Growing an architecture for a neural network

Ekaterina Shemyakova
(w/ Sergei Khashin)

University of Toledo
USA

arXiv:2108.02231 [cs.LG]

Abstract

We propose a novel automatic architecture search algorithm that uses both pruning connections and adding neurons.

To test the algorithm's effectiveness we consider two standard problems:

1. The brightness prediction problem, where we need to predict the brightness of the next point based on previous points within an image;
2. the approximation of the function defining the brightness of a black-and-white image.

The optimized networks significantly outperform the standard solution for neural network architectures in both cases: for the same error, we can have a significantly smaller complexity.

The *growing architecture* algorithm is a combination of ideas of pruning and constructing algorithms.

Our architectures do not need to be layered.

Architecture search approaches

Approaches to automatic architecture search

- *Empirical/statistical methods* that choose the weights according to the effect they make on the model's performance, see, e.g. Benardos and Vosniakos 2002.
- *Evolutionary algorithms* that start with selecting parent networks, then proceed with combination and mutations, and selecting the best ones. See e.g. Benardos and Vosniakos 2007; Koza and Rice 1991; Miikkulainen et al. 2019.
- *Pruning methods* that start with a larger than necessary multilayer network and then remove neurons that have little contribution to the solution. There are several different ways to decide which neuron is not needed, see e.g. Reed 1993; Mozer and Smolensky 1989; Karnin 1990; LeCun, Denker, and Solla 1990; Castellano, Fanelli, and Pelillo 1997. The known problem is that one usually does not know a priori how large the original network should be. Also starting with a large network could be excessively costly to trim the unnecessary units.

- *Constructive methods* that start with an initial network of small size, and then incrementally add new hidden neurons and/or hidden layers, see e.g. surveys Tin-Yau Kwok and Yeung 1997 Lee 2012, and e.g. papers Ma and K. Khorasani 2003; Ash 1989; Weng and Khashayar Khorasani 1996; Prechelt 1997; S.E. Fahlman 1990; Tin-Yau Kwok and Yeung 1996; Tin-Yan Kwok and Yeung 1997; Shaw et al. 2019.
  A known problem is that the size of the obtained multilayer networks is reasonable but rarely "optimal".
- *Cell-based methods* create the architecture from a smaller-sized blocks, see e.g. B. Zoph, Shlens, and Le 2018; Shaw et al. 2019; Wu et al. 2019.

Neuronetwork's complexity

We see the ever-increasing efficiency of neural networks.
At the same time, their complexity is growing.

Here we measure the *complexity* by the number of *weights*.

Those who do not work directly with neural networks usually expect the complexity of the network to be tens, hundreds, at most thousands. In reality, the complexity of modern neural networks is much higher.

Thus, for the standard MNIST handwritten digit classification problem, the number of learnable parameters in the best networks is hundreds of thousands and millions, while there are only $60,000$ training examples (small square 28×28 pixel grayscale images of handwritten single digits between 0 and 9).

Minimization of the network complexity is the goal of our work!

Realization

Most of our computations are realized in C++ instead of some conventional package (e.g. Keras/Python). This is because layered architectures are the main objective of such specialized packages, and dealing with non-layered ones presents such difficulties that outweigh their conveniences.

Hardware specification: Intel(R) Pentium(R) CPU G4500 @ 3.50GHz, 32GB and Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz 16 GB.

Software specification: Visual Studio 2019 Community (C++), Python 3.9.5, PyCharm 2022.3.2 Community Edition, Numpy 1.22.4, TensorFlow 2.5.0.

Neural networks

Neural networks

Neural networks consist of a graph that together with some other data implement a nonlinear function $f : X \to Y$.

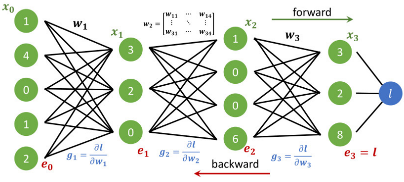


Figure: Example of a layered architecture from "Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks" by T. Hoefler et al.

Starting point: Given a training set (dataset), a subset in $X \times Y$, we start with some $f(x, w) : X \to Y$, parameterized by weights $w \in \mathbb{R}$.
Training for weights: we transform the input (x_0 on the picture) layer by layer to generate the output (l on the picture). This process is called "the inference", or "the forward pass". Training on known (x_0, l), we are searching for the values of w such that $f(x; w)$ is close to y.

