

Evolutionary federated learning on EEG-data

Gábor Szegedi, Péter Kiss, and Tomáš Horváth

Department of Data Science and Engineering
ELTE – Eötvös Loránd University, Faculty of Informatics
Budapest, H-1117 Budapest, Pázmány Péter sétány 1/C., Hungary
wayasam@gmail.com, {peter.kiss, tomas.horvath}@inf.elte.hu

Abstract: With the spread of digitalization across every aspects of society and economy, the amount of data generated keeps increasing. In some domains, this generation happens in such a massively distributed fashion that poses challenges for even collecting the data to build machine learning (ML) models on it, not to mention the processing power necessary for training. An important aspect of processing information that has been generated at users is privacy concerns, that is, users might be unwilling to expose anything that would enable one to draw any conclusion regarding to confidential information they possess. In this work, we present a experiment on a genetic algorithm based federated learning (FL) algorithm, that reduces the data transfer from individual users to the learner to a single fitness value.

1 Introduction

The paradigm of federated learning (FL) [1] addresses the more and more timely scenario in which the data to be processed is generated in a massively distributed environment, where traditional approaches for building machine learning (ML) models become extremely challenging, mostly in logistical point of view. That is, when data is generated at client devices as mobile phones, tablets or smart watches, collecting, storing and processing all these information in data centers might be difficult task (*aggregation problem*) and, according to the idea of FL, not necessary by all means.

Another problem regarding the traditional data center based solution is *privacy concerns*. It might happen that users of the applications that build on centralized model training are reluctant to share their possibly confidential data. We believe a particularly fitting scenario for this problem is the use case of medical applications. Each medical institute might have a lot of patient data, but that may be far from enough to train their own prediction models. Here, sharing the data across a big number of institute can yield a great help in developing automated diagnostic tools. But being the private nature of these data, hospitals probably decide not to share anything of this information either to protect their reputation or due to legal regulations.

As it is summarized in [1], the characteristics of data that FL is concerned with can be described as follows:

- **Massively Distributed** Data points are stored across a large number K of nodes. In particular, the number of nodes can be much bigger than the average number of training examples stored on a single node (n/K).
- **Non-IID** Data on each node may be drawn from a different distribution. That is, the data points available locally are far from being a representative sample of the overall distribution.
- **Unbalanced** Different nodes may vary by orders of magnitude in the number of training examples they hold.

Formally, we have K nodes and n data points, a set of indices \mathcal{P}_k ($k \in \{1, \dots, K\}$) of data stored at node k , and $n_k = |\mathcal{P}_k|$ is the number of data points at \mathcal{P}_k . We assume that $\mathcal{P}_k \cap \mathcal{P}_l = \emptyset$ whenever $l \neq k$, thus $\sum_{k=1}^K n_k = n$.

We can then define the local loss for node as $F_k(\mathbf{w}) \stackrel{\text{def}}{=} \frac{1}{n_k} \sum_{i \in \mathcal{P}_k} f_i(\mathbf{w})$, where $f_i(\mathbf{w})$ is the loss of our model at the i th training example, given the parametrisation \mathbf{w} . Thus the problem to be minimized will become:

$$\min_{\mathbf{w} \in \mathbb{R}^d} f(\mathbf{w}) = \sum_{k=1}^K \frac{n_k}{n} F_k(\mathbf{w}). \quad (1)$$

To solve the learning problem (6) for neural networks (NNs), the mainstream way is – starting from a common initial parametrisation – to train local models using some version of gradient descent methods, then aggregate the local model updates (e.g. gradients), or equivalently the local models themselves to update the global model. The global model then will be sent back to the worker nodes. Algorithm 1 is one of the most successful algorithms for federated NN training, called FederatedAveraging [2].

FederatedAveraging works pretty well solving the *aggregation problem*, however using gradients or, equivalently, the local models for the global aggregation step still exposes some information on users data. To address *privacy concerns*, the solution is usually to apply achievements of differential privacy[3][4][5] atop the gradient based learning process.

