from the list of available activations.

For max-pooling layers:

- changePoolSize - the size of pooling is changed.

For convolutional layers:

- changeNumberOfFilters - the number of filters is changed. The Gaussian mutation is used, the final number is rounded.
- changeFilterSize - the size of the filter is changed.
- changeActivation - the activation function is changed, randomly chosen from the list of available activations.

3.3 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 loss function on the testing set over k run is evaluated.

For the classification tasks, categorical crossentropy is used as the loss function.

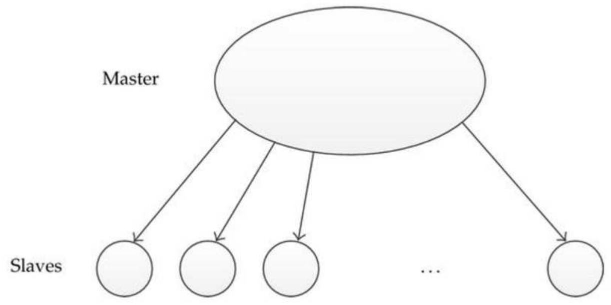

Figure 2: Master-slave parallel computational model.

3.4 Selection

As a parental selection operator, the tournament selection is used in our algorithm. It works as follows, in 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.

4 Asynchronous Evolution

In classical genetic algorithm the individuals are evaluated in generations. In each generation, new individuals are produced based on operators selection, mutation, and crossover, their fitness is evaluated and they replace the old generation. The fitness evaluations are independent and can be done in parallel.

There are many approaches to parallelization of genetic algorithms. In general, there are three classes of parallel GA approaches – single population master-slave model, single population fine-grained, and multi-population coarse grained model (see [2, 9] for more details).

In our work we use the master-slave parallelization method (Fig. 2). It works with single population of individuals, and the evaluations of individuals are performed in parallel – the master stores the population and the slaves evaluate the fitness. The algorithm itself is same as for serial GA, each individual may compete and mate with any other.

The fitness evaluation is parallelized so that a fraction of the population is assigned to each of the processors. Communication occurs only when a slave receives an individual to evaluate and when a slave returns a fitness value.

The algorithm is normally defined as synchronous, i.e. the master waits for the slaves to receive the fitness values for all the population and only then it proceeds to the next generation. Such synchronous master-slave algorithm has exactly the same properties as a simple sequential algorithm.

However, the disadvantage of such synchronous parallel approach is that in general, not all fitness evaluations require same time. In our particular case, some individuals represent large networks and require long time to evaluate, on the other hand there might be very small networks that are evaluated much faster. At the end of each generation, there are processors that are already finished and waiting for the ones evaluating large networks.

As a solution to this problem we have chosen to use *asynchronous evolution* [18]. In asynchronous evolution there is no notion of generations, but as soon as there is a free processor, a new individual is generated and send for evaluation. Such approach may significantly improve the usage of computationally resources. The scheme of the algorithm is presented in Alg. 1.

One of the features of asynchronous approach is that it is naturally biased towards solution with faster fitness evaluation. In our case we consider this feature to be an advantage, because we are more interested in smaller networks with shorter learning time. As our practical experiments in the next section imply, this feature is not harmless, it seems in fact not sufficient and maybe a further discrimination of larger networks in fitness function can be beneficial.

The (synchronous and asynchronous) master-slave approach can be easily and efficiently implemented both on shared-memory and distributed-memory parallel computers. On a shared-memory multiprocessor, the population is stored in memory and each slave process can access the individuals assigned to itself. On a distributed-memory computer, the master process is responsible for storing the population, sending the individuals to other processes (slaves), collecting the results and producing new generation by genetic operators.

5 Experiments

For our experiments we have chosen the well known MNIST data set [13] and Fashion-MNIST data set [26].

Each data set contains 70 000 images of 28×28 pixel. 60 000 are used for training, 10 000 for testing. In MNIST, there are images of handwritten digits (see Fig. 3), while in Fashion-MNIST are images of fashion objects (Fig. 4).

Our implementation of the proposed algorithm is available at [23].