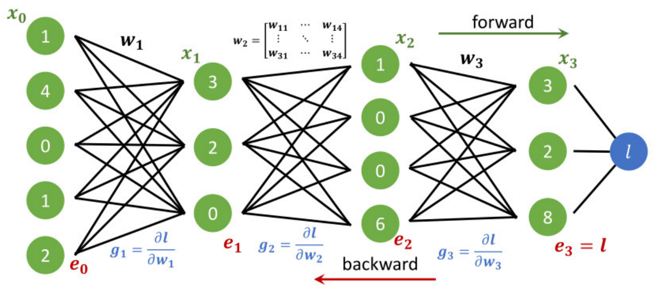
Finding a suitable network for a given problem consists of two steps:
1. decide on the network structure (architecture),
2. train for the weights.

The network architecture is traditionally designed manually and not changed during the training process.

Training for the weights starts with picking some weights and applying $f(x, w)$ storing the inputs of each layer. The quality is evaluated using a loss function $l : Y \times Y \to \mathbb{R},\ l : (y, f(x; w)) \mapsto \varepsilon$. At each hidden layer, the "backward pass" uses e.g. a gradient and update the weights using a "learning rule" to decrease the loss.

We iterate this until we find w such that $f(x; w)$ provides the desired accuracy. The accuracy is evaluated on a separate set of examples not used in training.

Example: input neurons are packed into a vector x_0,
there are two *hidden layers*, vectors x_1, x_2, and an *output layer*, vector x_3.
The *activation function*, e.g., ReLU $\sigma(x) = \max(0, x) : \mathbb{R}^n \to \mathbb{R}^n$.



Going from the first to the second layer is

$$x_1 = \sigma(w_1 x_0 + b_1)\,.$$

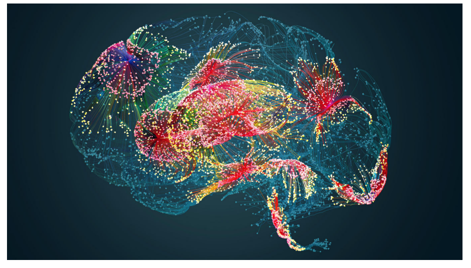
The network function is

$$f(x_0; w) = \sigma(w_3 \sigma(w_2 \sigma(w_1 x_0 + b_1) + b_2) + b_3)\,.$$

Our architecture growth algorithm general idea

Our optimized architecture incorporates pruning enhanced by construction, thus collectively referred to as "growing".

The biological motivation is as follows:



"... networks of brain cells tend to be dominated by a small number of connections that are much stronger than most." [1]
A human brain starts sparse, has an early phase of densification followed by massive pruning, and then remains at a relatively stable sparsity level. Yet, even fully-grown brains change up to 40% of their synapses each day. [2]

[1] Palmer, Lynn, Holmes, Nature Physics'24
[2] Hawkins 2017

"Prunning" or "sparsifying" of our architecture can enhance efficiency without sacrificing accuracy.

Removing neurons $\sim$ removing rows or columns in the layer weight matrices;

Removing inputs $\sim$ removing elements of the matrices.

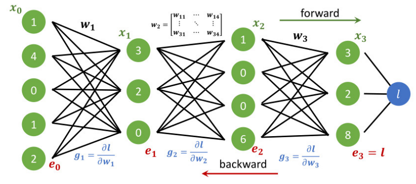


Figure: Example of a 4-layered architecture

Our algorithm step-by-step

Pruning procedure: one synapse removal

- Find synapses with relatively low weights. For each neuron, divide all of its input weights by its maximum input weight. Then, consider all these numbers collectively for all neurons and select a certain number (e.g., $N_1 = 8$) of the smallest averaged weights. These will be the candidates for removal.
- For each candidate, compute the value of the *current loss function* after its removal.
- Select (e.g. $N2 = 3$) synapses, the removal of which minimally impacts the loss function.
- For each of these N_2 synapses, we remove it, and then train the remaining neural network within a given time. The weights get adjusted, but we do not modify zeros.
- We fix the removal of only one synapse, that yields the best value for the *new loss function*.

We repeat the procedure until the loss function increases by no more than $(1 + \varepsilon)$ times, where ε is sufficiently small. In our experiments $\varepsilon = 0.002$.