In this paper we present a slightly different approach, namely, we investigated whether it is possible to train NNs in a federated fashion without using gradient in any context. To approach the problem, it seemed to be a simple choice to try evolutionary algorithms. Since a rich literature is already available on evolutionary optimization of

Algorithm 1 FederatedAveraging

```
1: procedure SERVER
2:   initialize  $w_0$ 
3:   for  $t = 0; 1; 2; \dots$  do
4:      $m \leftarrow \max(C \cdot K, 1)$ 
5:      $S_t \leftarrow m$  client nodes randomly
6:     for all  $k \in S_t$  in parallel do
7:        $w_{t+1}^k \leftarrow \text{ClientUpdate}(k, w_t)$ 
8:     end for
9:      $w_{t+1} = \sum_{k=1}^K \frac{n_k}{n} w_{t+1}^k$ 
10:  end for
11: end procedure
12: procedure CLIENTUPDATE( $k, w$ )
13:   $\mathcal{B} \leftarrow$  split  $\mathcal{P}_k$  to set of batches
14:  for all  $b \in \mathcal{B}$  do
15:     $w \leftarrow w - \eta \nabla f(w, b)$   $\triangleright$  gradient of loss on batch  $b$ 
16:  end for
17:  return  $w$ 
18: end procedure
```

NNs, we only transfer this knowledge into the federated environment.

For the concrete task to be solved by our method we have chosen classification of EEG-signals using convolutional neural networks (CNNs).

The main contributions of this paper are

1. a proof of concept for applicability of genetic algorithms to federated training of NNs without using vulnerable gradients;
2. presenting Federated Neuroevolution (FNE) a simple algorithm for the federated training, applying a distributed fitness function.

2 Neuroevolution

Evolutionary algorithms (EAs) follow the pattern of evolution as it is observed by biologists in the nature. According to this, in an infinite cycle of life, the most apt individuals can produce offsprings possessing a potentially slightly changed (mutated) mixture of their genomes that might result an enhanced ability to face challenges in their life. The main assumption in biology is that those individuals survive and create descendants with a bigger chance, who, in some aspects are superior to the others. The main structure of an EA is sketched in Algorithm 2

Algorithm 2 EA

```
1: generate an initial population  $G_0, i = 0$ 
2: repeat
3:    $\forall \text{individual}_j \in G_i : f_j = \text{fitness}(\text{individual}_j)$ 
4:   select parents from  $G_i$  based on their fitness
5:   produce offspring generation  $G_{i+1}$ 
6:    $\forall \text{individual}_j \in G_{i+1} : \text{individual}_j \leftarrow \text{mutate}(\text{individual}_j)$ 
7: until termination criterion is satisfied
```

EAs – as nature inspired methods in general – are often used to discover very complex, high dimensional and/or non-convex search problems, therefore, attempts to apply these methods on optimizing NNs has a long history.

Recently, nature-inspired methods in relation with NNs are used mostly for hyper-parameter tuning that includes searching for an efficient architecture.

A big part of this rich literature is concerned specifically CNN-s, what we apply for our problem. Methods of Genetic CNN [6], hierarchical evolution [7], large-scale evolution [8], asynchronous CNN evolution [9] and automatic CNN design [10] give graph based methods to design automatically the stack convolutional layers (skipping potentially fully connected parts of the network) for image classification through genetical evolution of subsequent layers with various innovative encoding techniques.

In these scenarios, the learning itself is still based on calculating the gradient and updating the model according to that (backpropagation).

Using backpropagation though being based on calculation of gradients and on applying them on the weights of the network is exactly what we want to avoid in our experiment. Before the monocacy of derivative based training algorithms however biology-inspired training methods was a rather popular research topic, thus there is a rich, though a bit dated literature concerned with our constrained problem. [11] and [12] give a summary of the these initial approaches to neuroevolution (NE).

There is a very interesting branch of applications of NE for general NN-s, that includes techniques to purely genetically train the architecture along with the weights of the networks. Among the most important algorithms that belong here it might be worth to mention NeuroEvolution of Augmenting Topologies (NEAT) [13], and Hypercube-based NEAT (HyperNEAT) [14] and its specialization for modular evolution of NN-s, HyperNEAT-LEO [15] and Generative NE [16]. Despite of the power of HyperNEAT, we decided first to focus on training a predefined architecture, thus our method is based on more "traditional" NE algorithms.

For applying evolutionary approach on an issue, one need to specify an encoding of the problem, a selection, a crossover and a mutation method as well as a fitness function.

In the rest of this section we shortly describe the stages of an EA along with a couple of examples of how these stages have been implemented in some work on the field of NE, that gave inspiration to our algorithm. At the end of the section we also describe approaches aiming at handle overfitting that has been proven a serious problem in NE.

2.1 Encoding

Genetic algorithms work on sequence of features that would be mixed, or altered according some granularity defined over them. Thus the first step in solving a problem genetically is to provide a description of the search space.

We refer to this description as encoding, that can be direct or indirect.

Direct encoding is the more traditional way of problem encoding, where sections of the genome more or less correspond to specific parameters. Some of the early methods handle some switches as well, that control the connectivity of the specific perceptrons.

