Neural Computing and Applications (2023) 35:20273–20298
https://doi.org/10.1007/s00521-023-08806-4

ORIGINAL ARTICLE

Bio-inspired machine learning: programmed death and replication

Andrey Grabovsky[1,2] · Vitaly Vanchurin[3,4]

Received: 29 November 2022 / Accepted: 28 June 2023 / Published online: 20 July 2023

Abstract
We analyze algorithmic and computational aspects of biological phenomena, such as replication and programmed death, in the context of machine learning. We use two different measures of neuron efficiency to develop machine learning algorithms for adding neurons to the system (i.e., replication algorithm) and removing neurons from the system (i.e., programmed death algorithm). We argue that the programmed death algorithm can be used for compression of neural networks and the replication algorithm can be used for improving performance of the already trained neural networks. We also show that a combined algorithm of programmed death and replication can improve the learning efficiency of arbitrary machine learning systems. The computational advantages of the bio-inspired algorithms are demonstrated by training feedforward neural networks on the MNIST dataset of handwritten images.

Keywords Machine learning · Neural networks · Bio-inspired algorithms · Neuron correlations · Pruning algorithms · Constructive algorithms · Classification

1 Introduction

Artificial neural networks [1–3] have been successfully used for solving computational problems in natural language processing, pattern recognition, data analysis, etc. In addition to the empirical results, a number of statistical approaches to learning were developed [4–6] (see also [7] for a recent book on the subject) and some steps were taken toward developing a fully thermodynamic theory of learning [8]. The thermodynamic theory was recently applied to model biological systems with evolutionary phenomena viewed as (either fundamental or emergent) learning algorithms [9, 10]. In particular, the so-called programmed death and replication (of information processing units, such as cells or individual organisms) were shown to be of fundamental importance for biological evolution modeled through learning dynamics. More generally, many statistical [8], quantum [11], critical [12] and even gravitational [13] systems can be modeled using learning dynamics, and perhaps the entire universe may be viewed as a neural network that is undergoing learning evolution [14]. However, for the quantum behavior to emerge it is essential that the total number of neurons is not fixed and could change over time [11]. This implies that in the emergent quantum systems, individual neurons should be constantly removed and added, similar to biological organisms, where individual cells constantly die and replicate. In this paper, we analyze the machine learning algorithms that are inspired, first and foremost, by biology [9] (i.e., the programmed death and replication), but at the same time have direct connection to physics in general [14] and to emergent quantum mechanics in particular [11] (i.e., removal and addition of neurons).

Consider an artificial neural network that is being trained for some training dataset using some version of stochastic gradient descent. In this very general setup all of the neurons process information, but it is not clear which ones are more efficient and which ones are less efficient. For example, if we are to remove a single neuron what

✉ Andrey Grabovsky
a.v.grabovsky@inp.nsk.su

Vitaly Vanchurin
vvanchur@d.umn.edu

[1] Budker Institute of Nuclear Physics, Novosibirsk, Russia 630090

[2] Novosibirsk State University, Novosibirsk, Russia 630090

[3] National Center for Biotechnology Information, NIH, Bethesda, MD 20894, USA

[4] Duluth Institute for Advanced Study, Duluth, MN 55804, USA

should it be so that the overall increase of the average loss function would be minimal? In other words, how do we find the least efficient neuron that has the least impact on the overall performance of the neural network? Can such a neuron be identified locally (e.g., by analyzing its state, bias and weights) or do we need to study global properties of the network (e.g., by analyzing non-local statistical or spectral properties)? If all neurons with low efficiency (or least loaded) can be identified, then one may be able to develop an algorithm (i.e., programmed death algorithm) that would be useful, for example, for compression of neural networks. Likewise, if one can identify the neurons with high efficiency (or most loaded), then, perhaps, one can use this information to develop replication algorithms where additional neurons would be added to the system to reduce the load on the most efficient neurons. Moreover, the two algorithms may also be used in conjunction (with programmed death followed by replication) in order to improve the learning rate of the existing machine learning algorithms. In this paper we will give answers to all of the above questions by carrying out analytical calculations (a more general, but only approximate method) and by conducting numerical experiments (a more special, but exact method).

Elimination of neurons is addressed by pruning algorithms. Optimal Brain Surgeon algorithm [15] is applied to the network in a local minimum in error or loss function. One Taylor expands the error as a function of all the weights to the second order and constructs the matrix (Hessian matrix) of the second order derivatives of the error w.r.t. the weights. Next, one solves the problem of minimizing the error increase if one of the weights is set to 0. After that, one eliminates the weight which gives the least error increase if this increase is not too big. The corresponding weights' adjustments are given via the inverse Hessian matrix.

Pruning based on the principle of maximum correlation of errors (MAXCORE) [16] is based on backpropagation of errors. One calculates the errors on the output neurons for the whole training set. From them, one calculates the errors on the hidden layer connected to the output layer. The product of these two error matrices gives the matrix of cross correlation of errors of these two layers. Then, one removes the weights corresponding to the smallest in absolute value elements of this matrix checking that the corresponding error increase is not too big. This procedure is repeated going one layer back to the input layer.

The algorithms based on the variance nullity measure (VNM) [17] calculate the sensitivities of the network's output, i.e., their partial derivatives w.r.t. the weights, biases, and inputs for the input data. Then, they estimate variances of these sensitivities and prune the corresponding elements if these variances are not too big.

All these methods use the error on the output layer to estimate the importance of the particular element, e.g., weight, bias, or input. For each element they calculate a function of the final error and eliminate the element if the value of this function is not too big. In this respect they are global algorithms since they measure the work of the element from the output of the whole net. For a net with very many hidden layers such calculations may become demanding.

Another approach is to estimate the importance of the particular hidden neuron locally, i.e., from its values on the data set and the values of the weights connecting it to the next level in the feedforward net. The neural network pruning by significance (N2PS) algorithm [18] works this way. One assigns the total net value to a hidden neuron estimating the total value of this neuron on the whole data set and calculates the activation function of it. Then one defines the significance measure of this neuron summing the absolute values of the previous result and all the weights connecting this neuron with the next layer. If the significance of the neuron is smaller than the average for its layer, the neuron is pruned.

Zeng and Yeung method [19] assigns relevance to each hidden neuron as a product of neuron's sensitivity and the sum of absolute values of its outgoing weights. Here, sensitivity is the expectation of the difference between the value of the neuron and the value of the neuron calculated for the shifted input. The shift in input is given by its expected absolute value over the data set. Then, one finds the neuron with the smallest relevance and prunes it adjusting the bias in the next layer.

We also propose a local pruning algorithm. It is based on the utilization of the linear dependence of the neurons. Suppose the signals on all the neurons of one level are exactly linear dependent. Then one can express one of the neurons though the linear combination of the other neurons on this level without changing the output of the net thus reducing the number of the neurons by one. In this case it is not important which neuron to express through the others. In reality such linear dependence is approximate and the choice of the neuron to expel introduces an error at the next layer. We propose three algorithms how to construct this approximate linear dependence and how to estimate the neurons efficiency, i.e., its influence on the next layer.

Introduction of new neurons into the net is addressed by the constructive algorithms [20, 21]. They add a hidden neuron to the net when it is stuck in a local minimum so that the error reduces less than a predefined amount during a set number of epochs. One may add a neuron to the existing hidden layer or form a new layer. A new constructive algorithm (NCA) [22], e.g., checks how well the previously added neurons work and adds a new neuron to the new layer instead of adding it to the existing hidden

layer if expected outputs of two previously added hidden neurons differ less than a predefined value. This is the way to avoid redundancy because a new neuron in a new layer will have different input and therefore different functionality.

Here appears a question how to initialize the new neurons. NCA, for example, initializes the weights of a newly added neuron with zeros. In this paper, we introduce a new replication algorithm how to better set the weights for a new neuron when it is added to the existing hidden layer.

Optimization of the net's structure by combination of adding neurons or connections and eliminating them is known as hybrid approach. One may set different goals for this procedure, such as finding the parsimonious architecture or speeding convergence. The constructive algorithm to synthesize arbitrarily connected feedforward neural networks (CoACFNNA) [23], for example, has the former goal. It starts from an empty net and calculates the average joint mutual information (NMI) of input neurons and the outputs. Then, it connects the input neurons with the highest NMI and all outputs. Next, in the circle it adds connections between hidden neurons starting from the pairs with the highest NMI and new hidden neurons checking that the error improves after retraining. A new neuron may be added to the existing layers or as a new layer fully connected with previous and subsequent neurons. In a new circle all connections are tested on their influence on the output error and deleted if it improves the error. Next hidden neurons are sorted according to their NMI and neurons with the smallest NMI are deleted if error is reduced after retraining. As a result, a feedforward net with a more complex than a layer by layer connection is formed.

In this paper, we combine our program death and replication algorithms and show that one can accelerate convergence if once in a preset number of epochs one deletes a number of the least efficient hidden neurons and substitutes them with equal number of new neurons provided by the replication algorithm.

The paper is organized as follows. In the following section, we discuss the basics of artificial neural network and of the learning dynamics. In Sect. 3, we perform statistical analysis of the learning dynamics and introduce two different definitions of neuron efficiency. In Sect. 4, we develop the "programmed death", the "replication" and the combined, i.e., programmed death followed by replication, algorithms. In Sect. 5, we present numerical results for feedforward neural networks with two hidden layers and different architectures. In Sect. 6, the main results are summarized and discussed.

2 Neural networks

A classical neural network with N neurons can be defined as a septuple $(\mathbf{x}, \hat{P}, p_{\partial}, \hat{w}, \mathbf{b}, \mathbf{f}, H)$, where

1. $\mathbf{x}$ is a (column) state vector of neurons,
2. $\hat{P}$ is a boundary projection operator to subspace spanned by input/output neurons,
3. $p_{\partial}(\hat{P}\mathbf{x})$ is a probability distribution which describes the training dataset,
4. $\hat{w}$ is a weight matrix,
5. $\mathbf{b}$ is a (column) bias vector,
6. $\mathbf{f}(\mathbf{y})$ is an activation map, and
7. $H(\mathbf{x}, \mathbf{b}, \hat{w})$ is a loss function.

The training data are associated only with boundary neurons $\hat{P}\mathbf{x}(t)$ that are updated periodically from the probability distribution $p_{\partial}(\hat{P}\mathbf{x})$, but the period depends on the architecture. For example, for a feedforward architecture, the period could equal to the number of layers so that, in-between updates of the training data, the signal has time to propagate throughout the entire network. In contrast to the boundary neurons, evolution of the bulk neurons depends on the state of all neurons,

$$\left(\hat{I} - \hat{P}\right)\mathbf{x}(t+1) = \left(\hat{I} - \hat{P}\right)\mathbf{f}(\hat{w}\mathbf{x}(t) + \mathbf{b}), \tag{1}$$

where the activation map acts separately on each component, i.e., $f_i(\mathbf{y}) = f_i(y_i)$. Here $\hat{I}$ is the identity operator in the space of all neurons: $\hat{I}\mathbf{x} = \mathbf{x}$. For the sake of concreteness, we set activation functions on all boundary neurons to be linear

$$\hat{P}\mathbf{f}(\mathbf{y}) = \hat{P}\mathbf{y} \tag{2}$$

and on all bulk neurons to be hyperbolic tangent

$$\left(\hat{I} - \hat{P}\right)\mathbf{f}(\mathbf{y}) = \left(\hat{I} - \hat{P}\right)\tanh(\mathbf{y}). \tag{3}$$

The main objective of machine learning is to find bias vectors $\mathbf{b}$ and weight matrices $\hat{w}$ which minimize a time (or ensemble) average of some suitably defined loss function. For example, the boundary loss function is given by

$$H_{\partial}(\mathbf{x}, \mathbf{b}, \hat{w}) = \frac{1}{2}(\mathbf{x} - \mathbf{f}(\hat{w}\mathbf{x} + \mathbf{b}))^T \hat{P}(\mathbf{x} - \mathbf{f}(\hat{w}\mathbf{x} + \mathbf{b})), \tag{4}$$

where because of the inserted projection operator $\hat{P}$ the sum is taken over squared error at only boundary neurons. For example, in a feedforward architecture there is no error in the input boundary layer and all of the error is on the output boundary layer due to a mismatch between propagated data and training data. Another example is the bulk loss function defined as

$$H(\mathbf{x},\mathbf{b},\hat{w}) = \frac{1}{2}(\mathbf{x}-\mathbf{f}(\hat{w}\mathbf{x}+\mathbf{b}))^T(\mathbf{x}-\mathbf{f}(\hat{w}\mathbf{x}+\mathbf{b})) + V(\mathbf{x}), \tag{5}$$

where in addition to the first term, which represents a sum of local errors over all neurons, there may be a second term $V(\mathbf{x})$ which represents local objectives. For example, one may add a term rewarding signals close to the upper and lower boundaries of the signal after activation via

$$V(\mathbf{x}) = -\frac{m}{2}\sum_i \mathbf{x}^T\mathbf{x} \tag{6}$$

with an appropriately chosen m.

During learning the trainable variables (i.e., bias vector $\mathbf{b}$ and weight matrix $\hat{w}$) are continuously adjusted (or transformed)

$$b_i(t+T) = b_i(t) - \gamma\frac{\partial\langle H(\mathbf{x},\mathbf{b},\hat{w})\rangle}{\partial b_i}, \tag{7}$$

$$w_{ij}(t+T) = w_{ij}(t) - \gamma\frac{\partial\langle H(\mathbf{x},\mathbf{b},\hat{w})\rangle}{\partial w_{ij}}, \tag{8}$$

where γ is the learning rate and the time-averaged quantities are defined as

$$\langle\ldots\rangle \equiv \lim_{T\to\infty}\frac{1}{T}\sum_{t=1}^{T}\ldots \tag{9}$$

The time interval T can depend on the mini-batch size and on the number of layers. Note that in general the ensemble average (for a given training dataset $p_\partial(\hat{P}\mathbf{x})$) and the time average (for a given time interval T) need not be the same, but, if the trainable variables (i.e., weights and biases) change very slowly, the two averages are approximately the same.

In the long run, the learning system settles down in a local minimum of the average loss function and the learning effectively stops. For certain algorithms the system can still transition to an even lower minimum of the loss function, but such transitions are usually exponentially suppressed. The main problem is that in a local minimum continuous transformations (e.g., stochastic gradient decent) cannot be effective and discontinuous transformations must be performed instead. In Sects. 4 and 5, we shall describe one such transformation by combining two algorithms: programmed death and replication. More generally, the programmed death and replication transformations (or algorithms) need not be combined and can be used separately. The programmed death algorithm may be used to compress, either gradually (i.e., one neuron at a time) or suddenly (i.e., a bunch of neurons at once), the neural network. Such compression could be relevant, for example, if relatively large computational resources are available during the training phase, but the resources are limited during predicting phase. In addition, the replication algorithm may be used for improving the performance of an already pre-trained neural network. This can be done, for example, by adding new neurons in order to assist the most efficient neuron.

3 Statistical analysis

In the previous section, we mentioned the discrete machine learning algorithms (programmed death and replication) that rely on determining the efficiency of individual neurons. In this section we will describe the two definitions of neuron efficiency that will be used in Sect. 4 for developing these algorithms and in Sect. 5 for presenting the results of the numerical experiments.

3.1 Covariance matrix

To study the statistical efficiency of neurons, it is convenient to introduce the covariance matrix

$$C_{ij} \equiv \langle(x_i-\langle x_i\rangle)(x_j-\langle x_j\rangle)\rangle \equiv \langle\Delta x_i\Delta x_j\rangle = \langle x_ix_j\rangle - \langle x_i\rangle\langle x_j\rangle, \tag{10}$$

where $\Delta x_j \equiv x_j - \langle x_j\rangle$. The variables in brackets $\langle x_j\rangle$ denote the expected values of x_j. One can measure them as the average values of x_j over the training dataset.