Constructing procedure: adding a neuron

1. Add an additional neuron to the beginning of the neural network (in layer 1/2) and connect it to all input parameters and to all other neurons of the original network. We lose the layered structure at this point
2. Assigning weights to the new neuron. Starting with all zero weights does not work well. Instead, we make several random weights assigning attempts (from the interval $[-1, 1]$) (e.g. $N_3 = 100$), and choose the one for which the loss function is the smallest.
3. Train the new network for some time (longer than before, but also not for long). As a result, the value of the loss function improves as we have more options.

Architecture growing algorithm

We start with an arbitrary architecture and then execute the following procedure.

1. Remove all redundant connections (synapses) as described above.
2. If the complexity of the network reaches the preset limit, end the procedure.
3. Add a neuron.
4. Return to step (1).

Example 1: brightness prediction

Example 1. Brightness prediction, Data

To illustrate our idea, consider the brightness prediction problem for an image point knowing the brightness of several previous points (we take five).

This is needed for image compression algorithms.

The *previous points* are ordered as shown in the table. Here $*$ is the current point, and the first column and the first row give the (x, y)-coordinates of the points relative to $*$. The other numbers in the table indicate the order in which the points will be considered.

-	2	1	3
4	0	*	-

Table: Ordering of the previous five points. Here $*$ is the current point.

For the experiments, we choose a graphics file of size 682×512:

Figure: Test image

Every point is characterized as (R, G, B), $R, G, B \in [0, 256)$. For a black-and-white picture $R = G = B$. We will call this number brightness.

We need to construct a function of five arguments (points numbered $0, 1, 2, 3, 4$).
The data can be organized into the following matrix:

$$\begin{matrix} *_1 & 0_1 & 1_1 & 2_1 & 3_1 & 4_1 \\ *_2 & 0_2 & 1_2 & 2_2 & 3_2 & 4_2 \\ \vdots & & & & & \end{matrix}$$

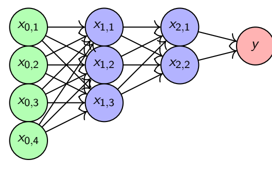
Number of rows: 682×512 (subtract boundary points).

Instead, we can consider a matrix of increments with respect to the second column (effectively subtract the second column (0_i-th) from every column of the matrix, as a result, we get a matrix with the second zero column which can be removed). Also, the entries become much smaller.
We also divide all numbers by 255, to make them smaller. Such are usually better for neural networks (as standard training methods have been developed to govern such situations).

Now, we need to construct a function of four arguments.

Example 1, Keras networks

First, we use Keras to build a 3-layer neural network with N_1 neurons in the first hidden layer, with N_2 neurons in the second hidden layer, and one output neuron.



Example 1, Keras networks

Within Keras package (inside Python):

The training method is Adam.

The number of training epochs is 10000. (Divide data into batches, and for each interaction of the training we are minimizing the loss function computed only for a specific batch. We go through all batches eventually as we run more and more iterations. This is a standard trick.)

We choose *tanh* (hyperbolic tangent) as the activation function.

$N1, N2$ is the number of neurons in layers 1 and 2.

N_par is the total number of parameters.

The loss function S is the sum of squares of increments divided by the number of points.

N1	N2	N_par	loss function
5	5	61	112.4630
7	6	90	104.1467
8	7	111	103.5898
9	7	123	101.0789
8	9	131	97.1992
10	11	183	94.9745
12	11	215	87.9522
12	14	257	86.6209
15	15	331	83.8935
17	17	409	77.1056
20	19	519	70.5793
20	20	541	66.5645
22	22	639	61.8804
25	25	801	60.5896

Every row corresponds to a network built by Keras based on our particular dataset and for our particular problem. So, the loss function value of 60 means that the absolute error is $\sqrt{60} \approx 7.5$. With a bare eye, we cannot distinguish between $1-2$.

Compare them with our optimized networks:

As the starting point, we use the simplest Keras's network, corresponding to the first row in the table.

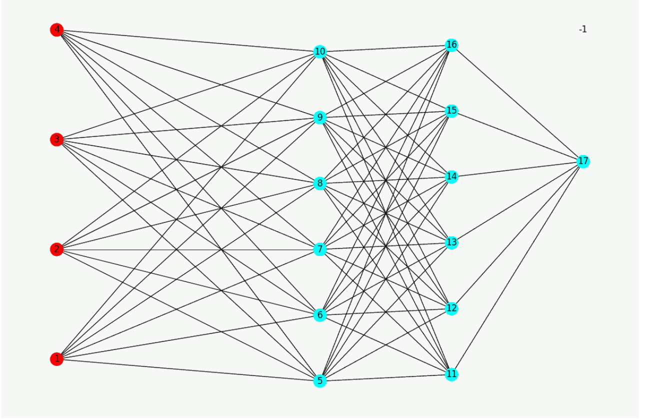


Figure: Initial neuronetwork

6 neurons in the 1st and in the 2nd layers.