[17] proposes a system based on a parallel genetic algorithm, ANNA ELEONORA, for learning both topology and for connection weights. Topology Utilizes binary representation of networks, with granularity encoding that is handled through one bit flag to determine connectivity, that is, whether the given edge is present in the recent setup or not, followed by the substring of weights. These substrings are ordered in a way that connections into the same neuron are grouped together.

[18] presents a variant of EA applied immediately for float weights. The input of the EA is a vector x of variables, that are the parameters of the model (that is the weights of the connections), the biases, and the newly invented link switches. Link switches are variables, that control the connectivity of the network, that is, negative value represents that the edge is switched off. The search space is constrained by upper and lower bounds on variables (weights): $x \in I_1 \times I_2 \times \dots \times I_d$, where $I_i = [l_i, u_i]$, $l_i, u_i \in \mathbb{R}$ for $i = 1, 2, \dots, d$.

Theoretically using the connectivity features of the encoding the first method is able to evolve the architecture too. The issue with this approach to encoding is that the problem space grows very fast as we scale up the network (which we need if we want to solve complex problems).

Indirect Encoding The scaling problem of Direct Encoding can be solved with Indirect Encoding, that instead of separated representation of model parameters uses generative information. In HyperNEAT [14], which is maybe the most important representative of this class, genes of the genome are defining functions based on which weights can be generated.

2.2 Fitness

The fitness function serves to specify how well a given individual performs on the problem to be solved. A higher value of the fitness function means a better solution for the problem, while lower fitness value reports a poor performance. Fitness is often normalized thus a function that produces a fitness value 1 for a perfect solution, and 0 for completely wrong setup can work well. As an example for a normalized fitness in ML scenarios, [19] proposes a fitness function for NN defined as $f_{norm} = \frac{1}{1+err}$, where $err = \sum_{k=1}^m \frac{\sum_{i=1}^d |y_i - \hat{y}_i|}{md}$, with d denoting the output dimension, and m the number of examples, applying mean absolute error.

2.3 Crossover

Crossover is a method that defines, how we combine individuals of a generation to create offsprings for the next generations. One simple way is – as in [17] – to combine the parts of parent individuals at some cutting points. Another approach is presented in [18], where crossover is actually taking the average of the corresponding weights of the two individuals: $x^{(t+1)} = \frac{x_1(t) + x_2(t)}{2}$ where $x(t)$ s are the individuals represented as vectors of parameters.

2.4 Mutation

Mutation methods serve for adding extra variance to the individual genomes to enable them to discover a bigger part of the search space. [17] provides a representation that translates different topologies and encoding length into a common string format granting compositionally different descriptions. At mutations, it applies three separate probabilities for swapping bits such that granularity bits, connectivity bit, and weight bits. For effectively explore the search space, it uses EA simplex [20], instead of taking three populations and creating a fourth based on those.

In [18], where the possible values for genes are constrained, mutation is carried out according to the following formula: $x^{(t+1)} = x^{(t)} + B\delta$, $B \in \mathbb{R}^2$ is a diagonal matrix, with a diagonal $B_{ii} \in \{0, 1\}$, and $l_i \leq x_i^{(t+1)} + \delta_i \leq u_i$. Based on this rule, the algorithm generates three individuals/chromosomes: at the first only one element of the diagonal of B can be one, at the second one a random number of diagonal of elements, and at the last, $B_{ii} = 1$ for all $i = 1, \dots, d$. The one with the best fitness of these three will replace the weakest one in the next population.

2.5 Overfitting

Using EA usually involves a high computation demand, which can be reduced through decreasing the number of evaluations of the model, that is the size of training data on which we want to try out the models defined by a given generation of the genetic algorithms.

Earlier applications of EA usually did not use separation of data into training and test set (like [21], for example). Practitioners soon realized, however, that models trained this way perform poorly on not seen data points, revealing the tendency of evolutionary methods to strongly overfit on the training problems. This issue got in the center of interest, when as an attempt to reduce run time they tried to use subsets of the training data to evaluate individuals.

[22] made comprehensive experiments proving that evolution is potentially able to extrapolate from the randomly chosen test sets. A very promising direction to reduce overfitting is random sampling, where at each generation, a random subset of the training data is chosen and evolution is performed based on the fitness on that sample. The Random Sampling Technique (RST) [23] was originally used for speeding up the GE runs in [24], however, it

was already used for preventing overfitting. [25] and [26] drive some experiments on RST, where they were testing two parameters, the Random Subset Size (RSS) and the Random Subset Reset (frequency of changing the subset). They have found, interestingly, that the techniques performs best when both these values are set to one, that is in each iteration the fitness should be tested using a new randomly chosen data point.