C is a positive definite symmetric matrix whose eigenvalues λ_l are real nonnegative numbers and the corresponding orthonormal eigenvectors $\mathbf{v}^{(l)}$ are given by

$$C\mathbf{v}^{(l)} = \lambda_l\mathbf{v}^{(l)}. \tag{11}$$

In a general net, it is sensible to consider the covariance matrix of all neurons of the net. However, for a feedforward net, we will use the covariance matrix of the neurons on a particular hidden layer, which we want to prune. Then, the dimension of this matrix is the square of the number of neurons on the particular hidden layer.

If the states of neurons are approximately linearly dependent, i.e., one can find such numbers a_0 and a_k that

$$\sum_k a_kx_k \approx a_0, \tag{12}$$

then at least one of the eigenvalues must be zero. For example, if $\lambda_1 = 0$, then by setting

$$\mathbf{v}_k^{(1)} = a_k \tag{13}$$

we get

$$\sum_k \mathbf{v}_k^{(1)}\langle x_k\rangle = \left\langle\sum_k a_kx_k\right\rangle \approx \langle a_0\rangle = a_0, \tag{14}$$

or

$$\sum_k (x_k - \langle x_k \rangle) \mathbf{v}_k^{(1)} \approx 0. \tag{15}$$

in agreement with zero-eigenvalue equation

$$C\mathbf{v}^{(1)} = \sum_k C_{ik} \mathbf{v}_k^{(1)} = \left\langle \Delta x_i \sum_k a_k \Delta x_k \right\rangle \approx 0. \tag{16}$$

In this limit, any one of the neurons can be removed after appropriate adjustment of weights.

3.2 Efficiency of neurons

In general, $\lambda_i \neq 0$ for all i and then one can define efficiency of individual neurons as the degree of nonlinearity or how poorly the output of a given neuron can be approximated by a linear function of the outputs of all other neurons (for a given training dataset $p_{\mathcal{D}}(\hat{P}\mathbf{x})$). The accuracy of approximation of the state of neuron k is high (and thus the efficiency should be low) if λ_i is small and $\left(\mathbf{v}_k^{(i)}\right)^2$ is large. Therefore, we can define the efficiency of neuron k as

$$E'_k = \min_i \frac{\lambda_i}{\left(\mathbf{v}_k^{(i)}\right)^2}, \tag{17}$$

where the smallest ratio is obtained by looking at all eigenvalues and the corresponding eigenvectors. Then, we use Eq. (15) rewritten for the i-th eigenvalue, i.e.,

$$\sum_k (x_k - \langle x_k \rangle) \mathbf{v}_k^{(i)} \approx 0 \tag{18}$$

to remove the k'th neuron:

$$x_k = \frac{\sum_j \mathbf{v}_j^{(i)} x_j - \sum_{j \neq k} \mathbf{v}_j^{(i)} x_j}{\mathbf{v}_k^{(i)}} \to \frac{\langle \sum_j \mathbf{v}_j^{(i)} x_j \rangle - \sum_{j \neq k} \mathbf{v}_j^{(i)} x_j}{\mathbf{v}_k^{(i)}}. \tag{19}$$

The mistake we make via such a substitution reads

$$\begin{aligned} \delta\left(\sum_l w_{ql} x_l + b_q\right) &= \sum_{l \neq k} w_{ql} x_l + w_{qk} \frac{\left\langle \sum_j \mathbf{v}_j^{(i)} x_j \right\rangle - \sum_{j \neq k} \mathbf{v}_j^{(i)} x_j}{\mathbf{v}_k^{(i)}} \\ &\quad + b_q - \left(\sum_l w_{ql} x_l + b_q\right) \\ &= w_{qk} \frac{\sum_j \mathbf{v}_j^{(i)} x_j - \langle \sum_j \mathbf{v}_j^{(i)} x_j \rangle}{\mathbf{v}_k^{(i)}} \\ &= w_{qk} \frac{\sum_j \mathbf{v}_j^{(i)} \Delta x_j}{\mathbf{v}_k^{(i)}} \sim w_{qk} \sqrt{E'_k} \end{aligned} \tag{20}$$

since

$$C_{ij} \mathbf{v}_i^{(m)} \mathbf{v}_j^{(n)} = \left\langle \left(\sum_i \mathbf{v}_i^{(m)} \Delta x_i\right) \left(\sum_j \mathbf{v}_j^{(n)} \Delta x_j\right) \right\rangle = \lambda_n \delta^{mn}, \tag{21}$$

where $\delta^{mn} = 1$ for $m = n$ and 0 for $m \neq n$. Unfortunately, such an algorithm may not be very useful in practice since it involves calculation of the covariance matrix—a computational task which scales as $\mathcal{O}(N^2)$. What we really want is to be able to identify an approximate linear dependence of neurons, but with an algorithm whose complexity would scale as $\mathcal{O}(N)$.

Consider a linear expansion of activation function

$$x_i = f_i(y_i) + f'_i(y_i) \sum_k w_{ik} (x_k - \langle x_k \rangle) + \ldots \tag{22}$$

where

$$y_i \equiv \sum_k w_{ik} \langle x_k \rangle + b_i. \tag{23}$$

At the zeroth order $\langle x_i \rangle \approx f_i(y_i)$ and at the first order

$$x_i - \langle x_i \rangle \approx f'_i(y_i) \sum_k w_{ik} (x_k - \langle x_k \rangle). \tag{24}$$

If the direct impact of neuron k on neuron i is small, then

$$C_{ii} \gg f'_i(y_i)^2 w_{ik}^2 C_{kk}, \tag{25}$$

or

$$1 \gg f'_i(y_i)^2 w_{ik}^2 C_{kk} C_{ii}^{-1}. \tag{26}$$

In this limit, all possible variations of the signal

$$x_k \approx \langle x_k \rangle \tag{27}$$

do not significantly modify the expected signal x_i, and (if our task is to implement a programmed death algorithm) then we must identify the least efficient neuron by summing over all impacts that a given neuron k has on all other neurons i, i.e.,

$$E_k \equiv C_{kk} \sum_i f'_i(y_i)^2 w_{ik}^2 C_{ii}^{-1}. \tag{28}$$

If $E_k \ll 1$, then we can drop all of the connection from neuron k to neuron i, or the neuron k can be removed without sacrificing much of the neural network performance. More precisely, we can set $w_{ik} = 0$ for all i and then use (27) to readjust biases b_i's of all other neurons so that the input to i-th neuron remains approximately the same. See Eq. (41) with $k = 1$. Note that all of C_{jj}'s and all of $f'_j(y_j)$'s can be calculated locally by analyzing statistics of the signals for each of the neurons separately—a computational task which scales as $\mathcal{O}(N)$.

3.3 Conditional distribution

In the limit opposite to (25) or (26), the impact of neuron k on neuron i is large,

$$C_{ii} \ll f_i'(y_i)^2 w_{ik}^2 C_{kk}, \tag{29}$$

or

$$1 \ll f_i'(y_i)^2 w_{ik}^2 C_{kk} C_{ii}^{-1}, \tag{30}$$

and variations of signal x_k can significantly modify the signal x_i. However, since variations of $f_i'(y_i) w_{ik}(x_k - \langle x_k \rangle)$ must remain much larger than variations of $(x_i - \langle x_i \rangle)$, variations of all other incoming signals $f_i'(y_i) w_{ij}(x_j - \langle x_j \rangle)$ (where $j \neq k$) must be anti-correlated with $f_i'(y_i) w_{ik}(x_k - \langle x_k \rangle)$. More precisely, all of the incoming signals must be approximately linearly dependent and then the output x_k can be approximated as a linear function of the outputs of all other neurons

$$x_k \approx \sum_j \frac{w_{ij}}{w_{ik}} \langle x_j \rangle - \sum_{j \neq k} \frac{w_{ij}}{w_{ik}} x_j + \frac{x_i - \langle x_i \rangle}{f_i'(y_i) w_{ik}}. \tag{31}$$

Then the conditional distribution for variable x_k (with all other x_j's for $j \neq k$ fixed) can be modeled as a Gaussian

$$p(x_k) \propto \exp\left(-\frac{1}{2} \frac{(\mu_i - x_k)^2}{\sigma_i^2} \right) \tag{32}$$

with mean

$$\mu_i \equiv \sum_j \frac{w_{ij}}{w_{ik}} \langle x_j \rangle - \sum_{j \neq k} \frac{w_{ij}}{w_{ik}} x_j \tag{33}$$

and variance

$$\sigma_i^2 = \left(f_i'(y_i)^2 w_{ik}^2 C_{ii}^{-1} \right)^{-1}. \tag{34}$$

To improve the estimate further, we can average over all such approximations for $i \neq k$ and then the overall distribution is proportional to a product of Gaussians

$$p(x_k) \propto \exp\left(-\frac{1}{2} \sum_{i \neq k} \frac{(\mu_i - x_k)^2}{\sigma_i^2} \right) \propto \exp\left(-\frac{1}{2} \frac{(M_k - x_k)^2}{S_k^2} \right) \tag{35}$$

with mean

$$M_k = \frac{\sum_{i \neq k} \mu_i \sigma_i^{-2}}{\sum_{i \neq k} \sigma_i^{-2}} = S_k^2 \sum_{i \neq k} f_i'(y_i)^2 w_{ik} C_{ii}^{-1} \times \left(\sum_j w_{ij} \langle x_j \rangle - \sum_{j \neq k} w_{ij} x_j \right) \tag{36}$$

and variance

$$S_k^2 = \left(\sum_{i \neq k} \sigma_i^{-2} \right)^{-1} = \left(\sum_{i \neq k} f_i'(y_i)^2 w_{ik}^2 C_{ii}^{-1} \right)^{-1}. \tag{37}$$

Note that the neurons efficiency (28) is inversely proportional to the variance

$$E_k = \frac{C_{kk}}{S_k^2} \tag{38}$$

and so the variance is large for neurons with smaller efficiencies and vice versa.

4 Machine learning algorithms

In this section, we develop the three machine algorithms: programmed death, replication and combined (i.e., programmed death followed by replication), using the two measures of efficiencies of individual neurons, or just efficiencies, that were introduce in the previous section. Without loss of generality, we assume that the least efficient neuron is the $k = 1$ neuron where the efficiency is defined either by Eq. (17) (and then the linear dependence of Eq. (15) should be used) or by Eq. (28) (and then the linear dependence of Eqs. (27) or (36) should be used). For the numerical experiments of Sect. 5, we shall refer to

A1. "Connection cut" algorithm—method of Eqs. (27) and (28).
A2. "Probability" algorithm—Eqs. (36) and (28).
A3. "Covariance" algorithm—method of Eqs. (15) and (17).

However, once the least efficient $k = 1$ neuron is identified (using either (17) or (28)) and once an approximate linear relation is established (using either (15), (27) or (36))

$$\sum_j a_j x_j \approx a_0 \tag{39}$$

all three algorithms (i.e., covariance, connection cut and probability) are treated similarly. Also note that in a feedforward architecture the approximate linear dependence (39) can only be established between neurons j on the same layer with the least efficient neuron $k = 1$.

4.1 Programmed death

In the programmed death algorithm, the least efficient neuron is removed from the neural network and the rest of the network is re-wired to accommodate the changes.

The activation dynamics of neuron $x_i = f_i(z_i)$ is determined by the state of all other neurons x_j only through a linear function

$$z_i = \sum_j w_{ij} x_j + b_i, \tag{40}$$

which can be approximated using (39) as

$$\begin{aligned} z_i &= \sum_{j\neq 1} w_{ij} x_j + w_{i1} x_1 + b_i \\ &\approx \sum_{j\neq 1} w_{ij} x_j + w_{i1}\left(\frac{a_0}{a_1} - \sum_{l\neq 1} \frac{a_l}{a_1} x_l\right) + b_i \\ &= \sum_{j\neq 1}\left(w_{ij} - w_{i1}\frac{a_j}{a_1}\right) x_j + \left(b_i + w_{i1}\frac{a_0}{a_1}\right). \end{aligned} \tag{41}$$

If we are to disconnect a neuron from the network with a minimal modification to the states of other neurons, then we have to readjust the biases and weights so that z_i's remain approximately the same. This can be done using the following discrete transformation of the weight matrix

$$w'_{ij} = \begin{cases} 0 & \text{if} \quad i = 1 \ \text{ or } \ j = 1 \\ w_{ij} - w_{i1}\dfrac{a_j}{a_1} & \text{otherwise} \end{cases} \tag{42}$$

and of the bias vector

$$b'_i = \begin{cases} 0 & \text{if} \quad i = 1 \\ b_i + w_{i1}\dfrac{a_0}{a_1} & \text{otherwise} \end{cases}. \tag{43}$$

In other words, the transformation sets all of the signals to and from the "dead" neuron to zero and readjusts all other weights and biases so that equations (41) are approximately satisfied for $i \neq 1$. In this way, it is ensured that the performance of the neural network is not significantly altered or that the value of the average loss function, for a given set of training data, is not significantly changed.

The proposed algorithms have similarities with the works of other authors. As local algorithms [18, 19], they use only the expected data of the particular neurons and the weights connecting them to other neurons to estimate their efficiency. The difference with [18, 19] is in the way we estimate the neuron's efficiency from these data. In the feedforward net our algorithms A1 and A2 use variances of the neurons' signals on the particular neuron's level and the next level to calculate the particular neuron's efficiency. In the case which we study in Sect. 5, e.g., with 784 input and 100 next layer neurons, it means measuring 884 variances and multiplying them and the weights connecting these 2 layers to estimate the input neurons' efficiencies (17). Our covariance algorithm A3 estimates the neuron's efficiency (28) by the covariance matrix of the neurons (on the specific layer for a feedforward net) without knowledge of the weights, i.e., in this particular case one needs to measure the 784×784 covariance matrix and find its eigenvalues and eigenvectors. In this respect, A3 is more computationally demanding than A1 and A2.

This is in contrast to the global algorithms, which estimate the efficiency of a particular neuron or connection from the output error. For example, the OBS algorithm [15] calculates the Hessian matrix of the second derivatives of the final error with respect to the weights. In the example above we have 784×100 weights connecting the input neurons and the second layer neurons. Therefore the Hessian matrix for these weights has the size 78400×78400. Then one has to invert it. This task is more computationally demanding than our algorithms.

Formulas (42) and (43) are similar to the weight and bias adjustments of other authors. For example algorithm of [19] changes biases keeping the weights, which is a particular case of (42) and (43) with $a_1 = 1$, $a_j = 0$ and $a_0 = \langle x_1 \rangle$. We have such substitution in algorithm A1.

The programmed death algorithm is equivalent to a biological phenomenon known as the programmed death, e.g., programmed cell death. From a more practical point of view, the algorithm can be used for compression of neural networks for further use, for example, on devices with constrained computational resources.

4.2 Replication

In the replication algorithm, the learning system adds a new neuron to the neural network in order to reduce the load on the most efficient neuron, not necessarily immediately, but in the long run. Once again, the efficiency of individual neurons can be defined by either (17) or (28), but now the main challenge is to introduce coupling between the two neurons: the new (or "child") neuron "c" and the most efficient (or "parent") neuron "p". By following the biological analogy of the phenomenon of cell replication, we shall only couple the neurons at the time of the replication. This must correspond to some clever reinitialization of biases and weights to and from the child and parent neurons that in the long run would lead to the largest decrease of the average loss function.

For example, consider reinitialization described by a discrete transformation of the weight matrix

$$w'_{ij} = \begin{cases} w_{pj} & \text{if} \quad i = c \\ \dfrac{1}{2} w_{ip} & \text{if} \quad j = c \ \text{ or } \ j = p \\ w_{ij} & \text{otherwise} \end{cases} \tag{44}$$

and of the bias vector

$$b'_i = \begin{cases} b_p & \text{if} \quad i = c \\ b_i & \text{otherwise} \end{cases}. \tag{45}$$

This transformation first splits in half all of the outgoing weights from the parent neuron "p" and then copies all of the outgoing weights from the parent "p" to child neuron

"c". As a result, the overall performance of the neural network is not altered and we end up with two identical (or replicated) neurons that are, however, linearly dependent

$$x_c = x_p. \tag{46}$$

This means that if the programmed death procedure were to be executed right after replication, then it would immediately identify and delete one of these neurons. Moreover, if the two neurons carry exactly the same information, the positive effect of the replication on learning would be only marginal.