Intermediate network:

Neurons numbered 5 to 13 have been added (one at each iteration) as an extra 1/2 layer.

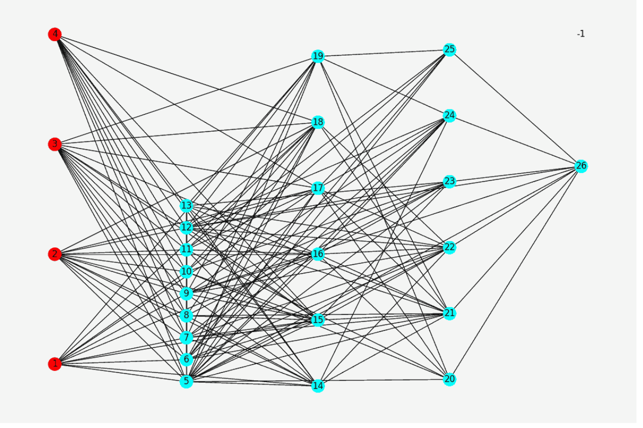


Figure: In the middle of optimization

6 neurons in the 1st and in the 2nd layers and 9 additional neurons.

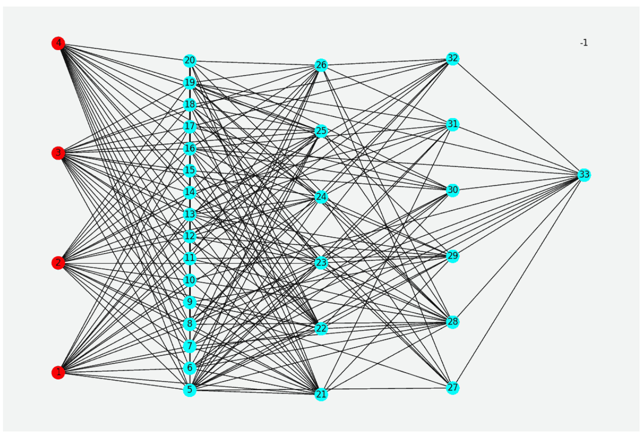


Figure: At the end of optimization

6 neurons in the 1st and in the 2nd layers and 16 additional neurons.

Comparison of our optimization and Keras:

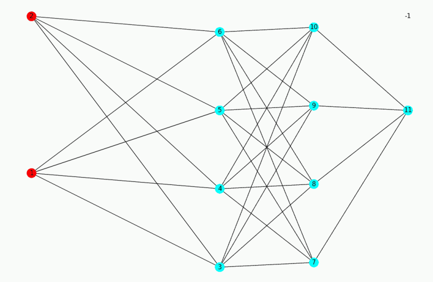
Loss fun	Optimized # of params	Keras params	Ker/Opt ratio
109.58	79	79	1.00
103.66	68	108	1.59
96.59	93	145	1.56
94.78	106	184	1.74
91.46	113	199	1.76
86.31	138	264	1.91
84.72	149	309	2.07
82.25	153	343	2.24
81.22	166	352	2.12
80.30	178	359	2.02
79.03	189	374	1.98
77.70	202	398	1.97
76.82	211	414	1.96
75.26	230	440	1.91

Example 2

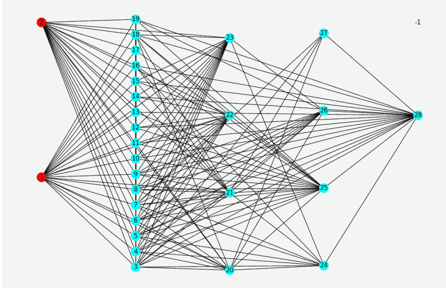
Example 2

For visualization, consider the following problem: approximate the same black-and-white graphics file of size 682×512 using a function of two input variables, $f(x, y)$. We start with a 3-layered architecture with two inputs, N_1 neurons in the first layer, N_2 neurons in the second layer, and 1 output neuron. Here $4 \leq N_1, N_2 \leq 40$.

We initially train the networks using *TensorFlow/Keras*, and subsequently optimize them using our approach.



Optimized initial network.



Optimized at an intermediate stage.