In [27], the authors present versions of “interleaved sampling”, that means, instead of random subsets, fitness at each round is evaluated alternating between one and all training samples, with various switching frequencies. As a result, they find that, on their test datasets, the best technique would be to switch in each round between single sample and all sample evaluation.

3 The problem

3.1 Data

For the experiment, we used the EEG Database Data Set [28]. The dataset contains 120 EEG trial data about 122 patients who either belong to the alcoholic or to the control group. In each trial, the patients were shown one or two images of the Snodgrass and Vanderwart picture set [29]. After showing them the stimuli, their brain activation was measured for 1 second on 64 points at 256 Hertz. The measurements are then labelled according to which group they belong to, thus the task of the model to be built is to predict which class of the two does a sample belong to.

3.2 Network architecture

For the network architecture to train, we decided to use the shallow convolutional network from [30], that has been designed specifically for EEG based multiclass prediction problems. The essence of the network is three convolutional layer that are intended to recognize specific patterns in the signals. After two convolutional layers there is a pooling layer and then comes the third convolutional layer. On the output of this layer we applied batch normalization, then added the output dense layer with sigmoid activation.

For the control experiment, we used the AdaDelta optimizer [31] with Categorical Cross-Entropy loss function. At training we used a batch size of 64, 1.0 as learning rate, $\rho = 0.95$, and $\epsilon = 10^{-7}$.

The control model after 100 epochs achieved a validation accuracy of 95% (see Figure 1).

4 The proposed methods

The algorithm runs according to the process defined in Algorithm 2. For starting off we create an initial generation in which for each individuals initialize the weights of the models randomly. From the initial generation then we iterate along the fitness-selection-crossover-mutation loop. In this section we describe the particular methods we used for the different stages.

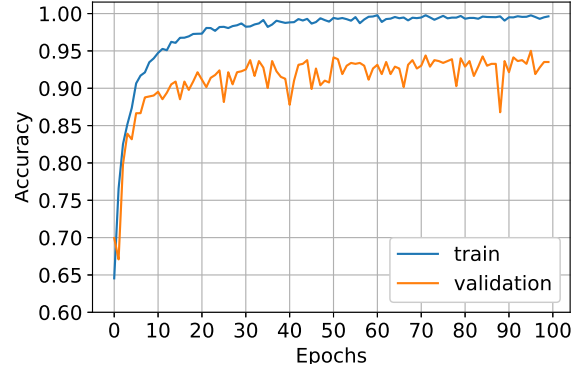


Figure 1: Baseline accuracy (backpropagation)

Selection The candidate set of individuals for crossover is created by sorting the current generation’s models based on their fitness and selecting the $n - 1$ fittest models for crossover. The last parent selected for mating is not among the fittest ones, but chosen randomly from the rest, to add more variance.

Crossover functions Crossover method defines the way according to which new individuals will be generated from the parent generation.

In our method, we pick two parents randomly from the pool of parents to produce the required offspring amount.

We run experiments with four crossover methods. The first three require flattening the vector of weights. These first three approaches are rather popular in EA research.

- **Halving mix:** In this approach, that is a simplified version of the one in [17], the vector of values from the parents are taken to create the offspring vector by taking the first half of it from the first parent and the second half from the second parent. This was the original approach in [32] too. Formally:

$$\{offspring_i\}_{i=1}^n = \begin{cases} a_i, & \text{if } i \leq n/2 \\ b_i, & \text{if } i > n/2 \end{cases} \quad (2)$$

where n is the length of the model vectors and $\mathbf{a} = (a_1, a_2, \dots, a_n)$, $\mathbf{b} = (b_1, b_2, \dots, b_n)$ are the parent vectors.

- **Interleave mix:** Here, the vector of values from the parents are taken to create the offspring vector by interleaving the two parent vectors. Formally:

$$\{offspring_i\}_{i=1}^n = \begin{cases} a_i, & \text{if } i \bmod 2 = 0 \\ b_i, & \text{if } i \bmod 2 = 1 \end{cases} \quad (3)$$

where n is the length of the model vectors and \mathbf{a}, \mathbf{b} are the parent vectors.

- **Mean mix:** In this method, similarly to [18], the vector of values from the parents are taken to create the offspring vector by taking the mean of the two parent vectors at each index. Formally:

$$\{\text{offspring}_i\}_{i=1}^n = \left\{ \frac{a_i + b_i}{2} \right\} \quad (4)$$

where n is the length of the model vectors and \mathbf{a}, \mathbf{b} are the parent vectors.