To avoid the problem of immediate removal of a newly replicated neuron and to improve the learning efficiency we can set the outgoing weights from the child and parent neurons to be arbitrary given that

$$w'_{ip} + w'_{ic} = w_{ip}. \tag{47}$$

For example, we can split the outgoing signals between the child and parent neurons by defining a discrete transformation of the weight matrix

$$w'_{ip} = \chi_i w_{ip}, \tag{48}$$

$$w'_{ic} = (1 - \chi_i) w_{ip}, \tag{49}$$

where $\chi_i \in \{0, 1\}$ is a random bit. Note, that the randomization procedure can be improved further by employing continuous probability distributions $P(\chi_i)$ which may or may not be symmetric.

The replication algorithm can be used for increasing the effective dimensionality of the space of trainable variables in the regions where the learning resources are most needed. This may be important for initialization of the neural networks as well as for improving the performance of already trained neural networks. More generally, as we shall see below, the replication algorithm can be used in conjunction with programmed death algorithm for improving the convergence of arbitrary learning systems.

5 Numerical results

In the previous section, we developed two bio-inspired machine learning algorithms (i.e., programmed death, replication) and suggested that the programmed death followed by replication can be used for increasing the learning efficiency of arbitrary algorithms. In this section we will analyze the performance of these algorithms by training a feedforward neural network with four layers (i.e., two hidden layers) and different neural architectures:

N1. 784 neurons → 5 neurons → 20 neurons → 10 neurons

N2. 784 neurons → 10 neurons → 10 neurons → 10 neurons

N3. 784 neurons → 20 neurons → 5 neurons → 10 neurons

N4. 784 neurons → 100 neurons → 5 neurons → 10 neurons

We use network N1 in 5.1, 5.6 and 5.7, networks N2 and N3 in Sects. 5.1, 5.6, and network N4 in Sects. 5.2, 5.3, 5.8. We chose nets N1–N3 to see how the work of the algorithms differs for expanding, contracting and constant net architecture in terms of the number of neurons in the consecutive layers. Net N4 was chosen with a large number of neurons to better see the effect of pruning on the second layer. When we discuss the replication algorithms the net N1 is most sensitive to the addition of a neuron to layer 2.

For plotting the numerical results we use the neuron efficiency calculated according to Eq. (28) for algorithms A1 "connection cut" and A2 "probability", and via (17) for the algorithm A3 "covariance" from Sect. 4.

The neural networks were trained on the MNIST [24] dataset of 60,000 data and 10,000 validation 28 × 28 pixel gray level handwritten digits with 784 input channels and 10 output classes 0,..., 9, $p_\theta(\hat{P}\mathbf{x})$; with linear activation function, $f(y) = y$, for input neurons; with the softmax activation function $x_i \to \frac{e^{x_i}}{\sum_k e^{x_k}}$ for the final layer; with nonlinear activation function, $f(y) = th(y)$, for bulk neurons (i.e., second and third layers); and with cross-entropy loss function, $H(\mathbf{x}, \mathbf{b}, \hat{w})$. The training was done via the stochastic gradient descent method with the batch size 600, momentum 0, L2 regularization parameter 0.001, and the constant learning rate 0.001. Calculations were done in Wolfram Mathematica 12.3.1.0 by 16Gb Intel i7 4702MQ and 16Gb AMD Ryzen 7 5000 systems. Time measurements were done with the latter system.

5.1 Programmed death

For numerical testing of the programmed death algorithms, described in the previous section, we trained the feedforward neural networks N1, N2 and N3 for 50,000 epochs.

In Fig. 1, we plot the efficiency of neuron, E_k or E'_k, versus change in the average loss function, $\Delta\langle H\rangle$, (i.e., loss after neuron is removed minus loss before neuron is removed) for three different neural networks (N1, N2 and N3) and three different algorithms (A1, A2 and A3) at epochs 500 and 50,000. For the individual runs, efficiency of every neuron on the second layer is calculated, then each neuron is removed and the change in the loss function is calculated. Statistics is acquired by running fifty simulations with different initialization for every algorithm and neural architecture. In the third row (or for the neural

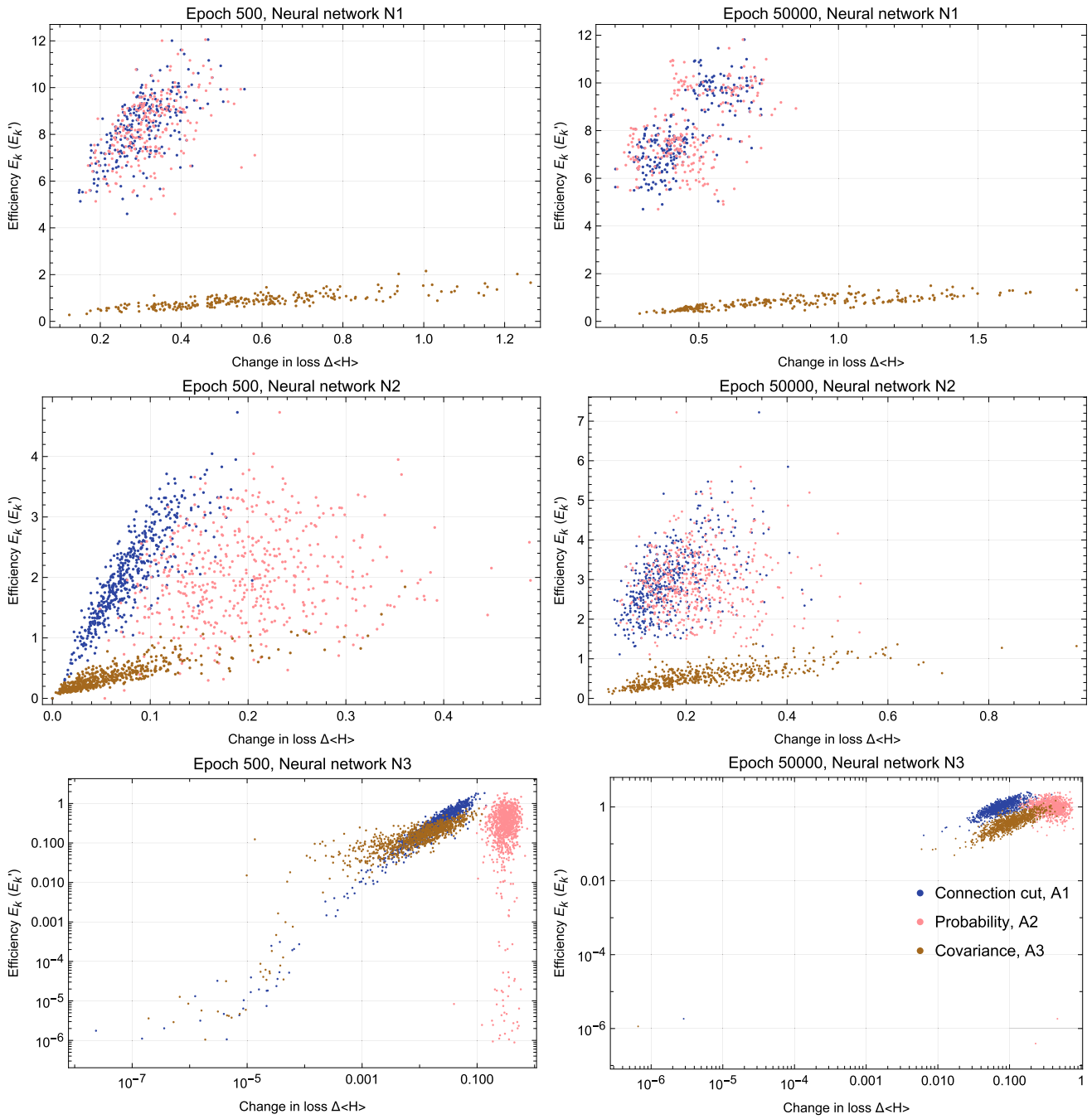

Fig. 1 Efficiency of neuron, E_k or E_k', versus change in the average loss function, $\Delta\langle H\rangle$, for algorithms A1 (first row), A2 (second row) and A3 (third row), and for neural architectures N1 (blue dots), N2 (pink dots) and N3 (brown dots) after 500 (left plots) and 50,000 (right plots) epochs

network N3) and for "connection cut" and "probability" plots (or for A1 and A3 algorithms) the log-log plot is used to show that there are many neurons with efficiency smaller than $\leq 10^{-3}$ at epoch 500, but only one such neuron at epoch 50000. In programmed death algorithms, such low-efficiency neurons can be removed without significantly changing the performance of the neural network.

In Fig. 2, we plot change in the average loss function, $\Delta\langle H\rangle$, as a function of time (or the number of epochs) for three different algorithms (A1, A2 and A3) and for three different neural architectures (N1, N2 and N3). For each run of the simulation every neuron on the second layer is removed and then $\Delta\langle H\rangle$ is calculated. For N1 neural architecture we executed 50 separate runs of the simulation, for N2 neural architecture—25 runs, and for N3 neural architecture—13 runs all with different initial conditions. The solid lines show the mean $\Delta\langle H\rangle$ averaged over all neurons on the second layer and all runs, and the dashed

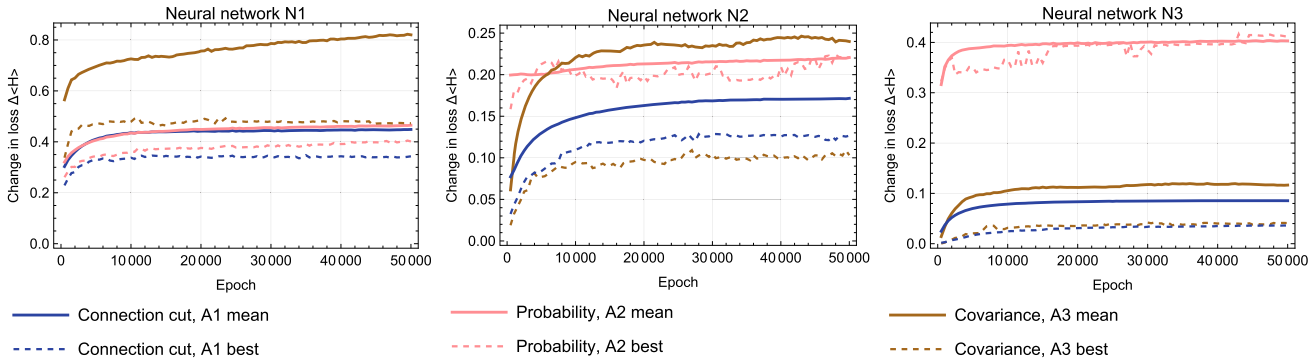

Fig. 2 $\Delta\langle H\rangle$ as a function of learning time for algorithms A1 (blue lines), A2 (pink lines) and A3 (brown lines) and for neural networks N1 (left plot), N2 (middle plot) and N3 (right plot)

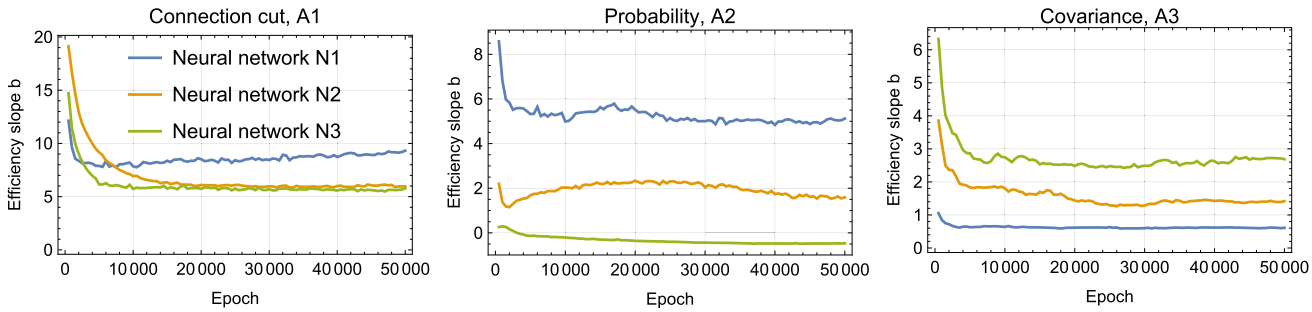

Fig. 3 Slopes $b(t)$ of the linear fits (of E_k (or E'_k) vs. $\Delta\langle H\rangle$) as a function of time t

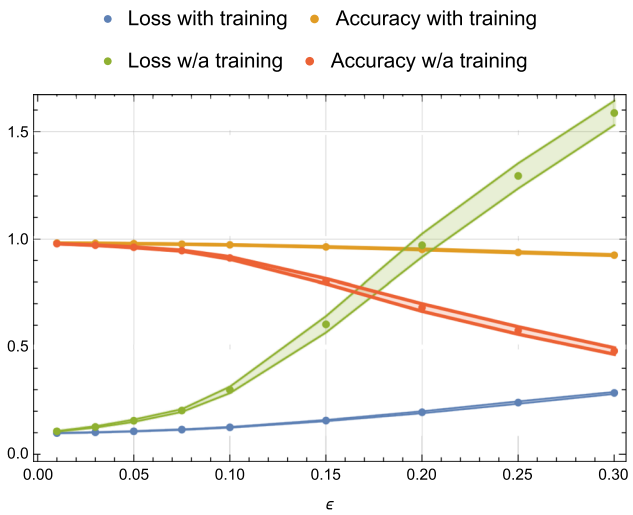

Fig. 4 Averaged results for 28 N4 nets trained for 5000 epochs to accuracy 0.97981 ± 0.00023 and pruned by A1 algorithm. Results are given after first pruning and after 500 epochs training and second pruning with the same cutoff

lines show the mean $\Delta\langle H\rangle$ for only the least efficient neuron on the second layer in each run averaged over all runs. The least efficient neuron is the one with the smallest efficiency, i.e., smallest E'_k for the A1 and smallest E_k for A2 or A3 algorithms. Clearly, only when the dotted line is much lower than the solid line (of the same color) the removal of the least efficient neuron would lead to the smallest distortion to the overall performance of the network. With this respect only for the neural networks N2 and N3, and only for the algorithm A2 (or pink lines) the corresponding probability method is not very useful.