Optimization results:

Loss fun	Opt params	Keras params	Keras/opt
0.01685493	37	37	1.00
0.01429114	54	76	1.41
0.01308279	70	109	1.56
0.01283806	80	112	1.40
0.01136803	106	140	1.32
0.01059770	131	165	1.26
0.00958063	176	280	1.59
0.00935498	200	298	1.49
0.00897876	220	455	2.07
0.00870600	241	516	2.14
0.00849940	280	549	1.96
0.00841038	300	564	1.88
0.00828379	327	584	1.79
0.00825470	339	592	1.75

Table: Comparison of our optimization and Keras results

The approximation is far from perfect.

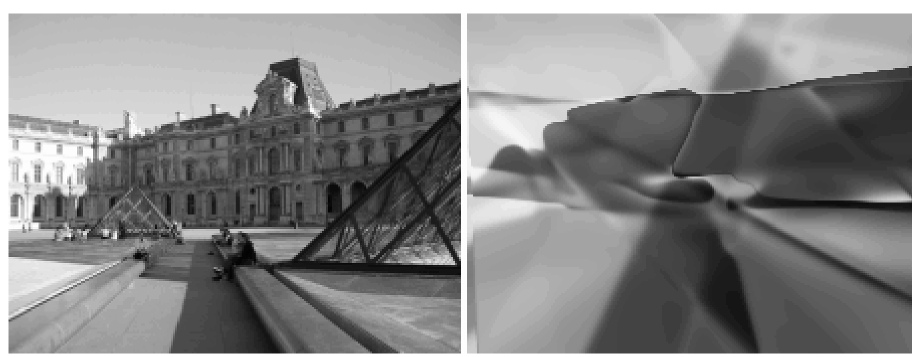
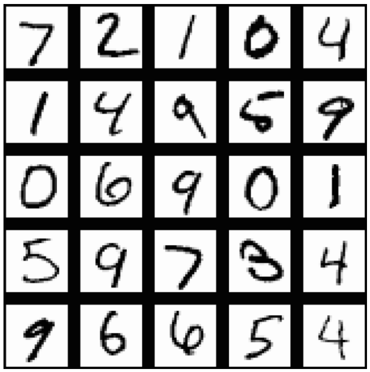


Figure: Source and approx image

Example 3

Example 3. MNIST

Consider the standard task MNIST: $70,000$ pictures of size $28 * 28$, which depict handwritten numbers:



The first $60,000$ is (standardly) used for training, and $10,000$ is used for testing (against over-training).

In the starting network, we have input layer $|x_0| = 28 \times 28$. We build a usual 3-layer neural network with seven neurons in the first hidden layer, and seven neurons in the second hidden layer, and ten output neurons (since this is a classification problem and we have ten digits, if the seventh number is the largest, then the guess is 7).

The total number of parameters is 5631.

We use Softmax loss function: the output layer $y = (y_0, \dots, y_9)$, where y_i are some real numbers, usually they stay within $[-10, 10]$. Construct

$$z_i = \frac{e^{y_i}}{e^{y_0} + \cdots + e^{y_9}}.$$

These z_i will be all positive and sum up to 1.

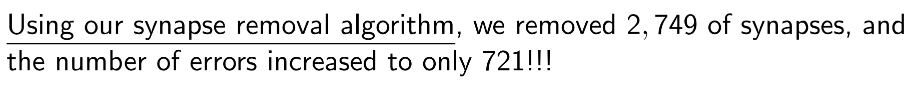
$$l(\underbrace{A}_{\text{correct answer}}, y) = -\ln(z_A)$$

E.g. let the correct answer be $A = 7$, and the network correctly returned $y_7 >> y_i$ for all other i. Then z_7 will be almost 1 and the others will be very small.

Standard Keras network:

```
model = tf.keras.Sequential(
 [
    tf.keras.layers.Dense(N1, activation = tf.nn.relu,
       input_shape=(v_len,)),
    tf.keras.layers.Dense(N2, activation = tf.nn.tanh),
    tf.keras.layers.Dense(num_classes,
       activation="softmax"),
 ]
)
model.compile(optimizer=tf.keras.optimizers.Adam(),
      loss='sparse_categorical_crossentropy',
      metrics=['accuracy'])
```

If we use a standard Keras network, then after training it has 711 errors (per $10,000$ test images) with 5631 parameters.


Using our synapse removal algorithm, we removed $2,749$ of synapses, and the number of errors increased to only 721!!!

That is, with a slight deterioration in quality, we were able to remove 49% of the synapses.