- **Kernelwise mix:** In this algorithms we used a more coarse units for the crossover. In each convolutional layer there are multiple kernels/filters that hold key pattern information. Similarly, in case of fully connected layers, input weights belonging to a single neurons describes some pattern in the previous layers. These information portions are kept intact during crossover. The offspring model is created by randomly mixing the kernels inside each layer.

In our experiments, the first three crossover methods did not converge. This could be because these approaches are very low level and do not care about the network structure or the patterns learned in the kernels.

Kernelwise mixing is a higher level approach, what we tried after taking a look at how genetics works in nature. In nature, the heredity is also a higher level mixing of genes, instead of low level mix of organic molecules. Thus, traits of the parents are kept intact. The resemblance to genetics can be summarized as follows: the DNA is the network's weights, a gene is a filter and an organic molecule is a float value. With this latest method, mixing the evolutionary training was converging so we were applying this in our approach.

4.1 Mutation functions

Crossover on its own results in generations that are only combinations of the initial generation according to the defined rules. Thus using merely crossover restrict the space searched by the algorithm. To break this random alternations of the offspring are applied in form of mutation functions.

For defining a mutation function we must define the number of mutated values and the scale of the mutation on these values. For the former we used probabilistic value determining the chance of mutation for each value in the model. The latter is a float value determining how much is the impact on each mutating value.

There are the following two main approaches we tried for mutating values in a network:

- **Mutate by offset (from [32]):** Here, we add a random value to the selected values. In our implementation, the offset was a random value between $[-\text{mutation_rate}, \text{mutation_rate}]$.

- **Mutate by multiplication (from [33]):** Here, we multiply the selected values with a random value. In our implementation, the multiplication factor was a random value between $\left[\frac{100 - \text{mutation_rate}}{100}, \frac{100 + \text{mutation_rate}}{100} \right]$.

After experimenting, we found a lot better convergence rate with the second approach.

4.2 Federated fitness function

For fitness, which should be maximized during the evolutionary training, we have chosen the Negative Mean Squared Error (NMSE), that is defined as in Equation (5).

$$f_{NMSE}(w) = -1 * \frac{1}{n} \sum_{i=1}^n \sum_{j=1}^d (\hat{y}_j^{(i)} - y_j^{(i)})^2 \quad (5)$$

where $\hat{\mathbf{y}}$ is the predicted output vector using parameters \mathbf{w} , \mathbf{y} is the target output vector, d is the output dimension and n is the number of examples. This is slightly different function, than the one in the example in section 2.2, but it's behaviour is the same ($\frac{\partial}{\partial w_i} f_{NMSE}(\mathbf{w}) * \frac{\partial}{\partial w_i} f_{norm}(\mathbf{w}) > 0, \forall \mathbf{w}, i$)

Applying the NMSE fitness for the original optimization problem in Equation (6) our task will be to maximize NMSE with respect to \mathbf{w} :

$$\max_{\mathbf{w} \in \mathbb{R}^d} f_{NMSE}(\mathbf{w}) = - \sum_{i=1}^n \|\hat{\mathbf{y}}^{(i)} - \mathbf{y}^{(i)}\|_2^2. \quad (6)$$

4.3 Federated optimization and avoiding overfitting

In our setup the generation of individuals, that is the selection, the crossover and mutation happens at a centralized location at a parameter server. The connected nodes of the system participate in the optimization through evaluating the different proposed setup. The fitness of an individual can be calculated as a weighted average over the local fitness values, in theory, during the training tough, as we will see we should not use this measurement to prevent overfitting.

Avoiding overfitting has been studied in [25, 22, 26, 27], as it is discussed in Section 2.5. The main idea is that we must not include the entire training set in the whole duration of the training. Instead, what most articles propose, is to use subsets of the training data in each generation. The training subset can be changed every generation or kept intact for a few generations. Studies interestingly show that randomly selecting a single training sample is also very effective both for convergence and avoiding over-fitting. Another suggested tweak is to include the full dataset every once in a while.

Due the distributed nature of the problem, it was a rather natural idea to incorporate the native data partitioning of the federated setup, and to do the subset selection at a

higher level and treat the nodes as units of the subset creation, instead of specific data points. Thus, in each generation, the Federated Neuroevolution algorithm selects a subset of the nodes for evaluating the fitness of the current generation. To ensemble the evaluation sets, we have tried the following three approaches:

1. **Random single element for each generation:** Here, in each iteration we ask a randomly selected node to evaluate the population’s fitness on a randomly selected single training sample of it’s own.
2. **Random subset for each generation:** In this approach a random subset of nodes are selected to evaluate. We found this method the most efficient.
3. **Moving window subset for each generation:** Here, we first order the nodes and then select a slice of the list of nodes. This is the window and in every n generation we move the window to the right by 1.