Next, we make linear fits of E_k vs. $\Delta\langle H\rangle$ (for algorithms A1 and A2),

$$E_k = a + b\,\Delta\langle H\rangle, \tag{50}$$

or E'_k vs. $\Delta\langle H\rangle$ (for algorithm A3),

$$E'_k = a + b\,\Delta\langle H\rangle, \tag{51}$$

for all times. In Fig. 3, we plot the slopes $b(t)$ as a function of learning time, where the data are obtained from the same runs as for Fig. 2. Note that for algorithm A2 (second plot) and neural network N3 (green line) the slopes are almost zero (or slightly negative) and so the probability method is not very useful for identifying and removing a neuron which would give the smallest change in loss. In fact, according to Fig. 2, one should remove the least efficient neuron using algorithm A1 or A2 for neural network N1 and using algorithm A1 or A3 for neural networks N2 and N3. Moreover, according to Fig. 3, the algorithms A1 and A3 should work for all neural networks since the blue and yellow lines remain positive, which allows us to predict the effect of removing a given neuron, but the algorithm A2

Efficiency cutoff ϵ	Accuracy after pruning	Number of remaining neurons	Pruning time (sec)	Learning and second pruning time (sec)	Accuracy after learning	Accuracy after second pruning	Number of remaining neurons after second pruning
0.01	0.97840 ±0.0002	82.8 ±0.7	1.74 ±0.11	110.2 ±0.8	0.98113 ±0.00022	0.98112 ±0.00023	82.8 ±0.7
0.03	0.9713 ±0.0009	69.0 ±0.8	1.614 ±0.007	97.1 ±0.8	0.98003 ±0.00023	0.98001 ±0.00022	69.0 ±0.8
0.05	0.9616 ±0.0018	60.1 ±0.8	1.585 ±0.007	90.0 ±0.8	0.97868 ±0.00027	0.97868 ±0.00027	60.1 ±0.8
0.075	0.9461 ±0.0022	50.8 ±0.7	1.549 ±0.006	81.7 ±0.8	0.97635 ±0.00034	0.97635 ±0.00034	50.8 ±0.7
0.1	0.912 ±0.006	42.5 ±0.7	1.532 ±0.006	74.3 ±0.9	0.9730 ±0.0004	0.9730 ±0.0004	42.5 ±0.7
0.15	0.804 ±0.012	29.3 ±0.7	1.497 ±0.005	62.8 ±0.8	0.9634 ±0.0008	0.9634 ±0.0008	29.2 ±0.7
0.2	0.683 ±0.017	20.8 ±0.6	1.483 ±0.005	55.0 ±0.7	0.9519 ±0.0013	0.9517 ±0.0013	20.6 ±0.7
0.25	0.576 ±0.017	14.9 ±0.5	1.477 ±0.006	50.4 ±0.5	0.9381 ±0.0017	0.9379 ±0.0017	14.9 ±0.5
0.3	0.480 ±0.015	11.04 ±0.27	1.474 ±0.006	48.19 ±0.28	0.9251 ±0.0014	0.9251 ±0.0014	11.00 ±0.28

Fig. 5 Averaged results for 28 N4 nets trained for 5000 epochs to accuracy 0.97981 ± 0.00023 and pruned by A1 algorithm. Initial number of neurons on the second layer is 100. Maximal efficiency of the neuron on the second layer is 0.631 ± 0.026. After pruning the net was retrained for 500 epochs and pruned again with the same cutoff

Fig. 6 Averaged results for 28 N4 nets with 100 neurons on the second layer trained for 5000 epochs to accuracy 0.97981 ± 0.00023 and pruned by A1 algorithm with accuracy goal. First row gives results for pruning of the initial net. Next five rows give results for five iterations of retraining for 500 epochs and pruning again with the same accuracy goal

		Iteration time (sec.)	Accuracy before pruning	Accuracy after pruning	Number of remaining neurons after iteration
Accuracy goal	0.97	25.5 ±0.6	0.97981 ±0.00023	0.97060 ±0.00009	67.0 ±0.8
		102.1 ±1.1	0.97983 ±0.00018	0.97090 ±0.00015	58.6 ±0.8
		91.6 ±0.9	0.97995 ±0.00016	0.97136 ±0.00016	53.3 ±0.8
		85.8 ±0.9	0.98016 ±0.00015	0.97143 ±0.00023	49.5 ±0.7
		82.8 ±0.9	0.98022 ±0.00015	0.97203 ±0.00025	46.4 ±0.7
		80.0 ±0.7	0.98037 ±0.00014	0.97209 ±0.00026	44.0 ±0.6
	0.95	34.4 ±0.5	0.97981 ±0.00023	0.95213 ±0.00025	53.1 ±0.7
		90.5 ±1.0	0.97703 ±0.00023	0.9530 ±0.0004	43.1 ±0.5
		77.7 ±0.6	0.97533 ±0.00025	0.9530 ±0.0005	37.3 ±0.5
		71.3 ±0.6	0.97399 ±0.00024	0.9555 ±0.0006	33.7 ±0.6
		67.4 ±0.8	0.97325 ±0.00025	0.9569 ±0.0009	31.0 ±0.5
		65.7 ±0.6	0.97266 ±0.00024	0.9554 ±0.0007	28.8 ±0.5

would not work for neural network N3 since the green line on the rightmost plot remains near zero.

5.2 Pruning of the trained net

Pruning of the trained net is important for applications. We test our algorithm A1 on net N4 with 100 neurons in the second layer. In this section we trained 28 N4 nets with different initialization and applied algorithm A1 to reduce the number of neurons *only on the second layer*. Training was done for 5000 epochs. We choose A1 since from the analysis in the previous section we conclude that algorithm A1 works better than others since it is faster than A3 and more reliable than A2.

Efficiency cutoff for in neurons	$\frac{\text{cutoff}}{\text{Max efficiency}}$	Accuracy after pruning	Number of remaining input neurons	Pruning time (sec.)	Accuracy after retraining	Accuracy after retraining (validation)	Retraining time (sec.)
0.0001	0.00231 ±0.0000	0.9725 ±0.0004	479.5 ±1.0	3.79 ±0.11	0.9752 ±0.0004	0.9630 ±0.0005	247.8 ±1.2
0.0005	0.0116 ±0.0005	0.9706 ±0.0005	407.4 ±1.5	3.637 ±0.005	0.9741 ±0.0004	0.9625 ±0.0005	215.1 ±1.3
0.001	0.0231 ±0.0009	0.9685 ±0.0005	374.5 ±1.6	3.622 ±0.007	0.9730 ±0.0004	0.9621 ±0.0005	200.3 ±1.1
0.0025	0.0579 ±0.0023	0.9622 ±0.0006	319.8 ±2.1	3.576 ±0.005	0.9698 ±0.0004	0.9602 ±0.0005	176.5 ±1.4
0.005	0.116 ±0.005	0.9484 ±0.0011	272.4 ±2.4	3.536 ±0.005	0.9652 ±0.0006	0.9570 ±0.0007	156.4 ±1.4
0.0075	0.174 ±0.007	0.9244 ±0.0029	232.8 ±3.4	3.507 ±0.007	0.9593 ±0.0008	0.9524 ±0.0007	136.9 ±1.9
0.01	0.231 ±0.009	0.870 ±0.008	193. ±4.	3.470 ±0.005	0.9504 ±0.0012	0.9451 ±0.0013	117.5 ±2.0
0.015	0.347 ±0.014	0.592 ±0.029	121. ±5.	3.402 ±0.007	0.919 ±0.004	0.916 ±0.004	85.7 ±2.5
0.02	0.463 ±0.018	0.274 ±0.023	67. ±4.	3.349 ±0.010	0.846 ±0.012	0.846 ±0.011	62.4 ±1.8

Fig. 7 Averaged results for 28 N4 nets with initial neurons pruned by A1 algorithm with different efficiency cutoffs. Initial parameters of the nets are in line 5 in Fig. 5: accuracy is 0.9730 ± 0.0004, number of neurons on the second layer is 42.5 ± 0.7, second layer efficiency cutoff 0.1, maximal efficiency of the input neuron 0.0432 ± 0.0017. After pruning, the net was retrained for 500 epochs

There are several strategies to follow in pruning the trained net. One can introduce the cutoff ϵ for neuron efficiency (28) and eliminate all neurons with smaller efficiencies. Then, one can retrain and do the same. The results of such approach are in Fig. 4 and the table in Fig. 5.

One can see that learning after pruning improves accuracy but does not change the number of neurons more efficient than the cutoff, i. e. second pruning with the same cutoff does not reduce the number of neurons. Hence, it does not affect loss and accuracy.

Another strategy is to set an accuracy goal and remove neurons one by one while accuracy is greater than the goal. Once removing the next neuron makes accuracy less than the goal, one stops pruning and retrains the net. After retraining, one repeats pruning again. The results of this approach are given in Fig. 6 for accuracy goals 0.97 and 0.95.

Comparing Figs. 5 and 6, one can see that the first strategy is more efficient than the second. One gets a net with accuracy 0.95 and 21 neurons on the second layer via the first strategy with efficiency cutoff 0.2 and one step of retraining for 56 s. According to the second strategy, one gets a net with accuracy 0.95 and 29 neurons on the second layer with 5 steps of retraining for 407 s. For accuracy 0.97, the first strategy with cutoff 0.1 gives a net with 43 neurons for 76 s and the second strategy gives a net with 44 neurons for 468 s.

5.3 Pruning of input neurons in trained net

Input neurons may also be inefficient. Although we introduced our algorithms for neurons in the hidden layers, one can apply them to the input neurons as well. As in the previous section, one discards the neurons with the efficiency less than a cutoff or until the net's accuracy, e.g., is greater than the accuracy goal.

To demonstrate how algorithm A1 with efficiency cutoff works on the input neurons, we applied it to the nets obtained in the previous section after A1 with efficiency cutoffs 0.1 and 0.2 reduced the number of the second layer neurons (see Fig. 5 lines 5 and 7). The results with different efficiency cutoffs for the initial neurons are in Fig. 7 for the nets obtained with the second layer cutoff 0.1. The input neurons eliminated for the cutoffs in Fig. 7 are shown in Fig. 8.

The results for the second layer cutoff 0.2 with different initial neuron efficiency cutoffs are in Fig. 9.

We also give accuracies for the validation set demonstrating that net reduction keeps accuracy on the unknown data.

Comparing the net's sizes in Figs. 7–9 for similar accuracies ~ 0.95, one finds the net with 193 input neurons and 43 second layer neurons in line 7 (in cutoff 0.01, 2nd layer cutoff 0.1) in Fig. 7 and the net with 337 input neurons and 21 second layer neurons in line 3 (in cutoff 0.001, second layer cutoff 0.2) in Fig. 9. These nets have $193 \times 43 = 8299$ and $337 \times 21 = 7077$ weights connecting input and second layer neurons. So in the situation with the large number of input neurons reduction of the hidden neurons, i.e., large efficiency cutoff for them gives smaller nets than the reduction of the input neurons, i.e., large efficiency cutoffs for them.

Figures 7 and 9 show that cutoff $\sim 0.05\times$(Max efficiency) reduces accuracy for $\sim 1\%$, which is recovered after retraining.

To demonstrate how algorithm A1 with accuracy goal works on the input neurons we applied it to the nets

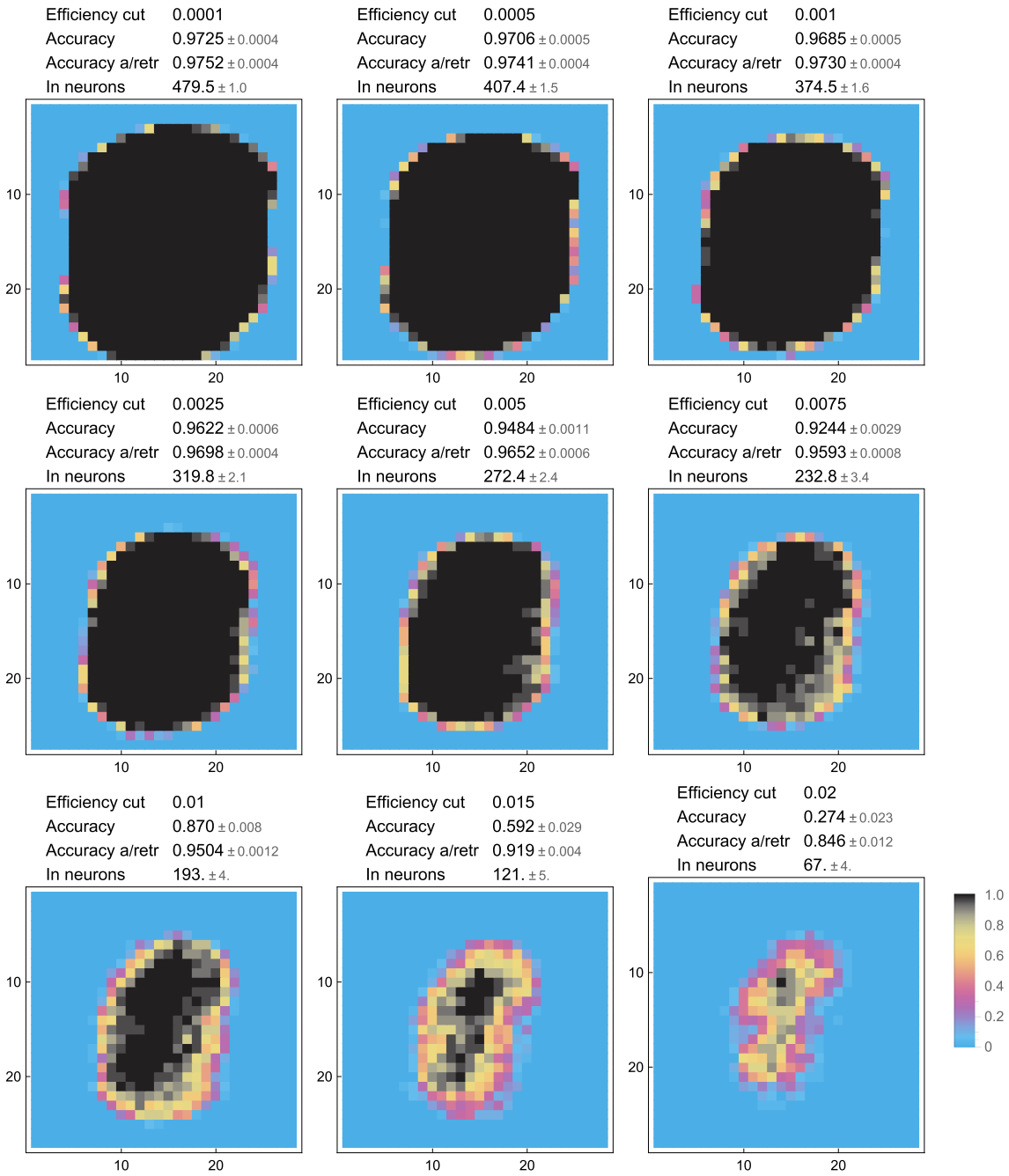

Fig. 8 Input neurons removed from the 28 × 28 pixel field by A1 algorithm with nine efficiency cutoffs. Nets have parameters from Fig. 7. Averaged results for 28 N4 nets. Black neurons were not removed in all 28 runs, and blue neurons were always removed in all 28 runs

obtained after A1 was applied to the neurons on the second layer with accuracy goals 0.97 and 0.95 after 5 steps of retraining (see Fig. 6). The results for the same accuracy goals are in Fig. 10.

One can see that 45–50%% of input neurons are inefficient and are removed by the algorithm. Figure 11 shows that they are the neurons in the boundary of the field.

The results in Fig. 10 show accuracies after pruning without retraining. Comparing the size of the net obtained with the accuracy goal 0.97 $436 \times 44 = 19184$ (Fig. 10) and the size of net obtained with cutoffs for the initial and second layer neurons 0.0005 and 0.1 (Fig. 7), $407 \times 43 = 17501$, one can see that the cutoff based approach is more effective. The results are similar for the nets with accuracy 0.95, see Fig. 7 and 10. Timewise cutoff pruning takes about 3.5 s while accuracy goal based pruning takes about 450 s.