Thus, with almost no loss of quality, the complexity of the neural network has almost halved.

Conclusions

We propose a novel automatic architecture search algorithm.

The algorithm alternates between pruning connections and adding neurons, without restricting itself to layered networks.

Instead, we search for architectures among arbitrary oriented graphs with weights (alongside biases and an activation function), allowing for networks without a layered structure. The objective is to minimize complexity while maintaining a specified error threshold.

In the examples examined, the complexity (number of connections) of the optimized neural network is approximately halved compared to the original.

References I

- Ash, T. (1989). “Dynamic Node Creation in Backpropagation Networks”. In: *Connection Science* 1.4, pp. 365–375.
- B. Zoph Vasudevan, Vijay, Jonathon Shlens, and Quoc V Le (2018). “Learning transferable architectures for scalable image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 8697–8710.
- Benardos, P.G. and G.C. Vosniakos (2002). “Prediction of surface roughness in CNC face milling using neural networks and Taguchi's design of experiments”. In: *Robotics and Computer-Integrated Manufacturing* 18.5-6, pp. 343–354.
- — (2007). “Optimizing feedforward artificial neural network architecture”. In: *Engineering Applications of Artificial Intelligence* 20.3, pp. 365–382.

References II

Castellano, Giovanna, Anna Maria Fanelli, and Marcello Pelillo (1997). "An iterative pruning algorithm for feedforward neural networks". In: *IEEE transactions on Neural networks* 8.3, pp. 519–531.

Hawkins, J. (2017). "Special report : Can we copy the brain? - What intelligent machines need to learn from the Neocortex". In: *IEEE Spectrum* 54.6, pp. 34–71. DOI: 10.1109/MSPEC.2017.7934229.

Karnin, E.D. (1990). "A simple procedure for pruning back-propagation trained neural networks". In: *IEEE Transactions on Neural Networks* 1.2, pp. 239–242.

Koza, J. and J. F. Rice (1991). "Genetic generation of both the weights and architecture for a neural network". In: *IJCNN-91-Seattle International Joint Conference on Neural Networks*. Vol. 2, pp. 397–404.

References III

Kwok, Tin-Yan and Dit-Yan Yeung (1997). "Objective functions for training new hidden units in constructive neural networks". In: *IEEE Transactions on neural networks* 8.5, pp. 1131–1148.

Kwok, Tin-Yau and Dit-Yan Yeung (1996). "Bayesian regularization in constructive neural networks". In: *International Conference on Artificial Neural Networks*. Springer, pp. 557–562.

— (1997). "Constructive algorithms for structure learning in feedforward neural networks for regression problems". In: *IEEE transactions on neural networks* 8.3, pp. 630–645.

LeCun, Yann, John S Denker, and Sara A Solla (1990). "Optimal brain damage". In: *Advances in neural information processing systems*, pp. 598–605.

Lee, Tsu-Chang (2012). *Structure level adaptation for artificial neural networks*. Vol. 133. Springer Science & Business Media.

References IV

- Ma, L. and K. Khorasani (2003). "A new strategy for adaptively constructing multilayer feedforward neural networks". In: *Neurocomputing* 51, pp. 361–385.
- Miikkulainen, R. et al. (2019). "Ch.15 - Evolving Deep Neural Networks". In: *Artificial Intelligence in the Age of Neural Networks and Brain Computing*, pp. 293–312.
- Mozer, Michael C and Paul Smolensky (1989). "Skeletonization: A technique for trimming the fat from a network via relevance assessment". In: *Advances in neural information processing systems*, pp. 107–115.
- Prechelt, Lutz (1997). "Investigation of the CasCor family of learning algorithms". In: *Neural Networks* 10.5, pp. 885–896.
- Reed, Russell (1993). "Pruning algorithms – a survey". In: *IEEE transactions on Neural Networks* 4.5, pp. 740–747.

References V

- S.E. Fahlman, C. Lebiere (1990). "The cascade-correlation learning architecture". In: *Advances in Neural Information Processing Systems 2*.
- Shaw, A.E. et al. (2019). "SqueezeNAS: Fast Neural Architecture Search for Faster Semantic Segmentation". In: *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*, pp. 2014–2024.
- Weng, Wei and Khashayar Khorasani (1996). "An adaptive structure neural networks with application to EEG automatic seizure detection". In: *Neural Networks* 9.7, pp. 1223–1240.
- Wu, Bichen et al. (2019). "Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 10734–10742.