The second approach of randomly selecting a subset of nodes had the best performance. Even if, according to the literature, method 1 works pretty well, in our experiments, the training did not converge at all. The third method seemed to be more promising, training convergence was slower with this method than in the case of the second method and the convergence also capped around 75% validation accuracy.

The main algorithm Using the fitness evaluation methods we described in Section 4.3, the main run of the optimization looks like the following:

- **Validation:** On the server, we retain a validation set and in each generation we calculate and store the validation accuracy of the fittest model of the current generation. This is not far fetched as we can assume that in a Federated setting the server driving the learning would already have a dataset of it’s own.
- **Avoiding critical points:** Based on the history of validation accuracies, we check the last n entries for a match with the current validation accuracy. If there is a match, we conclude that the evolution has reached some kind of critical point of the fitness function as local maximum or saddle point. That is, however we try to combine and mutate the individuals of the subsequent generations, the fitness/accuracy does not increase. Our hypothesis is, that in this case the population stuck in a higher region of the values of fitness function, and in the neighbourhood defined by our mutation rate the offsprings cannot find any increasing directions. In this case we start gradually increasing the mutation rate and the mutation chance multiplier which is initially set to 1. Once the algorithm is out of the local maximum, we reset the values of the mutation rate and mutation chance to the original values. There is an upper bound on the mutation multiplier.

- **Early stopping:** We save the fittest model of each generation, as an additional means to stop before we overfit.

5 Results

We have run the described evolution algorithm for 5000 generations (Figure 2). For our setup, we observed that the convergence was slow but steady, overall.

	Minimum value	Maximum value
Validation Accuracy	48.50%	85.28%
Fitness NMSE	-0.3297	-0.0903

Table 1: Federated Neuroevolution Performance on the EEG Dataset

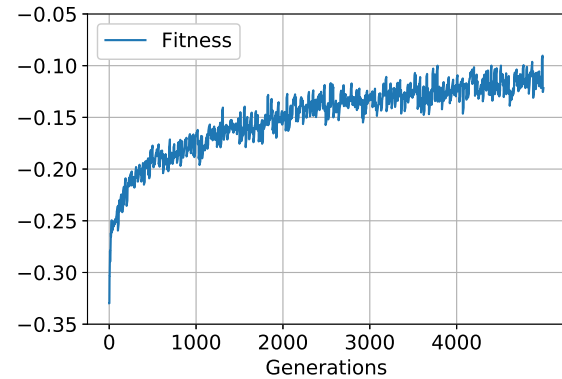
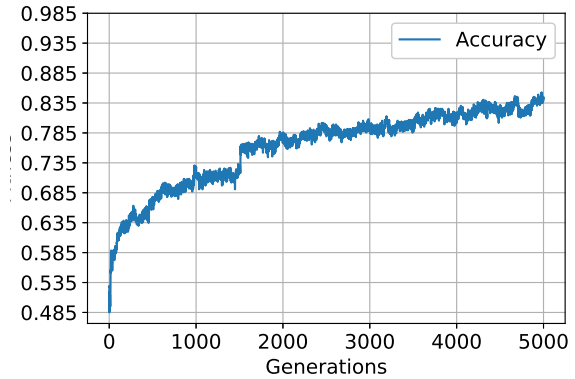


Figure 2: Running Federated Neuroevolution on the EEG dataset for 5000 generations. Fitness is NMSE from equation 5, Accuracy is validation accuracy.

From a fully random state, the algorithm was able to get to 85% validation accuracy as seen in Table 1. This is, of course, a lot less than the baseline but still a good result considering using Neuroevolution for training weights which is not the best method for training NNs.

6 Conclusion and future work

In this paper we described our experiments with a simple method, what we call Federated Neuroevolution (FNE), that is an application of EA adapted for FL of NNs.

We found that our method is applicable on the studied scenario yielding some advantages over the traditional FL methods.

An advantage of EA, compared to the gradient based algorithms originated from [1], [34] or [2], is that it requires even less client data transfer to the server. While FedAVG exposes the client side data distribution and the gradients during learning, FNE only expose the amount of data points of the clients and an abstract fitness number of the model.

The clear disadvantage is that the convergence is a lot slower. We needed 5000 iterations of the algorithm to get to an 85% accuracy which is still less than the baseline's 95%. At this point though our purpose was merely to demonstrate the feasibility of derivative-free learning of NN-s in a FL scenario.