Efficiency cutoff for in neurons	$\frac{\text{cutoff}}{\text{Max efficiency}}$	Accuracy after pruning	Number of remaining input neurons	Pruning time (sec.)	Accuracy after retraining	Accuracy after retraining (validation)	Retraining time (sec.)
0.0001	0.00359 ± 0.0002	0.9507 ± 0.0014	451.2 ± 2.0	3.615 ± 0.005	0.9559 ± 0.0013	0.9479 ± 0.0010	223.8 ± 1.3
0.0005	0.0180 ± 0.0011	0.9472 ± 0.0015	375.9 ± 2.3	3.573 ± 0.009	0.9537 ± 0.0013	0.9462 ± 0.0011	192.0 ± 1.1
0.001	0.0359 ± 0.0022	0.9429 ± 0.0016	336.7 ± 3.1	3.536 ± 0.007	0.9515 ± 0.0014	0.9446 ± 0.0012	177.0 ± 1.5
0.0025	0.090 ± 0.006	0.922 ± 0.004	266. ± 5.	3.478 ± 0.006	0.9450 ± 0.0017	0.9395 ± 0.0016	147.5 ± 2.3
0.005	0.180 ± 0.011	0.780 ± 0.025	177. ± 8.	3.410 ± 0.009	0.9266 ± 0.0033	0.9236 ± 0.0030	110. ± 4.
0.0075	0.269 ± 0.017	0.57 ± 0.04	117. ± 8.	3.367 ± 0.012	0.891 ± 0.008	0.890 ± 0.007	83.7 ± 3.2
0.01	0.359 ± 0.022	0.37 ± 0.04	74. ± 7.	3.308 ± 0.013	0.829 ± 0.015	0.830 ± 0.015	67.0 ± 2.9
0.015	0.539 ± 0.033	0.140 ± 0.011	29.8 ± 3.4	3.256 ± 0.008	0.682 ± 0.021	0.687 ± 0.021	50.7 ± 1.1
0.02	0.72 ± 0.04	0.1008 ± 0.0012	15.9 ± 1.1	3.250 ± 0.009	0.578 ± 0.012	0.584 ± 0.012	46.72 ± 0.25

Fig. 9 Averaged results for 28 N4 nets with initial neurons pruned by A1 algorithm with different efficiency cutoffs. Initial parameters of the nets are in line 7 in Fig. 5: accuracy is 0.9517 ± 0.0013, number of neurons on the second layer is 20.6 ± 0.7, second layer efficiency cutoff 0.2, maximal efficiency of the input neuron 0.0278 ± 0.0017. After pruning the net was retrained for 500 epochs

Fig. 10 Averaged results for 28 N4 nets after elimination of input neurons with A1 algorithm with accuracy goal. Initial nets have parameters from Fig. 6

		Iteration time (sec.)	Accuracy before pruning	Accuracy after pruning	Number of remaining input neurons	Number of neurons on second layer
Accuracy goal	0.97	426. ± 11.	0.97209 ± 0.0003	0.970090 ± 0.00(	436. ± 8.	44.0 ± 0.6
	0.95	468. ± 11.	0.9554 ± 0.0007	0.950102 ± 0.00(	391. ± 8.	28.8 ± 0.5

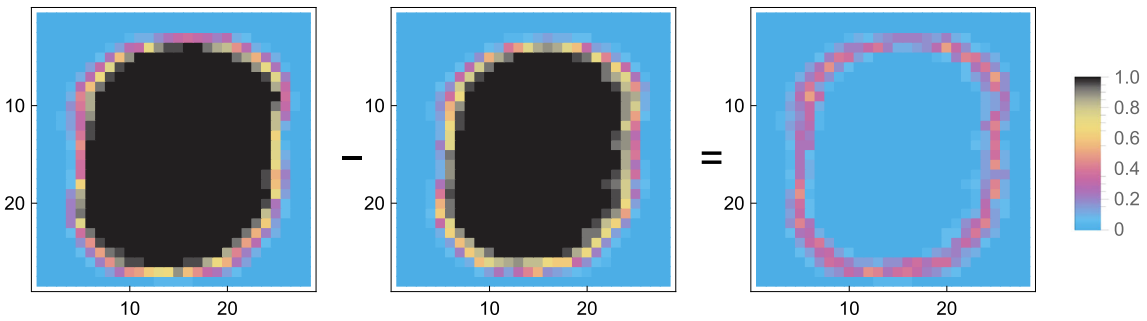

Fig. 11 Inefficient input neurons removed from the 28 × 28 pixel field by A1 algorithm with accuracy goals 0.97 (left) and 0.95 (center) and the neurons which were not necessary for accuracy 0.95 but were necessary for accuracy 0.97 (right). In Left and Center plots black neurons were not removed in all 28 runs and blue neurons were always removed in all 28 runs. Nets have parameters from Fig. 10. Averaged results for 28 N4 nets

5.4 Pruning of untrained net

One may also try pruning the input neurons in the untrained net since inefficient input neurons with zero variance will have zero efficiency independently of the weights. One can see the computational advantage of pruning before training in Fig. 12, where we use the A1 algorithm to prune the initial neurons in 28 N4 nets. The removed input neurons are shown in Fig. 13.

One can see that the input neurons in Fig. 13 removed before training are mainly the same as the ones removed after training depicted in Fig. 11.

Hidden layer neurons may also be inefficient upon initialization. If they have small efficiency from the start, then not only do they have small influence on the output but also they need more steps of learning to change the output. It means that their learning is inefficient, i.e., the gradients with respect to change in their weights are small. To test how pruning of the hidden neurons in the untrained

Efficiency cut %	Training time (min)	Accuracy	Number of input neurons
0	71.80 ± 0.20	0.97985 ± 0.00023	784
0.1	56.45 ± 0.27	0.97977 ± 0.00024	594.4 ± 0.5
1.0	48.81 ± 0.21	0.97945 ± 0.00025	502.0 ± 0.6
5.0	41.62 ± 0.23	0.97863 ± 0.00025	419.5 ± 0.8

Fig. 12 Averaged results for 28 N4 nets after elimination of input neurons with A1 algorithm with different efficiency cutoffs (% of max efficiency) for input neurons applied to the untrained net and trained for 5000 epochs after that. Average accuracy of the initial untrained nets is 0.099 ± 0.004. It changed within the error bars after initial pruning with all three cutoffs. Number of neurons on the second layer is 100

networks, we use the A1 algorithm to prune both the initial neurons and the second layer neurons in 28 N4 nets with the same cutoff in the presence of maximal neuron efficiency calculated on each layer separately. The results are given in Fig. 14.

One can see that although there is little change in training time with respect to the nets with larger number of neurons on the hidden layer, initial pruning of both the input and the hidden neurons leads to smaller nets without much loss in accuracy after training.

5.5 Comparison with other approaches

We will compare our algorithm A1 with one local algorithm, the N2PS algorithm [18], and one global algorithm, NNSP [17]. N2PS algorithm was introduced for a feed-forward net with the sigmoidal activation function and the mean squared error. It uses the sigmoidal function $\frac{1}{1+e^{-x}}$ to define the neuron's significance. Therefore, to compare with it we trained 28 N4 nets with the sigmoidal activation function instead of $th(x)$ and applied the pruning algorithms. The results are in Fig. 15.

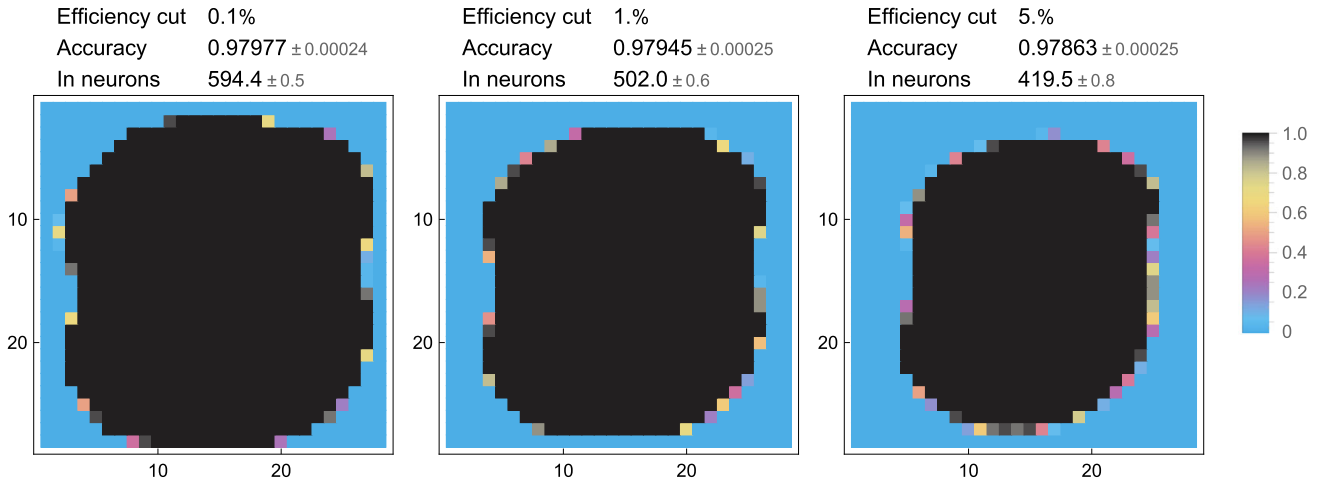

Fig. 13 Inefficient input neurons removed from the 28 × 28 pixel field by A1 algorithm with different efficiency cutoffs (% of max efficiency) for input neurons applied to the untrained net and trained for 5000 epochs after that. Black neurons were not removed in all 28 runs, and blue neurons were always removed in all 28 runs. Nets have parameters from Fig. 12. Averaged results for 28 N4 nets

Efficiency cut %	Training time (min)	Accuracy	Number of input neurons	Number of 2–nd layer neurons
0.1	56.80 ± 0.22	0.97975 ± 0.00024	594.4 ± 0.5	98.93 ± 0.19
1.0	48.46 ± 0.15	0.97925 ± 0.00026	502.0 ± 0.6	92.2 ± 0.6
5.0	42.2 ± 1.3	0.97748 ± 0.00025	419.5 ± 0.8	74.8 ± 1.1

Fig. 14 Averaged results for 28 N4 nets after elimination of input and second layer neurons with A1 algorithm with different efficiency cutoffs (% of max efficiency on the layer) for input and second layer neurons applied to the untrained net and trained for 5000 epochs after that. Average accuracy of the initial untrained nets is 0.099 ± 0.004

Method	Pruning time (sec)	Accuracy	Number of input neurons	Number of 2–nd layer neurons
N2PS	3.231 ±0.005	0.248 ±0.018	393.5 ±3.1	49.1 ±0.8
N2PS with tx_{ip}->$\frac{tx_{ip}}{np}$	3.324 ±0.008	0.122 ±0.005	509.6 ±0.7	49.1 ±0.8
N2PS with tx_{ip}->$\frac{tx_{ip}}{np}$ and keeping insignificant input neurons	3.125 ±0.005	0.273 ±0.016	274.4 ±0.7	49.1 ±0.8
A1 1% cutoff	3.927 ±0.004	0.756 ±0.008	425.5 ±1.7	95.0 ±0.4
A1 10% cutoff	3.781 ±0.008	0.682 ±0.014	285. ±4.	54.4 ±1.6

Fig. 15 Averaged results for 28 N4 nets with the sigmoidal activation functions after elimination of input and second layer neurons with (Line 1) N2PS algorithm, (Line 2) N2PS algorithm with total net input value changed to average input value for each neuron in the significance calculation (52), (Line 3) N2PS algorithm with total net input value changed to average input value for each neuron in the significance calculation and elimination of input neurons with the significance greater than the average, (Lines 4–5) A1 algorithms with different efficiency cutoffs (% of max efficiency on the layer) for input and second layer neurons. Average accuracy of the initial nets is 0.8695 ± 0.0023

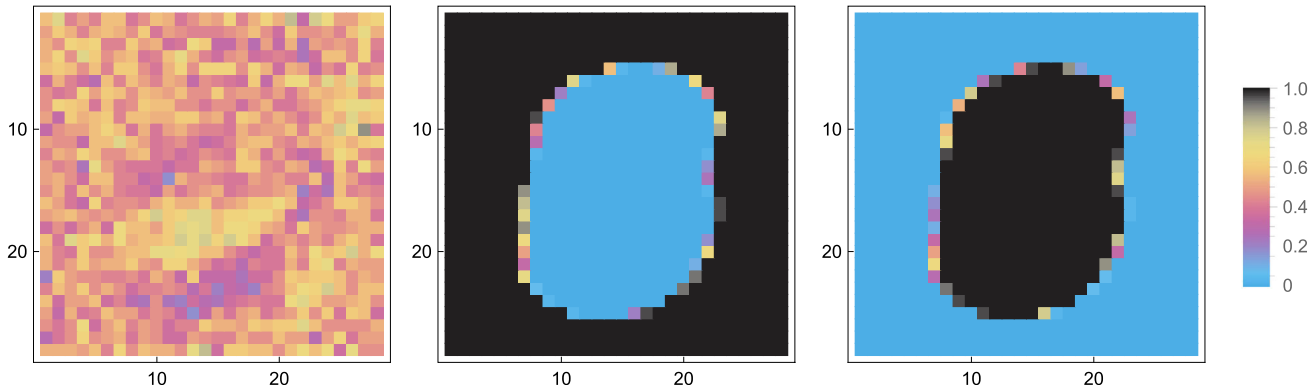

Fig. 16 Input neurons removed from the 28×28 pixel field by (left) N2PS algorithm, (center) N2PS algorithm with total net input value changed to average input value for each neuron in the significance calculation (52), (right) N2PS algorithm with total net input value changed to average input value for each neuron in the significance calculation and elimination of input neurons with the significance greater than the average. Averaged results for 28 N4 nets. Black neurons were not removed in all 28 runs and blue neurons were always removed in all 28 runs

One can understand the results of Fig. 15 looking at the N2PS criterion to prune the neuron. They define the significance of the input neuron i as

$$s_i = \sum_{j_1=1}^{m_1} \left| f(tx_{ip}) + w_{ij_1} \right|, \quad tx_{ip} = \sum_{p=1}^{np} x_{ip}, \quad f(x) = \frac{1}{1+e^{-x}}, \tag{52}$$

where m_1 is the number of neurons on the second layer, x_{ip} is the input neuron's value on the example p from the dataset with np examples, and tx_{ip} is the total net value of input neuron i on the dataset. The neuron is pruned if its significance is less than the average significance of the input neurons.

In MNIST dataset, there are 60000 examples. The input neurons in the corner of the pixel field are always white, i.e., have $x_{ip} = 1$ and the input neurons in the center of the field are sometimes black, i. e. 0, with the average value $\langle x_{ip} \rangle \geq 0.5$. Therefore the difference in $f(tx_{ip})$ in significance of the central neurons and the boundary neurons is $\sim e^{-30000} - e^{-60000}$, i.e., 0 compared to the weights ~ 1. It means that the algorithm will differentiate the input neurons only on the basis of the weights ignoring the input values altogether. And the weights are similarly distributed for all the neurons. Therefore for large datasets the input neurons will be randomly pruned. One can see it from Fig. 16 and from the results in Fig. 15.

Moreover, for smaller datasets or using the average input value instead of the total in (52), the always white neurons from the boundary will have greater significance than the neurons in the center of the pixel field. Indeed, they have greater $f(tx_{ip})$ and the weights are similarly distributed for all the neurons. It means that the algorithm will eliminate the central input neurons keeping the boundary input neurons in the net. One can see it in Fig. 16. If, however, we change the color coding making

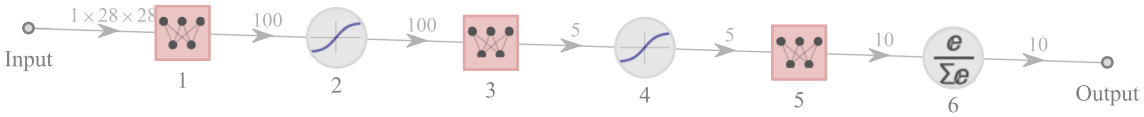

Fig. 17 Structure of the net N4. Elements 1, 3, 5 are linear layers with weights $w^{1,3,5}$ and biases. Elements 2, 4, 6 are activation functions: 2 and 4 are $th(x)$ and 6 is a softmax layer $x_i \to \frac{e^{x_i}}{\sum_k e^{x_k}}$. Numbers above the arrows stand for the number of output neurons of the corresponding layers

white 0 and black 1, the N2PS algorithm will eliminate the boundary neurons keeping the central ones.

Our algorithms have the following advantages. They work for any differentiable activation function, not only the sigmoidal. Next, they estimate neurons not on the neurons' values but on their (co)variances. Therefore, they are independent of the input encoding and the size of the dataset.