Algorithm 1 Asynchronous EA

```
procedure ASYNCEA(MINPOPSIZE,POPSIZE)
    $P \leftarrow \emptyset$
    while $|P| < minPopSize$ do
        if not node available then
            wait()
        end if
        while node available do
            ind ← RandomIndividual()
            evaluate(ind)
        end while
        evaluatedInd ← getEvaluatedIndividual()
        $P \leftarrow P \cup \{$evaluatedInd$\}$
    end while
    ind ← produceIndividual()
    evaluate(ind)
    while termination criterion not met do
        evaluatedInd ← getEvaluatedIndividual()
        $P \leftarrow P \cup \{$evaluatedInd$\}$
        if $|P| > popSize$ then
            discard the worst individual from $P$
        end if
        ind ← produceIndividual()
        evaluate(ind)
    end while
end procedure
```

We have run the algorithm with population of 30 individuals on 10 processors for one week.

When the best individual is obtained, the corresponding network is built and trained on the whole training set and evaluated on the test set, which was also done for the baseline model designed by human. For both models, the RMSProp algorithm for 20 epochs was used.

The resulting classification accuracy on the test set is listed in Tab. 1 and Tab. 4 for MNIST and Fashion-MNIST data sets respectively. The obtained accuracies can be further improved (especially in case of Fashion-MNIST) by tuning the parameters of RMSProp (default setup was used in our case). On both datasets the obtained evolved network gives competitive results to the baseline model.

In Tab. 2 and Tab. 3, there are listings of baseline and evolved architectures for MNIST and Fashion-MNIST, respectively. The `conv #32` stands for convolutinal layer with 32 filters, `dense #128` stands for dense layer with 128 neurons, and `pool` stands for max-pooling layer, etc.

On the MNIST data set, the evolved architecture has even less number of weights than the baseline model. However, on the Fashion-MNIST the evolution was not so successful and the evolved architecture is quite bloated. The architecture complexity is not reflected in the fitness function directly, but the asynchronous evolution should tend to prefer smaller networks.

We have also compared the synchronous parallelization with asynchronous one in terms of time requirements. We

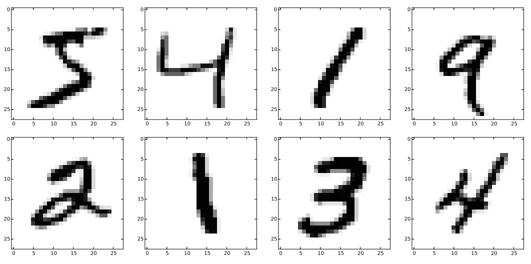

Figure 3: Example of MNIST data set samples.

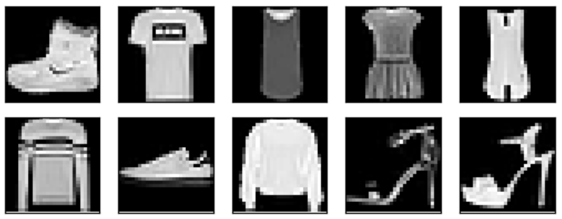

Figure 4: Example of Fashion-MNIST data set samples.

model	avg	std	min	max
baseline	98.97	0.07	98.84	99.13
evolved	**99.25**	0.09	99.10	99.37

Table 1: Test accuracies on the MNIST dataset.

Baseline network
conv #32 kernelsize=3 activation=relu
conv #32 kernelsize=3 activation=relu
pool poolsize=2
dense #128 dropout=0.5 activation=relu
Trainable params: 600,810

Evolved network
conv #22 kernelsize=2 activation=tanh
conv #31 kernelsize=5 activation=linear
pool poolsize=3
conv #33 kernelsize=5 activation=relu
dense #143 dropout=0.4 activation=relu
dense #42 dropout=0.0 activation=tanh
Trainable params: 431,659

Table 2: Baseline and evolved network for MNIST.

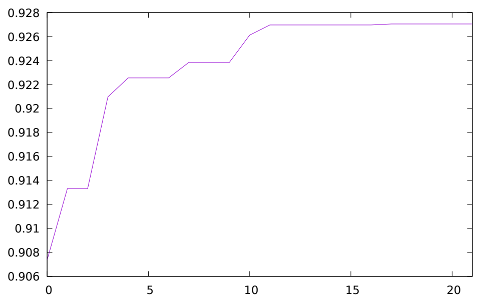

Figure 5: The fitness function through evolution.