In summary, the technique we introduced, trades off learning speed for privacy gains. We may need a lot of communication rounds which can be bad in a real-world setting of mobile users, but for some use cases, like for data from medical institutions, the rounds of communication is not of primary importance, while keeping data privacy is essential. Another aspect of techniques similar to FNE that might be interesting, is that there is no traditional, backpropagation based learning, that is at client side we can save this rather expensive stages of the learning process.

In the future we think there are several possible directions to develop FNE to make it practical. First the rather poor performance of the system might be improved through experimenting with different submethods (selection, crossover, etc.)

Following the trends in genetic algorithms, the search space could be extended to the network architecture too. This way we could reduce the bias and variance introduced by the model architecture that is chosen rather blindly at the initiation phase of the learning.

Bearing in mind the main purpose of the experiments, that is prevent the communicating the gradients, a range of derivative free methods are available as Differential Evolution [35], Particle Swarm Optimization[36] or other biology inspired methods like Artificial Bee Colony [37]. Similarly, advanced optimization methods as CMA-ES[38] might be applied.

It could be also interesting to experiment with more efficient utilization of resources, since in the current setup in each round the vast majority of nodes is idle.

Acknowledgements EFOP-3.6.3-VEKOP-16-2017-00001: Talent Management in Autonomous Vehicle Control Technologies - The Project is supported by the

Hungarian Government and co-financed by the European Social Fund.

Supported by Telekom Innovation Laboratories (T-Labs), the Research and Development unit of Deutsche Telekom.

References

- [1] J. Konečný, H. B. McMahan, D. Ramage, and P. Richtárik, "Federated optimization: Distributed machine learning for on-device intelligence," *arXiv preprint arXiv:1610.02527*, 2016.
- [2] H. B. McMahan, E. Moore, D. Ramage, S. Hampson *et al.*, "Communication-efficient learning of deep networks from decentralized data," *arXiv preprint arXiv:1602.05629*, 2016.
- [3] K. Chaudhuri, C. Monteleoni, and A. D. Sarwate, "Differentially private empirical risk minimization," *Journal of Machine Learning Research*, vol. 12, no. Mar, pp. 1069–1109, 2011.
- [4] C. Dwork, A. Roth *et al.*, "The algorithmic foundations of differential privacy," *Foundations and Trends® in Theoretical Computer Science*, vol. 9, no. 3–4, pp. 211–407, 2014.
- [5] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang, "Deep learning with differential privacy," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 308–318.
- [6] L. Xie and A. Yuille, "Genetic cnn," in *Proceedings of the IEEE International Conference on Computer Vision*, 2017, pp. 1379–1388.
- [7] H. Liu, K. Simonyan, O. Vinyals, C. Fernando, and K. Kavukcuoglu, "Hierarchical representations for efficient architecture search," *arXiv preprint arXiv:1711.00436*, 2017.
- [8] T. Desell, "Large scale evolution of convolutional neural networks using volunteer computing," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. ACM, 2017, pp. 127–128.
- [9] P. Vidnerová and R. Neruda, "Asynchronous evolution of convolutional networks," 2018.
- [10] Y. Sun, B. Xue, M. Zhang, and G. G. Yen, "Automatically designing cnn architectures using genetic algorithm for image classification," *arXiv preprint arXiv:1808.03818*, 2018.
- [11] D. Whitley, T. Starkweather, and C. Bogart, "Genetic algorithms and neural networks: Optimizing connections and connectivity," *Parallel computing*, vol. 14, no. 3, pp. 347–361, 1990.
- [12] X. Yao, "Evolving artificial neural networks," *Proceedings of the IEEE*, vol. 87, no. 9, pp. 1423–1447, 1999.
- [13] K. O. Stanley and R. Miikkulainen, "Efficient evolution of neural network topologies," in *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No. 02TH8600)*, vol. 2. IEEE, 2002, pp. 1757–1762.
- [14] J. Gauci and K. Stanley, "Generating large-scale neural networks through discovering geometric regularities," in *Proceedings of the 9th annual conference on Genetic and evolutionary computation*. ACM, 2007, pp. 997–1004.