The NNSP algorithm was introduced for a feedforward net with an input, output layers and one hidden layer (Eq. (3) in [17])

$$\hat{y} = g_o\left(\sum_{h=1}^{n_h} w_h^2.g_h\left(\sum_{i=1}^{n_i} w_{hi}^1.x_i + b_h^1\right) + b\right), \tag{53}$$

where g_o and g_h are the output and the hidden layer activation functions, w_h^2 and w_{hi}^1 are the weights connecting the hidden neurons with the output and input, and b and b_h^1 are the corresponding biases, n_h and n_i are the numbers of hidden and input neurons. Such a construction assumes one output neuron. This algorithm prunes input neurons and hidden weights and biases. To this end, it introduces the sensitivities as partial derivatives of the net's output with respect to the input neurons' values, hidden weights and biases. For the input neuron i, e.g., the sensitivity S_{x_i} reads (Eq. (7) in [17])

$$S_{x_i}(n) = \frac{\partial\hat{y}(n)}{\partial x_i(n)} = \sum_{h=1}^{n_h} g_o'(z(n)).w_h^2.(1-(x_h^1(n))^2)w_{hi}^1, \tag{54}$$

$$x_h^1(n) = g_h\left(\sum_{i=1}^{n_i} w_{hi}^1.x_i(n) + b_h^1\right), \quad g_h(x) = th(x) \quad \Rightarrow \tag{55}$$

$$g_h' = 1 - g_h^2 = 1 - (x_h^1(n))^2. \tag{56}$$

Here, n is the number of the pattern from the learning dataset. Then, one has to find variance of this sensitivity over the whole dataset

$$\hat{\sigma}_{x_i}^2 = \langle(S_{x_i} - \langle S_{x_i}\rangle)^2\rangle \tag{57}$$

and prune neurons with $\hat{\sigma}_{x_i}^2$ less than a cutoff checking that the net's performance does not deviate too much from the goal.

To apply this algorithm to our problem, we have to adjust it since our net N4 is a net with ten outputs and two hidden layers, as is shown in Fig 17.

First, we use the sum of sensitivity variances (57) for all the output channels as the measure of the neuron's efficiency. Next, we can choose what to consider the output channels $\hat{y}$. We can take as output the ten output neurons of layer 6 in Fig. 17 and modify the expression after the second equality sign in (54) accordingly. It is the global approach. Or, we can take as output the neurons going out of the first several levels of the N4 net if we want to study the sensitivity to input neurons. It is the local approach. It is interesting to see how the pruning quality changes with choice of the output.

Here, we will compare A1 and NNSP algorithms in input neuron pruning. We cannot choose the neurons going out of the first layer as output $\hat{y}$ since then the sensitivities are the weights w^1, which have zero $\hat{\sigma}_{x_k}$. Then, we start with the hundred neurons going out of layer 2 in Fig. 17 as output $\hat{y}$. The 100 sensitivities $S_{x_k}^i$, $i = 1, 2, ..100$ of the 784 input neurons x_k read

$$S_{x_k}^i\big|_2 = (1-(x_i^2)^2)w_{ik}^1. \tag{58}$$

Here, x_i^2 are the output values of level 2 in Fig. 17 and ^{2} denotes the layer rather than a power. Therefore the sum of the variances of these sensitivities gives the usefulness measure of the input neuron. It reads

$$\begin{aligned}\hat{\sigma}_{x_k}^2\big|_2 &= \sum_{i=1}^{100}\langle(S_{x_k}^i - \langle S_{x_k}^i\rangle)^2\rangle\big|_2 = \sum_{i=1}^{100}(\langle(S_{x_k}^i)^2\rangle - \langle S_{x_k}^i\rangle^2)\big|_2 \\ &= \sum_{i=1}^{100}(\langle(x_i^2)^4\rangle - \langle(x_i^2)^2\rangle^2)(w_{ik}^1)^2.\end{aligned} \tag{59}$$

Hence, one has to measure the 100 variances of the squares of the output of the second layer in Fig. 17.

Next, we choose the five neurons going out of layer 3 in Fig. 17 as output $\hat{y}$. Then the 5 sensitivities $S_{x_k}^i$, $i = 1, 2, ..5$ of the 784 input neurons read

$$S_{x_k}^i\big|_3 = \sum_{l=1}^{100} w_{il}^3(1-(x_l^2)^2)w_{lk}^1. \tag{60}$$

Here x_l^2 are the output values of level 2 in Fig. 17 and ^{2} denotes the layer rather than a power. Therefore the sum of

the variances of these sensitivities gives the usefulness measure of the input neuron. It reads

$$\hat{\sigma}^2_{x_k}\big|_3 = \sum_{i=1}^{5}\sum_{l,r=1}^{100}(\langle (x_l^2)^2(x_r^2)^2\rangle - \langle (x_l^2)^2\rangle\langle (x_r^2)^2\rangle)w_{il}^3 w_{ir}^3 w_{lk}^1 w_{rk}^1. \tag{61}$$

Hence, one has to measure the covariance matrix of the squares of the output of the second layer in Fig. 17 with 100^2 elements.

Next, we choose the five neurons going out of layer 4 in Fig. 17 as output $\hat{y}$. Then, the 5 sensitivities $S^i_{x_k}$, $i = 1, 2, ..5$ of the 784 input neurons read

$$S^i_{x_k}\big|_4 = \sum_{l=1}^{100}\underbrace{(1-(x_i^4)^2)}_{z_i}w_{il}^3\underbrace{(1-(x_l^2)^2)}_{u_l}w_{lk}^1. \tag{62}$$

Here x_l^2 and x_i^4 are the output values of levels 2 and 4 in Fig. 17 and ^{2,4} denote the layer rather than a power and we introduced z_i and u_l to shorten notation. The sum of the variances of these sensitivities gives usefulness of the input neuron x_k. It reads

$$\hat{\sigma}^2_{x_k}\big|_4 = \sum_{i=1}^{5}\sum_{l,r=1}^{100}(\langle z_i^2 u_r u_l\rangle - \langle z_i u_l\rangle\langle z_i u_r\rangle)w_{il}^3 w_{ir}^3 w_{lk}^1 w_{rk}^1. \tag{63}$$

Here one has to measure the covariance matrices $\langle z_i^2 u_r u_l\rangle$ and $\langle z_i u_r\rangle$ of the neurons on levels 2 and 4 in Fig. 17 with 5×100^2 and 5×100 elements.

We can choose the ten neurons going out of layer 5 in Fig. 17 as output $\hat{y}$. Then, the ten sensitivities $S^i_{x_k}$, $i = 1, 2, ..10$ of the 784 input neurons read

$$S^i_{x_k}\big|_5 = \sum_{j=1}^{5}\sum_{l=1}^{100}w_{ij}^5\underbrace{(1-(x_j^4)^2)}_{z_j}w_{jl}^3\underbrace{(1-(x_l^2)^2)}_{u_l}w_{lk}^1. \tag{64}$$

The sum of the variances of these sensitivities giving usefulness of the input neuron x_k reads

$$\begin{aligned}\hat{\sigma}^2_{x_k}\big|_5 = &\sum_{i=1}^{10}\sum_{j,n=1}^{5}\sum_{l,r=1}^{100}(\langle z_j z_n u_r u_l\rangle - \langle z_j u_l\rangle\langle z_n u_r\rangle)\\ &\times w_{ij}^5 w_{in}^5 w_{jl}^3 w_{nr}^3 w_{lk}^1 w_{rk}^1.\end{aligned} \tag{65}$$

Here one has to measure the covariance matrix $\langle z_j z_n u_r u_l\rangle$ of the neurons on levels 2 and 4 in Fig. 17 with $5^2\times 100^2$ elements.

Finally, we can choose the ten neurons going out of layer 6 in Fig. 17 as output $\hat{y}$. Using the softmax $y_q = \frac{e^{x_q}}{\sum_k e^{x_k}}$ derivative

$$\frac{\partial y_q}{\partial x^i} = \frac{\partial}{\partial x^i}\frac{e^{x_q}}{\sum_k e^{x_k}} = y_i(\delta_{qi} - y_q), \quad \delta_{qi} = \begin{cases}1, q = i\\ 0, q \neq i,\end{cases} \tag{66}$$

ten sensitivities $S^i_{x_k}$, $i = 1, 2, ..10$ of the 784 input neurons read

$$\begin{aligned}S^i_{x_k}\big|_6 = &\sum_{q=1}^{10}\sum_{j=1}^{5}\sum_{l=1}^{100}x_i^6(\delta_{qi}\\ &-x_q^6)w_{qj}^5\underbrace{(1-(x_j^4)^2)}_{z_j}w_{jl}^3\underbrace{(1-(x_l^2)^2)}_{u_l}w_{lk}^1.\end{aligned} \tag{67}$$

The sum of these sensitivities variances giving usefulness of the input neuron x_k reads

$$\begin{aligned}\hat{\sigma}^2_{x_k}\big|_6 = &\sum_{i,q,p=1}^{10}\sum_{j,n=1}^{5}\sum_{l,r=1}^{100}(\langle z_j z_n u_r u_l (x_i^6)^2(\delta_{qi}-x_q^6)(\delta_{pi}-x_p^6)\rangle\\ &-\langle z_j u_l x_i^6(\delta_{qi}-x_q^6)\rangle\langle z_n u_r x_i^6(\delta_{pi}\\ &-x_p^6)\rangle)w_{qj}^5 w_{pn}^5 w_{jl}^3 w_{nr}^3 w_{lk}^1 w_{rk}^1.\end{aligned} \tag{68}$$

Algorithm (output layer)	Accuracy after pruning	Number of remaining input neurons	Pruning time (sec.)	Accuracy after retraining	Retraining time (sec.)
A1	0.97568 ±0.00030	379.5 ±1.4	3.7485 ±0.0030	0.97925 ±0.00024	231.8 ±1.2
NNSP(2)	0.59 ±0.04	643. ±11.	3.013 ±0.012	0.98090 ±0.00033	369. ±5.
NNSP(3)	0.150 ±0.007	422. ±4.	5.761 ±0.008	0.9717 ±0.0007	252.5 ±1.8
NNSP(4)	0.221 ±0.009	514. ±6.	15.062 ±0.014	0.9774 ±0.0005	295.9 ±3.4
NNSP(5)	0.161 ±0.008	306. ±15.	59.17 ±0.08	0.9636 ±0.0018	191. ±7.

Fig. 18 Performance of NNSP algorithm with output layers 2–5 in Fig. 17 for input neurons' pruning compared to performance of the A1 algorithm. NNSP algorithm pruned input neurons with $\hat{\sigma}^2_{x_k} < 0.1 \max \hat{\sigma}^2_{x_k}$, A1 algorithm pruned input neurons with efficiencies less than $0.05 \times max(efficiency)$ cutoff. Averaged results for 28 N4 nets. Initial nets were trained for 5000 epochs to accuracy 0.97981 ± 0.00023. Retraining was done for 500 epochs

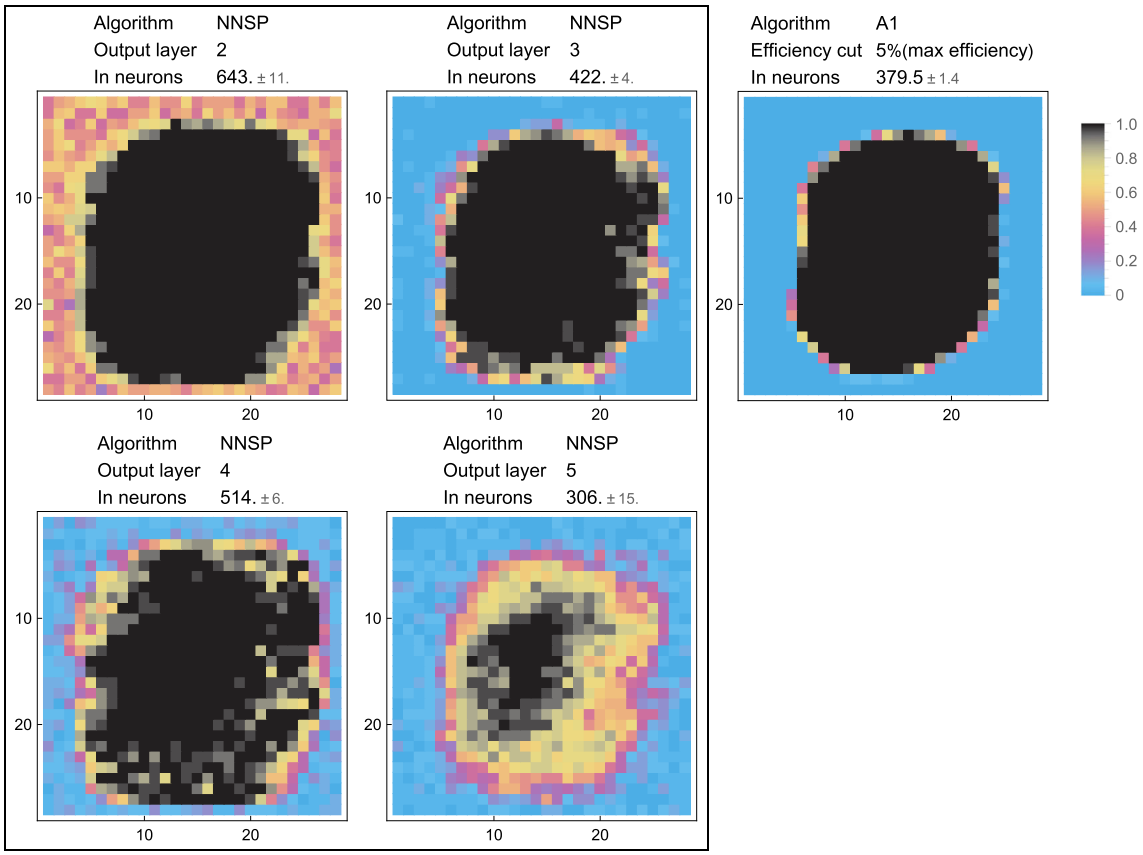

Fig. 19 Input neurons removed from the 28×28 pixel field by (left) NNSP algorithm with output taken at layers 2–5 in Fig. 17 where the input neurons with $\hat{\sigma}^2_{x_k} < 0.1 \max \hat{\sigma}^2_{x_k}$ were removed, (right) A1 algorithm with $0.05 \times max(efficiency)$ cutoff. Averaged results for 28 N4 nets. Black neurons were not removed in all 28 runs and blue neurons were always removed in all 28 runs. Initial nets were trained for 5000 epochs to accuracy 0.97981 ± 0.00023

Here, one has to measure the covariance matrix $\sum_{i=1}^{10} \langle z_j z_n u_r u_l (x_i^6)^2 x_q^6 x_p^6) \rangle$ of the neurons on levels 2, 4 and 6 in Fig. 17 with $5^2 \times 100^2 \times 10^2$ elements.

One can clearly see the increase in the computational demands to implement the NNSP algorithm as we choose the output closer to the final layer. We pruned the input neurons in 28 N4 nets according to this algorithm eliminating the input neurons with $\hat{\sigma}^2_{x_k} < 0.1 \max \hat{\sigma}^2_{x_k}$ for $\hat{\sigma}^2_{x_k}$ calculated on layers 2–5 in Fig. 17. Level 6 was beyond our computational resources. The results are given in Fig. 18.

The corresponding input pixel fields are shown in Fig. 19.

These figures also give results for the performance of our algorithm A1 for comparison.

Figure 18 shows that A1 and NNSP have comparable results in terms of number of pruned input neurons and accuracy after retraining. However, pruning time grows rapidly with the number of the output layer for NNSP algorithm from 3 to 60 s while A1 pruning time is about 3.7 s, which is comparable to NNSP with output layers 2 and 3. For layer 6 as output, one can estimate pruning time $\sim 100 \times 60$ seconds as the covariance matrix becomes 100 times bigger. However, this is exactly the logic of all global algorithms, which estimate usefulness of net's elements by the whole net's output. In this respect our algorithms are faster and independent of the number of layers in the net and the whole net's output. Thus for input neuron pruning in N4 net, algorithm A1 needs 784 variances of input neuron values and 100 variances of second layer neuron values. This number of measurements is ~ 8 times bigger than NNSP with output at layer 2, ~ 10 times smaller than NNSP with output at layer 3, ~ 60 times smaller than NNSP with output at layer 4, ~ 280 times smaller than NNSP with output at layer 5, and $\sim 28,000$ times smaller than NNSP with output at layer 6.