Baseline network
conv #32 kernelsize=3 activation=leakyRelu
pool poolsize=2
conv #64 kernelsize=3 activation=leakyRelu
pool poolsize=2
conv #128 kernelsize=3 activation=leakyRelu
pool poolsize=2
dense #128 dropout=0.3 activation=leakyRelu
Trainable params: 356,234

Evolved network
conv #46 kernelsize=3 activation=relu
conv #15 kernelsize=3 activation=relu
conv #36 kernelsize=4 activation=relu
conv #13 kernelsize=3 activation=relu
conv #36 kernelsize=3 activation=relu
pool poolsize=2
dense #235 dropout=0.4 activation=hard_sigmoid
dense #130 dropout=0.3 activation=tanh
Trainable params: 1,714,219

Table 3: Baseline and evolved network for Fashion-MNIST.

model	avg	std	min	max
baseline	91.64	0.37	90.77	91.97
evolved	**92.32**	0.52	91.07	92.86

Table 4: Test accuracies on the Fashion-MNIST dataset.

have run both algorithms on 5 processors for 4 days with population size 20. The asynchronous version made 140 fitness evaluations, while the synchronous version 100 fitness evaluations. So the asynchronous version may bring quite important time saving.

6 Conclusion

We have proposed an algorithm for automatic design of convolutional networks based on asynchronous evolution. The algorithm was tested in experiments on two image classification tasks, the MNIST and Fashion-MNIST data sets. The evolved networks were compared to baseline models, and they achieved competitive results (in terms of slightly better classification accuracies). We have shown that it is possible to automatically find solutions comparable to those designed by human expert.

The main limitation of the presented algorithm is its time complexity. The possibility to trade human expert knowledge for computational resources should be seen as an advantage in several scenarios. Our main motivation is to develop an autonomous system capable of creating machine learning models without human intervention. This may be useful for the cases where no expert is available or a new task without prior experience is encountered. Also, in critical cases where even a small performance gain is necessary, our approach has demonstrated its usability.

One direction of our future work is to try to lower the number of fitness evaluations using surrogate modelling [8] or to investigate other types of parallel evolutionary algorithms (such as multi-deme GA [9]). We also plan to tune automatically the learning algorithm, i.e. search for other hyper-parameters, such as the type of learning algorithm, learning rates, etc.

Acknowledgment

This work was partially supported by the Czech Grant Agency grant 18-23827S 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.

References

[1] J. Arifovic and R. 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] E. Cantú-Paz. A survey of parallel genetic algorithms. *CALCULATEURS PARALLELES*, 10, 1998.

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

[4] O. E. David and I. 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] D. Floreano, P. Dürr, and C. Mattiussi. Neuroevolution: from architectures to learning. *Evolutionary Intelligence*, 1(1):47–62, 2008.

[6] F. Gomez, J. Schmidhuber, and R. Miikkulainen. Accelerated neural evolution through cooperatively coevolved synapses. *Journal of Machine Learning Research*, pages 937–965, 2008.

[7] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[8] Y. Jin. Surrogate-assisted evolutionary computation: Recent advances and future challenges. *Swarm and Evolutionary Computation*, 1(2):61 – 70, 2011.

[9] D. S. Knysh and Victor M. Kureichik. Parallel genetic algorithms: a survey and problem state of the art. *Journal of Computer and Systems Sciences International*, 49(4):579–589, Aug 2010.

[10] J. Koutník, J. Schmidhuber, and F. 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.

[11] Y. Lecun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 5 2015.

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

[13] Y. LeCun and C. Cortes. The mnist database of handwritten digits, 2012.

[14] I. Loshchilov and F. Hutter. CMA-ES for hyperparameter optimization of deep neural networks. *CoRR*, abs/1604.07269, 2016.

[15] T. H. Maul, A. Bargiela, S.-Y. Chong, and A. 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.

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

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

[18] E. O. Scott and K. A. De Jong. Understanding simple asynchronous evolutionary algorithms. In *Proceedings of the 2015 ACM Conference on Foundations of Genetic Algorithms XIII*, pages 85–98, 2015.

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

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

[21] M. Suganuma, S. Shirakawa, and T. Nagao. A genetic programming approach to designing convolutional neural network architectures. *CoRR*, abs/1704.00764, 2017.

[22] 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.

[23] P. Vidnerová. GAKeras. `github.com/PetraVidnerova/GAKeras`, 2017.

[24] P. Vidnerová and R. Neruda. Evolution strategies for deep neural network models design. In *Proceedings ITAT 2017: Information Technologies - Applications and Theory. Aachen & Charleston: Technical University & CreateSpace Independent Publishing Platform, 2017 - (Hlaváčová, J.), CEUR Workshop Proceedings, V-1885*, pages 159–166.

[25] P. Vidnerova and R. Neruda. Evolving keras architectures for sensor data analysis. In *2017 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 109–112, Sept 2017.

[26] H. Xiao, K. Rasul, and R. Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017.