- [15] P. Verbancsics and K. O. Stanley, "Constraining connectivity to encourage modularity in hyperneat," in *Proceedings of the 13th annual conference on Genetic and evolutionary computation*. ACM, 2011, pp. 1483–1490.
- [16] P. Verbancsics and J. Harguess, "Generative neuroevolution for deep learning," *arXiv preprint arXiv:1312.5355*, 2013.
- [17] V. Maniezzo, "Genetic evolution of the topology and weight distribution of neural networks," *IEEE Transactions on neural networks*, vol. 5, no. 1, pp. 39–53, 1994.
- [18] H. Lam, S. Ling, F. H. Leung, and P. K.-S. Tam, "Tuning of the structure and parameters of neural network using an improved genetic algorithm," in *IECON'01. 27th Annual Conference of the IEEE Industrial Electronics Society (Cat. No. 37243)*, vol. 1. IEEE, 2001, pp. 25–30.
- [19] J.-T. Tsai, J.-H. Chou, and T.-K. Liu, "Tuning the structure and parameters of a neural network by using hybrid taguchi-genetic algorithm," *IEEE Transactions on Neural Networks*, vol. 17, no. 1, pp. 69–80, 2006.
- [20] H. Bersini and G. Seront, "In search of a good crossover between evolution and optimization," *Manner and Manderrick*, vol. 1503, pp. 479–488, 1992.
- [21] J. R. Koza and J. R. Koza, *Genetic programming: on the programming of computers by means of natural selection*. MIT press, 1992, vol. 1.
- [22] W. Langdon, "Minimising testing in genetic programming," *RN*, vol. 11, no. 10, p. 1, 2011.
- [23] C. Gathercole and P. Ross, "Dynamic training subset selection for supervised learning in genetic programming," in *International Conference on Parallel Problem Solving from Nature*. Springer, 1994, pp. 312–321.
- [24] Y. Liu and T. Khoshgoftaar, "Reducing overfitting in genetic programming models for software quality classification," in *Eighth IEEE International Symposium on High Assurance Systems Engineering, 2004. Proceedings*. IEEE, 2004, pp. 56–65.
- [25] I. Gonçalves, S. Silva, J. B. Melo, and J. M. Carreiras, "Random sampling technique for overfitting control in genetic programming," in *European Conference on Genetic Programming*. Springer, 2012, pp. 218–229.
- [26] I. Gonçalves and S. Silva, "Experiments on controlling overfitting in genetic programming," in *15th Portuguese conference on artificial intelligence (EPIA 2011)*, 2011, pp. 10–13.
- [27] —, "Balancing learning and overfitting in genetic programming with interleaved sampling of training data," in *European Conference on Genetic Programming*. Springer, 2013, pp. 73–84.
- [28] H. Begleiter. EEG Database Data Set. [Online]. Available: <https://archive.ics.uci.edu/ml/datasets/eeg+database>
- [29] J. G. Snodgrass and M. Vanderwart, "A standardized set of 260 pictures: norms for name agreement, image agreement, familiarity, and visual complexity," *Journal of experimental psychology: Human learning and memory*, vol. 6, no. 2, p. 174, 1980.
- [30] R. T. Schirrmester, J. T. Springenberg, L. D. J. Fiederer, M. Glasstetter, K. Eggenberger, M. Tangemann, F. Hutter, W. Burgard, and T. Ball, "Deep learning with convolutional neural networks for eeg decoding and visualization," *Human brain mapping*, vol. 38, no. 11, pp. 5391–5420, 2017.
- [31] M. D. Zeiler, "Adadelata: an adaptive learning rate method," *arXiv preprint arXiv:1212.5701*, 2012.
- [32] A. Gad. (2019) Artificial Neural Networks Optimization using Genetic Algorithm with Python. [Online]. Available: <https://towardsdatascience.com/artificial-neural-networks-optimization-using-genetic-algorithm-with-python-1fe8ed17733e>
- [33] R. Oullette, M. Browne, and K. Hirasawa, "Genetic algorithm optimization of a convolutional neural network for autonomous crack detection," in *Proceedings of the 2004 congress on evolutionary computation (IEEE Cat. No. 04TH8753)*, vol. 1. IEEE, 2004, pp. 516–521.
- [34] J. Chen, X. Pan, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting distributed synchronous sgd," *arXiv preprint arXiv:1604.00981*, 2016.
- [35] J. Ilonen, J.-K. Kamarainen, and J. Lampinen, "Differential evolution training algorithm for feed-forward neural networks," *Neural Processing Letters*, vol. 17, no. 1, pp. 93–105, 2003.
- [36] B. A. Garro, H. Sossa, and R. A. Vazquez, "Design of artificial neural networks using a modified particle swarm optimization algorithm," in *2009 International Joint Conference on Neural Networks*. IEEE, 2009, pp. 938–945.
- [37] B. A. Garro, H. Sossa, and R. A. Vázquez, "Artificial neural network synthesis by means of artificial bee colony (abc) algorithm," in *2011 IEEE Congress of Evolutionary Computation (CEC)*. IEEE, 2011, pp. 331–338.
- [38] N. Hansen, S. D. Müller, and P. Koumoutsakos, "Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (cma-es)," *Evolutionary computation*, vol. 11, no. 1, pp. 1–18, 2003.