Figures 18 and 19 also show that NNSP algorithm does not need to consider the final layer as output since it eliminates the unnecessary input neurons with layers 2, 3, 4, and 5 as output, i.e., when applied locally. Figure 19 tells us that the eliminated neurons are on the boundary of the pixel field and Fig. 18 tells us that after retraining the net's accuracy does not deteriorate. Surely, on can choose

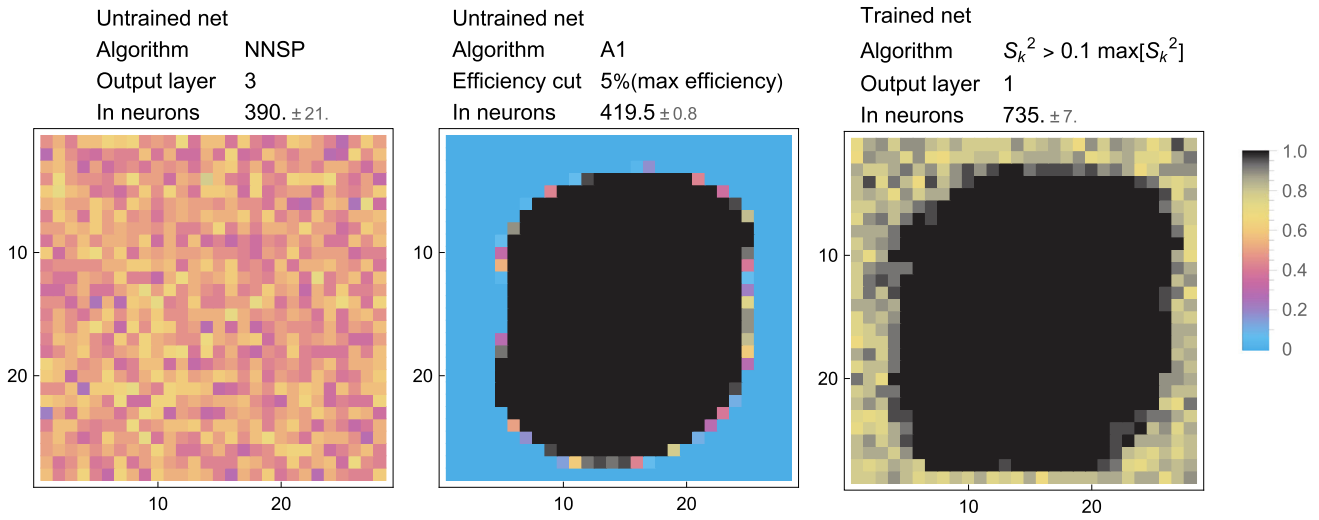

Fig. 20 Input neurons removed from the 28 × 28 pixel field of the untrained net (left) by NNSP algorithm with output taken at layer 3 in Fig. 17 where the input neurons with $\hat{\sigma}^2_{x_k} < 0.5 \max \hat{\sigma}^2_{x_k}$ were removed, (center) by A1 algorithm with $0.05 \times max(efficiency)$ cutoff. (right) Input neurons with $S^2_{x_k}|_1 < 0.1 \max S^2_{x_k}|_1$ were removed from the trained net with accuracy 0.97981 ± 0.00023. Averaged results for 28 N4 nets. Black neurons were not removed in all 28 runs and blue neurons were always removed in all 28 runs

the threshold for $\hat{\sigma}^2_{x_k}$ differently at each layer as output and vary the number of eliminated input neurons. This observation may indicate that global algorithms do extra calculations propagating the information about input neuron usefulness further along the net.

Figure 18 gives uncertainties for the number of eliminated neurons. For A1 algorithm, it is from 3 to 10 times smaller than for the NNSP algorithm with different output layers. It indicates that our algorithm is much more stable with respect to net initialization and learning.

One can understand it in the following way. First, our algorithm measures the variance of the input neuron's value. If it is zero this neuron will be eliminated whatever its weights are. NNSP algorithm measures variance of the sensitivity of the output to the input neuron, i.e., variance of the partial derivatives of the output with respect to the input neuron's value. However, if the input neuron's value does not change, it is not important whether this partial derivative together with its variance are big or small. In other words the sensitivity to this neuron may be big and varying but the neuron has always constant value and may be substituted by a bias and eliminated from the net. Nevertheless, Fig. 19 shows that NNSP algorithm prunes the neurons on the boundary of the pixel field although the transition from the boundary unnecessary neurons to the central always useful neurons is very wide compared to the A1 algorithm. It means that the sensitivities to the boundary input neurons became smaller than the average through learning, i.e., the net via training made the weights connected to the boundary neurons smaller than average. Since learning is a stochastic process, these weights became smaller in general but varying from neuron to neuron. As a result, we see wide boundaries between the central and the peripheral input neurons in Fig. 19.

This statement can be tested via pruning the untrained net. Figure 20 shows the input neurons removed from the 28 N4 untrained nets by NNSP algorithm with output layer 3 (left) and by A1 algorithm (center). One can see that NNSP algorithm does not work on the untrained net, while A1 works although it eliminates 40 neurons less than on the trained net in Fig. 19. Another way to prove this statement is to prune input neurons in the trained net with sensitivities measured on layer 1 in Fig. 17 less than a cutoff, i.e.,

$$S^2_{x_k}|_1 = \sum_{i=1}^{100} (S^i_{x_k})^2|_1 = \sum_{i=1}^{100} (w^1_{ik})^2 < 0.1 \max_k S^2_{x_k}|_1. \tag{69}$$

The result is given in Fig. 20 (right). One can see that while training the net learns that the neurons in the boundary are unnecessary. Therefore, the better is the net's accuracy the better is the NNSP algorithm performance.

The conclusion is that our algorithm can prune input neurons in the untrained net while NNSP cannot.

Comparison of hidden neurons pruning by A1 and NNSP algorithms may be done in a similar way, which we do not present here.

Another global algorithm is OBS algorithm [15]. It is a global algorithm which calculates the Hessian matrix of second derivatives of output error with respect to the net's parameters. For N4 net with 784 input and 100 hidden neurons on the first layers we have 784×100 weights and we need to invert the matrix with $(784 \times 100)^2$ elements, which is beyond our computational resources. Moreover by construction as NNSP algorithm, the OBS algorithm works

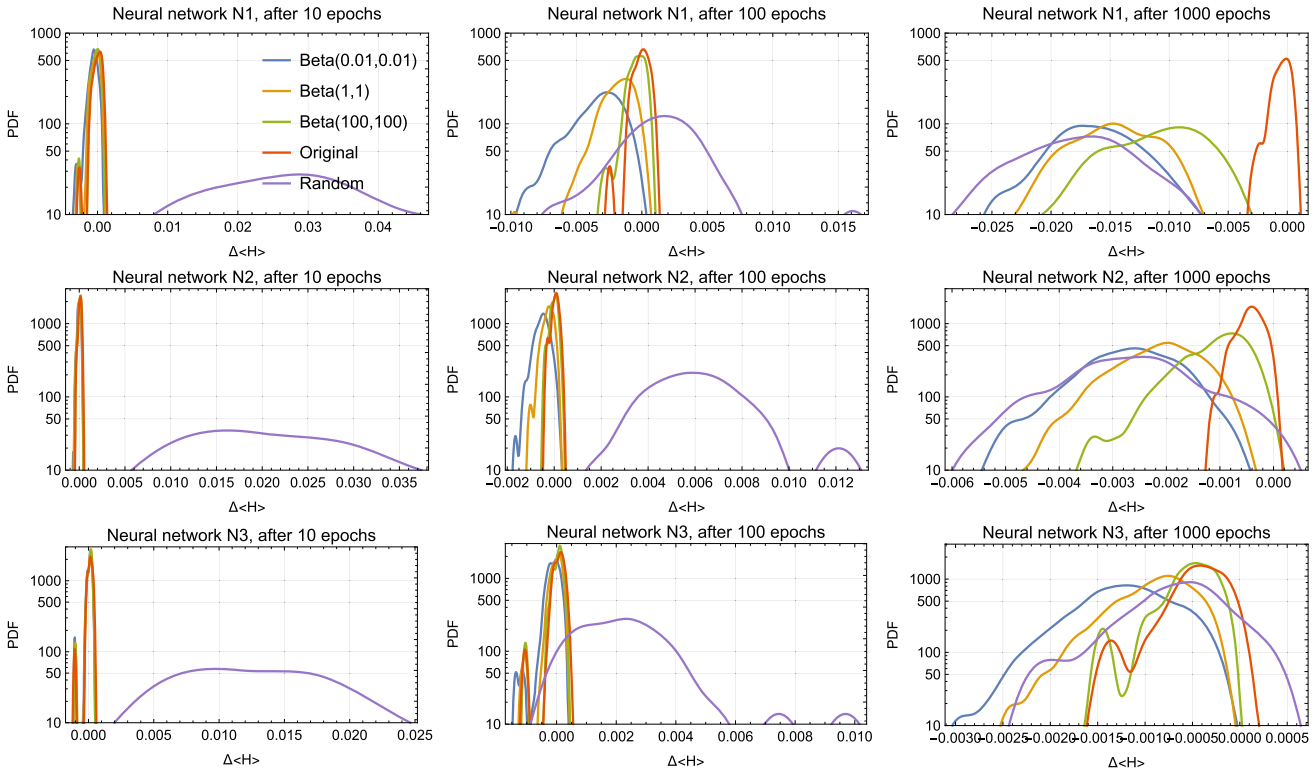

Fig. 21 PDF of $\Delta\langle H\rangle$ for neural networks N1 (first row), N2 (second row) and N3 (third row) after 10 (first column), 100 (second column), and 1000 (third column) additional epochs

	Original B5	Random B4	B1
After 10 epochs retraining	0.91759 ± 0.00033	0.9099 ± 0.0006	0.91776 ± 0.00033
After 100 epochs retraining	0.91759 ± 0.00034	0.91678 ± 0.00033	0.91864 ± 0.00031
After 1000 epochs retraining	0.91777 ± 0.00033	0.92294 ± 0.00034	0.92313 ± 0.00026

Fig. 22 Averaged accuracies of 28 N1 nets trained for 10000 epochs to accuracy 0.91751 ± 0.00034. After training one new neuron was added according to algorithm B1, randomly (B4) or not added (original B5). Then retraining was done for 10, 100, and 1000 epochs

with the net in a local minimum. So one does not apply it to the untrained net.

5.6 Replication

For numerical testing of the replication algorithms, described in the previous section, we trained the feedforward neural networks N1, N2 and N3 for 25,000 epochs. Then we use one of the algorithms to add one more neuron to the second layer and run the simulation for 10, 100, and 1000 additional epochs.

In Fig. 21, we plot PDF (or probability distribution function) of $\Delta\langle H\rangle$, acquired from fifty runs with different initialization, for different neural networks and for different numbers of additional epochs. We use the β distribution to model probability distribution $P(\chi_i)$ for splitting weights between the parent and child neurons in Eqs. (48) and (49).

The five lines on each plot describe five different algorithms:

B1. "Beta(0.01, 0.01)"—neuron is replicated with Beta distribution and both shape parameters equal to 0.01 (blue lines in Fig. 21),

B2. "Beta(1, 1)"—neuron is replicated with Beta distribution and both shape parameters equal to 1 (yellow lines in Fig. 21),

B3. "Beta(100, 100)"—neuron is replicated with Beta distribution and both shape parameters equal to 100 (green lines in Fig. 21),

B4. "Random"—neuron with random weights (drawn from the same distribution as other weights) is added (purple line),

B5. "Original"—no new neurons are added (red lines in Fig. 21).

Original B5

	Peaked at 3	Peaked at 1	Validation set
After 10000 epochs training	0.9404 ± 0.0013	0.8812 ± 0.0018	0.8405 ± 0.0028
After 10 epochs retraining	0.9072 ± 0.0019	0.9029 ± 0.0014	0.8452 ± 0.0028
After 100 epochs retraining	0.8873 ± 0.0018	0.9139 ± 0.0012	0.8518 ± 0.0028
After 1000 epochs retraining	0.8720 ± 0.0021	0.9312 ± 0.0012	0.8569 ± 0.0028

B1

	Peaked at 3	Peaked at 1	Validation set
After 10000 epochs training	0.9404 ± 0.0013	0.8812 ± 0.0018	0.8405 ± 0.0028
After 10 epochs retraining	0.9056 ± 0.0019	0.9033 ± 0.0014	0.8445 ± 0.0030
After 100 epochs retraining	0.8881 ± 0.0018	0.9141 ± 0.0011	0.8526 ± 0.0026
After 1000 epochs retraining	0.8746 ± 0.0018	0.9325 ± 0.0009	0.8595 ± 0.0024

Random B4

	Peaked at 3	Peaked at 1	Validation set
After 10000 epochs training	0.9404 ± 0.0013	0.8812 ± 0.0018	0.8405 ± 0.0028
After 10 epochs retraining	0.8989 ± 0.0026	0.8995 ± 0.0016	0.8377 ± 0.0034
After 100 epochs retraining	0.8836 ± 0.0020	0.9132 ± 0.0012	0.8493 ± 0.0028
After 1000 epochs retraining	0.8740 ± 0.0019	0.9344 ± 0.0010	0.8613 ± 0.0023

Fig. 23 Averaged accuracies of 28 N1 nets trained for 10,000 epochs on the dataset peaked on "3" (6000 - "3" and 600 each other digit). After training one new neuron was added according to algorithm B1, randomly (B4) or not added (original B5). Then retraining was done on the dataset peaked on "1" (6000 - "1" and 600 each other digit). Accuracies are given on these 2 sets and the validation set of 10000 evenly distributed digits

For algorithms B1, B2, and B3 the most efficient neuron (smallest E'_k or E'_k) on the second layer was replicated. One can see in Fig. 21 that after 1000 additional epochs the most efficient algorithm is B1 (i. e. outgoing connections from a parent neuron are randomly split between parent and child neurons) and the least efficient algorithm is B3 (i. e. new outgoing weights from parent and child neuron equal to half of old outgoing weight from parent neuron). The main reason is that algorithm B1 creates a child neuron which is maximally independent from parent neuron while for algorithm B3 the parent and child neurons are equivalent and it would take time for them to diverge due to stochastic learning dynamics. B4 algorithm is inefficient in the short run (i.e., after 10 or 100 epochs), but the algorithm becomes as efficient as B1 in a long run (i.e., after 1000 epochs).

The additional neuron gives the most pronounced effect for network N1 since it has only 5 neurons on the second layer. In terms of accuracy, it is given in Fig. 22, where the one can clearly see the advantage of the B1 algorithm.

5.7 Replication on a new dataset

One may meet a situation when the net was trained on a set unevenly distributed among the output classes with a majority of data of one particular class. Then a new training set becomes available peaked at a different class. In this section we study the ability of our most efficient replication algorithm B1 to train on such a new set. To this end we trained 28 N1 nets on the set with 6000 examples of number "3" and 600 examples of each other digit for 10,000 epochs. The total number of examples in the set is 11,400. Then retrained these nets on the set with 6000 examples of number "1" and 600 examples of each other digit. In Fig. 23, we compare the work of three algorithms B1, B4, and B5. One can see that random addition of a neuron (B4) becomes most efficient in a long run, i.e., after 1000 epochs while algorithm B1 behaves better in intermediate time interval 100 epochs. In a short run of 10 epochs, B1 is comparable to the original algorithm and they both are better than random addition of a neuron.

