Eötvös Loránd University
Faculty of Informatics

Algorithms and Models for Federated Machine Learning

dr. Tomáš Horváth
Head of Data Science and Engineering Department at ELTE

Gábor Szegedi
Computer Science MSc

Budapest, 2019.

Acknowledgement

First of all I would like to thank my soon-to-be wife Leona for supporting me throughout my studies and these past 10 years of my life. In the days when I faced challenges seemingly impossible to overcome you were there to support me. Thank you for everything.

I would also like to thank dr. Tomáš Horváth for accepting me into his team and introducing me to interesting projects, one of which is this thesis itself. I hope I can participate in many more projects of the department in the future.

Finally a big thanks to Dr. Krisztián Buza for helping overcome technical issues and Péter Kiss for all the great ideas on how to overcome challenges throughout the coding of my thesis.

Contents

Chapter 1

Introduction

As a big Sci-fi fan and programmer I was always interested in Robotics and Artificial Intelligence (AI). During my bachelor's studies this interest grew larger with each passing semester and each Sci-fi book I read. At the end of my bachelor's I decided I would not stop learning and pursue my master's studies at ELTE in the hopes of broadening my knowledge about Artificial Intelligence (AI). I wanted to be part of The AI Revolution [33] to benefit from it and to control it as well to all of our benefits.

I personally think that the motivation for investing in the field roots from human laziness. Isaac Asimov has laid this out perfectly in the Foundation Series [3] where humans create the Positronic Brain that drives the robots of his universe. These robots do all the hard work and eventually even the thinking instead of humans. We see more and more materialization of this theory if we carefully look around us. Autonomous vacuum cleaners, self-driving cars, photo categorization programs have a common feature: They free us from work that would otherwise be left for us to do.

There is a key difference between man and machine regarding the learning process. If you show a picture of a Pudú and a Kodkod to a human he will be able to correctly classify a set of pictures of these animals. This is not the case with Machine Learning (ML) algorithms however. These algorithms require massive amounts of example data and lots of iterations of training for the machines to be

able to do the same. The boom in Artifical Neural Networks (ANNs) we could see in these past years is due to the immense growth in processing power and data storage capabilities. These two factors enable us to effectively train the ANNs these days.

Another driving factor of the recent the AI boom is user generated data. A lot of services are built upon and improved continually with the data gathered from users through mobiles or their computer browsers. A good example of this would be Google Photos [1] which is a service that can automatically label user photos into categories like cats or dogs. This is achieved after lots of training on data generated dynamically by the users of the application.

There is a problem however with the current popular training methods. All of them rely on users uploading their data into the cloud where the services are trained based on all the data collected from all of the users. There are two issues with this. The first is that users are giving up some of their Privacy. The second is that the data can be very big in size so exchanging it could require a lot of network traffic.

An interesting solution to this problem is Federated Learning which comes from the paper *Federated Optimization: Distributed Machine Learning for On-Device Intelligence* [18]. Here the authors propose a training method where the users need not send their data to the server. Instead, the training of the model is done by the server sending a version of the model to the users and the users evaluate the model and send back the Gradients to the server which aggregates these gradients and updates the model.

In this thesis I propose a new, privacy preserving training methodology called Federated Neuroevolution which is done via distributively evaluating the Fitness Function of the utilized Evolutionary Algorithm.

Chapter 2

Neural Networks

There are some problems in the real world that are just very hard to grasp and formulate using standard algorithmic approach. For example, trying to formalize in code what does a cat look like, or to have an algorithm that separates cat images from dog images we would most likely fail. But our brains and even the brains of very simple animals are in fact very good at this sort of pattern recognition. Trying to understand and simulate them is where the field of Artifical Neural Networks (ANNs) resides.

2.1 Core concepts

With Artifical Neural Networks we are essentially trying to mimic the inner workings of the brain in hopes of solving complex problems which would be near impossible to formulate with a hand crafted Control Flow. [1]

[1] We can not exactly reproduce how the brain works as biologists do not yet have a definitive answer to that. This is because of technical limitations as even the most precise MRI is not precise enough to show us the low level mechanisms of the biological brain. So biologists have a top-down view of the brain whereas we programmers inherently use bottom-up approach when creating an algorithm.