5.8 Programmed death followed by replication

To demonstrate the computational advantage of a combined algorithm, i.e., programmed death followed by replication, we used algorithm A1 (for neuron removal) and B1 (for neuron addition) with neural architecture N4. The main reason for using N4 (as opposed to N1, N2 or N3) is that one needs a large number of neurons on the second layer for the effect to be most noticeable. We first run the simulation for Δt epochs, calculate efficiencies (28)

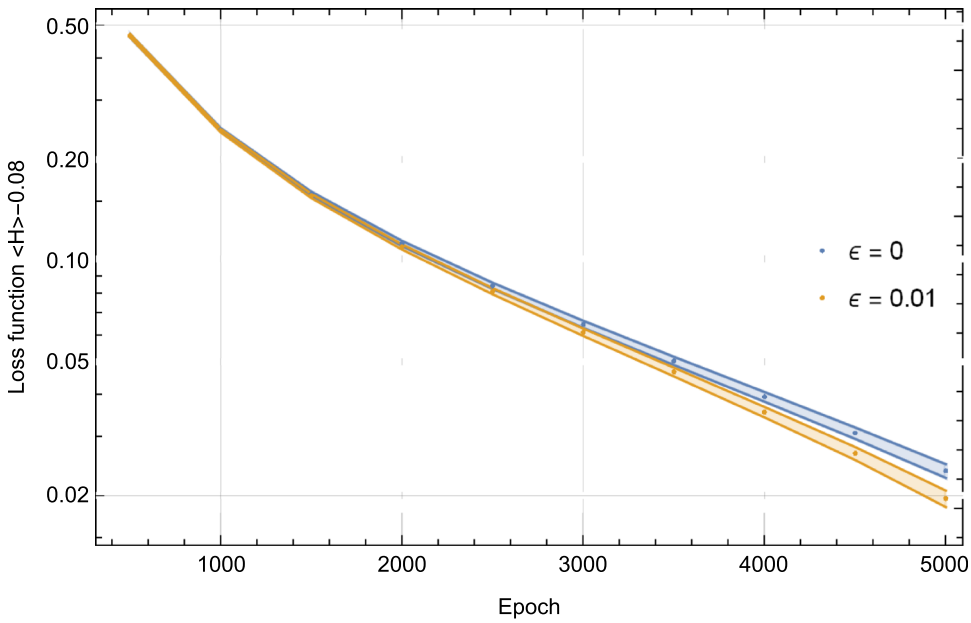

Fig. 24 Log-linear plot of average loss function (minus asymptotic loss function) vs. time for a combined A1–B1 algorithm with cutoff $\epsilon = 0.01$ and $\Delta t = 500$ epochs (yellow line with error bands) and for original algorithm $\epsilon = 0$ (blue line with error bands) averaged over 28 runs with different initialization

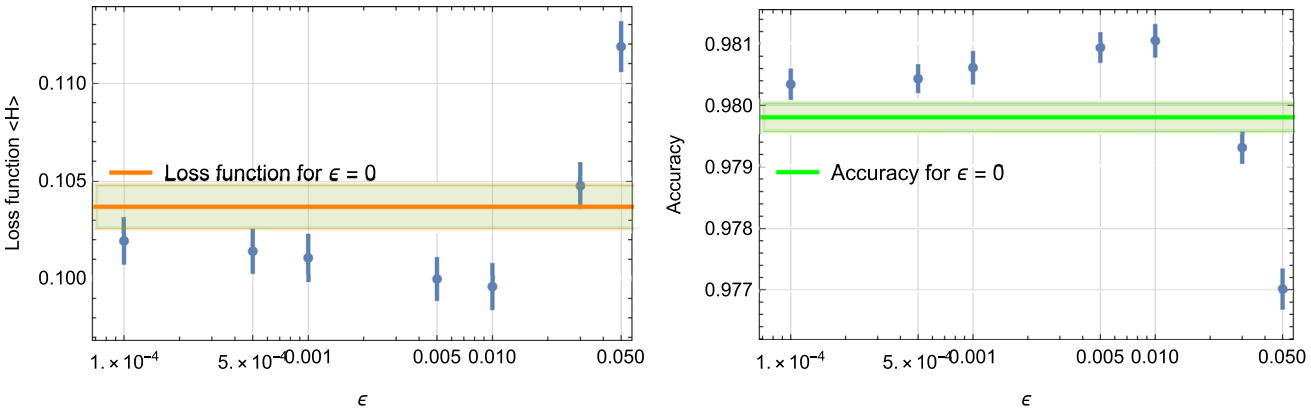

Fig. 25 Average loss (left) and accuracy (right) after 5000 epochs versus cutoff threshold ϵ averaged over 28 runs with different initialization. Bands show the results for $\epsilon = 0$

of all neurons on the second layer and use algorithm A1 to remove all neurons (but not more than $N/2$) whose efficiencies are less than a cutoff ϵ. If n neurons were removed, then we use algorithm B1 to replicates n most efficient neurons and continue the simulation for another Δt epochs and then execute the combined A1-B1 algorithm again, etc.

In Fig. 24, we plot the average loss function for the combined A1–B1 algorithm for 5000 training epochs with $\epsilon = 0.01$ and $\Delta t = 500$ epochs, and the original algorithm (or more precisely with $\epsilon = 0$ and $\Delta t = 500$).

In Fig. 25, we plot the average loss function and accuracy for the combined A1-B1 algorithm after 5000 training epochs with different choice of ϵ and $\Delta t = 500$ epochs compared with the original algorithm (or more precisely with $\epsilon = 0$ and $\Delta t = 500$).

In Fig. 26, we plot the number of neurons affected by the combined A1-B1 algorithm for different ϵ and $\Delta t = 500$ epochs. For the optimal parameters $\epsilon = 0.01$ and $\Delta t = 500$ about 40% of neurons are affected by the algorithm after $\Delta t = 500$ epochs, but this fraction goes to 20% after $2\Delta t = 1000$ epochs, etc.

The computational advantage of the combined A1-B1 algorithm is easy to understand. Indeed, the inactive neurons are constantly removed by the programmed death algorithm A1 and become active by the replication algorithm B1 which improves the learning efficiency for a carefully chosen cutoff threshold.

6 Conclusion

In this article, we took a small step toward developing machine learning algorithms inspired by the well-known biological phenomena. In particular, we analyzed the

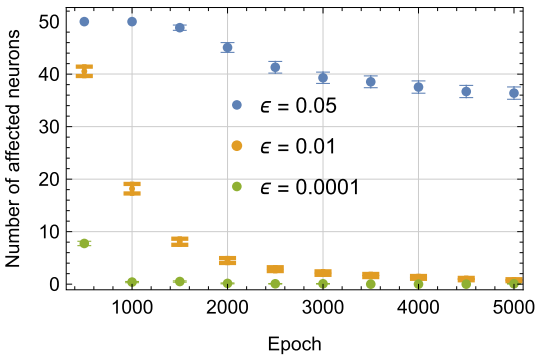

Fig. 26 Number of neurons affected by the combined A1-B1 algorithm with cutoff ϵ and $\Delta t = 500$ epochs vs. time averaged over 28 runs with different initialization

computational advantage of the programmed death and replication, since they represent the most essential phenomena not only in biology [9], but also in physics [11]. Indeed, in both biological and physical systems, the fundamental information processing units can either be added to the system or removed from the system, which gives rise to the biological phenomena of replication and programmed death [9] and to the physical phenomena of emergent quantumness [11].

The developed programmed death and replication algorithms may have a wide range of applications in machine learning. For example, the programmed death algorithm may be useful for compression of neural networks for the use on devices with limited computational resources. In contrast, the replication algorithm may be useful for improving the performance of already trained neural networks on the devices where additional computational resources are available. We have also shown that a combination of programmed death and replication algorithms (i.e., reconnecting the least efficient neuron to reduce the load on the most efficient neuron) may be useful for improving the learning efficiency of an arbitrary machine learning system.

More generally, when the machine learning system is stuck in a local minimum of the average loss function, the continuous transformations (e.g., stochastic gradient decent) cannot be efficient and a discrete transformation must be performed instead. With this respect the programmed death followed by replication is an example of such transformation. Indeed, the rewiring of connections from the least efficient (i.e., programmed death) to assist the most efficient neurons (i.e., replication) is a discrete transformation that may turn out to be useful for certain machine learning tasks.

Although our analytical results are robust, the numerical results are only preliminary and a lot more numerical testing is needed in order to confirm the computational advantages of the proposed algorithms. Moreover, so far we have analyzed the discrete transformations that correspond to only two biological phenomena, programmed death and replication, and there are many other important biological [9] and physical [14] phenomena that can be analyzed in a similar manner which we leave for future work.

A Appendix

Here, we list the pruning algorithms customized for the feedforward neural network with n neurons $\{x_1(t), ...x_n(t)\} = \{x_1, ...x_n\}$ in an hidden layer t such that

$$x_i = f\left(\sum_k w_{ik}^{t-1} x_k(t-1) + b_i^{t-1}\right), \qquad x_j(t+1) = f\left(\sum_k w_{jk}^t x_k + b_j^t\right). \tag{70}$$

A.1 Connection cut algorithm

1. Measure variances of neurons on level t and level $t+1$
$$C_{kk}^t = \langle \Delta x_k(t)^2 \rangle, \quad C_{ii}^{t+1} = \langle \Delta x_i(t+1)^2 \rangle. \tag{71}$$
2. Find the neuron l with minimal efficiency (28)
$$E_l = \min_k E_k = \min_k C_{kk}^t \sum_i \frac{(w_{ik}^t)^2}{C_{ii}^{t+1}} f_i'\Big(\sum_j w_{ij}^t \langle x_j \rangle + b_i^t\Big)^2. \tag{72}$$
3. Use $x_l = \langle x_l \rangle$ as linear dependence equation (39) with
$$a_{k\neq l} = 0, \quad a_l = 1, \quad a_0 = \langle x_l \rangle. \tag{73}$$
4. Remove neuron l from the net according to (41)
$$\sum_k w_{jk}^t x_k + b_j^t \simeq \sum_{k\neq l} w_{jk}^t x_k + \tilde{b_j^t}, \quad \tilde{b_j^t} = w_{jl}^t \langle x_l \rangle + b_j^t. \tag{74}$$
5. Do so while there are neurons with efficiency less than a cutoff or while accuracy or loss stays acceptable.

A.2 Probability algorithm

1. Measure variances of neurons on level t and level $t+1$
$$C_{kk}^t = \langle \Delta x_k(t)^2 \rangle, \quad C_{ii}^{t+1} = \langle \Delta x_i(t+1)^2 \rangle. \tag{75}$$
2. Find the neuron l with minimal efficiency (28)
$$E_l = \min_k E_k = \min_k C_{kk}^t \sum_i \frac{(w_{ik}^t)^2}{C_{ii}^{t+1}} f'\left(\sum_j w_{ij}^t \langle x_j \rangle + b_i^t\right)^2. \tag{76}$$

3. Use linear dependence equation (39) $\sum_j a_j x_j = a_0$ with

$$a_j = \sum_i \frac{w_{il}^t w_{ij}^t}{C_{ii}^{t+1}} f'\left(\sum_k w_{ik}^t \langle x_k\rangle + b_i^t\right)^2, \quad a_0 = \sum_j a_j \langle x_j\rangle. \tag{77}$$

4. Remove neuron l from the net according to (41)

$$\sum_k w_{jk}^t x_k + b_j^t \simeq \sum_{k\neq l} \tilde{w}_{jk}^t x_k + \tilde{b}_j^t, \tag{78}$$

where

$$\tilde{w}_{jk}^t = w_{jk}^t - w_{jl}^t \frac{a_k}{a_l}, \quad \tilde{b}_j^t = w_{jl}^t \frac{a_0}{a_l} + b_j^t. \tag{79}$$

5. Do so while there are neurons with efficiency less than a cutoff or while accuracy or loss stays acceptable.

A.3 Covariance algorithm

1. Measure covariance matrix $C_{kj}^t = \langle \Delta x_k \Delta x_j\rangle$ (10) of the neurons on hidden layer t and find its eigenvectors $\mathbf{v}$ and eigenvalues λ (11).
2. Find the neuron l and the eigenvalue λ_p with minimal efficiency (17)

$$E_l' = \min_k E_k' = \min_{i,k} \frac{\lambda_i}{\left(\mathbf{v}_k^{(i)}\right)^2} = \frac{\lambda_p}{\left(\mathbf{v}_l^{(p)}\right)^2}. \tag{80}$$

3. Use $\sum_k \mathbf{v}_k^{(p)} x_k = \lambda_p$ as linear dependence equation (39) with

$$a_k = \mathbf{v}_k^{(p)}, \quad a_0 = \lambda_p. \tag{81}$$

4. Remove neuron l from the net according to (41)

$$\sum_k w_{jk}^t x_k + b_j^t \simeq \sum_{k\neq l} \tilde{w}_{jk}^t x_k + \tilde{b}_j^t, \tag{82}$$

where

$$\tilde{w}_{jk}^t = w_{jk}^t - w_{jl}^t \frac{a_k}{a_l}, \quad \tilde{b}_j^t = w_{jl}^t \frac{a_0}{a_l} + b_j^t. \tag{83}$$

5. Do so while there are neurons with efficiency less than a cutoff or while accuracy or loss stays acceptable.

Acknowledgements V.V. was supported in part by the Foundational Questions Institute (FQXi) and the Oak Ridge Institute for Science and Education (ORISE).

Data availability The MNIST dataset [24] analyzed during the current study is available in the MNIST database, http://yann.lecun.com/exdb/mnist/.

Declarations

Conflict of interest The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Galushkin AI (2007) Neural networks theory. Springer, Berlin, p 396
2. Schmidhuber J (2015) Deep learning in neural networks: an overview. Neural Netw 61:85–117
3. Haykin Simon S (1999) Neural networks: a comprehensive foundation. Prentice Hall, Hoboken
4. Vapnik Vladimir N (2000) The nature of statistical learning theory. Information Science and Statistics
5. Hopfield JJ (1982) Neural networks and physical systems with emergent collective computational abilities. PNAS 79(8):2554–2558
6. Shwartz-Ziv R, Tishby N (2017) Opening the black box of deep neural networks via information. arXiv:1703.00810 [cs.LG]
7. Roberts D, Yaida S, Hanin B (2022) The principles of deep learning theory: an effective theory approach to understanding neural networks. Cambridge University Press, Cambridge
8. Vanchurin V (2021) Toward a theory of machine learning. Mach Learn: Sci Technol 2:035012
9. Vanchurin V, Wolf YI, Katsnelson MO, Koonin EV (2022) Towards a theory of evolution as multilevel learning. Proc Natl Acad Sci USA 119:e2120037119
10. Vanchurin V, Wolf YI, Koonin EV, Katsnelson MO (2022) Thermodynamics of evolution and the origin of life. Proc Natl Acad Sci USA 119:e2120042119
11. Katsnelson MI, Vanchurin V (2021) Emergent quantumness in neural networks. Found Phys 51(5):1–20
12. Katsnelson MI, Vanchurin V, Westerhout T (2021) Self-organized criticality in neural networks. arXiv:2107.03402
13. Vanchurin V (2022) Towards a theory of quantum gravity from neural networks. Entropy 24:7
14. Vanchurin V (2020) The world as a neural network. Entropy 22:1210
15. Hassibi B, Stork DG (1992) Second order derivatives for network pruning: optimal brain surgeon. Adv Neural Inform Proc Syst 5
16. Medeiros CMS, Baretto GA (2013) A novel weight pruning method for MLP classifiers on the MAXCORE principle. Neural Comput Appl 22:71–84
17. Thomas P, Suhner M-C (2015) A new multilayer perceptron pruning algorithm for classification and regression applications. Neural Process Lett 42(2):437–458
18. Augasta MG, Kathirvalavakumar T (2011) A novel pruning algorithm for optimizing feedforward neural network of classification problems. Neural Process Lett 34:241–258
19. Zeng X, Yeung DS (2006) Hidden neuron pruning of multilayer perceptrons using a quantified sensitivity measure. Neuro Comput 69:825–837
20. Kwok TY, Yeung DY (1997) Constructive algorithms for structure learning in feedforward neural networks for regression problems. IEEE Trans Neural Netw 8(3):630–645
21. Parekh R, Yang J, Honavar V (2000) Constructive neural-network learning algorithms for pattern classification. Trans Neural Netw 11(2):436–451
22. Monirul IMd, Abdus SMd, Faijul Md, Xin Y, Kazuyuki M (2009) A new constructive algorithm for architectural and functional adaptation of artificial neural networks. IEEE Trans Syst, Man, Cybern Part B, Cyberne: Publ IEEE Syst, Man, Cybern Soc 39(6):1590–1605

23. Puma-Villanueva WJ, dos Santos EP, Von Zuben FJ (2012) A constructive algorithm to synthesize arbitrarily connected feed-forward neural networks. Neurocomputing 75(1):14–32
24. LeCun Y, Bottou L, Bengio Y, Haffner P (1998) Gradient-based learning applied to document recognition. Proc IEEE 86(11):2278–2324