The ANN is a Directed Graph based on a collection of Artificial Neurons. In the graph, the nodes are the neurons and they can have edges between them in any direction. If a neuron receives a signal from an incoming edge it can send out signals on the outward edges. Each neuron evaluates an activation function that determines how strong the activation of the neuron is, and thus how strong the outgoing signal is. The input of the activation function is the sum of inputs to the neuron. The edges have weights to them that modify the strength of the signals traveling on that edge. These signals and weights are represented by real numbers. Inside the neuron, there usually reside some non-linear function that will calculate the output of the neuron (see figure 2.1 for the most common non-linear functions used as activation functions).

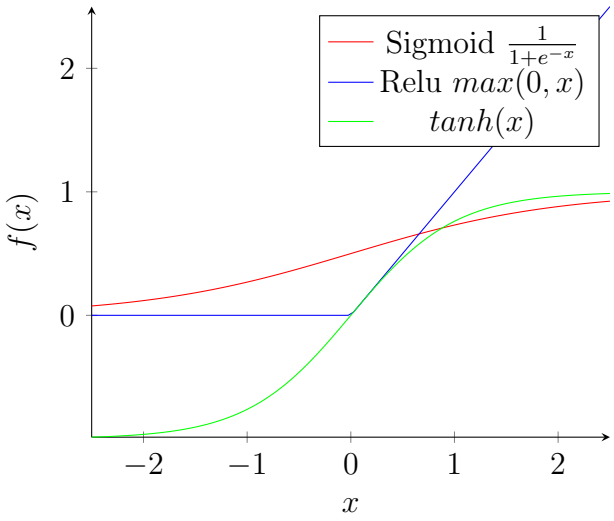


Figure 2.1: Non-linear functions used in ANNs as activation functions

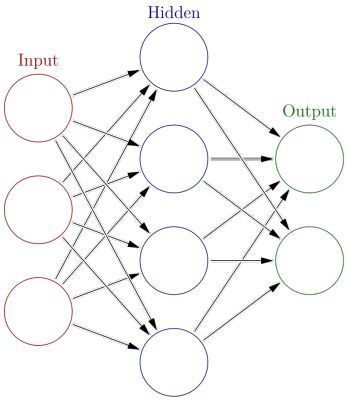


Figure 2.2: A simple ANN topology

The networks are usually following some kind of structure. The most common is the layered structure where multiple neurons are making up a layer and these layers are stacked on each other. A neuron in a layered structure only has incoming edges from neurons in the layer directly below and outgoing edges to neurons in the layer directly above it. In such a layered structure the first layer is called the *input layer* where the stimuli to the network comes in. For example visual data as in each pixel of an

image is stimulating a neuron of the input layer. The last layer is called the *output layer* which will return the result of the calculations. An example output could be 2 numbers where the numbers refer to the likeliness of the input image depicting a cat or a dog respectively. The layers between the input layer and the output layer are called *hidden layers*. Such architecture can be seen in figure 2.2.

2.2 Most Common Neural Network Architectures

As with any directed graph, ANNs can have many shapes, sizes and structures based on which we can categorize them. In this section we will review the different aspects based on which we can categorize the networks.

2.2.1 Deep vs Wide Networks

The networks can be categorized based on the size and number of hidden layers.

We refer to a Neural Network as a Deep Neural Network (DNN) if it has many[2]hidden layers between the input layer and the output layer. This is the most common model used nowadays as it can be trained to recognize patterns with high precision. Example tasks include Image Classification, Natural Language Processing, Fraud Detection, etc. An example of a Deep Neural Network can be seen in Figure 2.3.

A Neural Network is considered wide if the hidden layers have a lot more[2] neurons in them than what is usual. A network being wide does not rule out it being deep as well however it is possible to have just one very wide hidden layer in Neural Network.

[2]These terms are not exactly quantified in the industry as these might be relative to the given application domain.

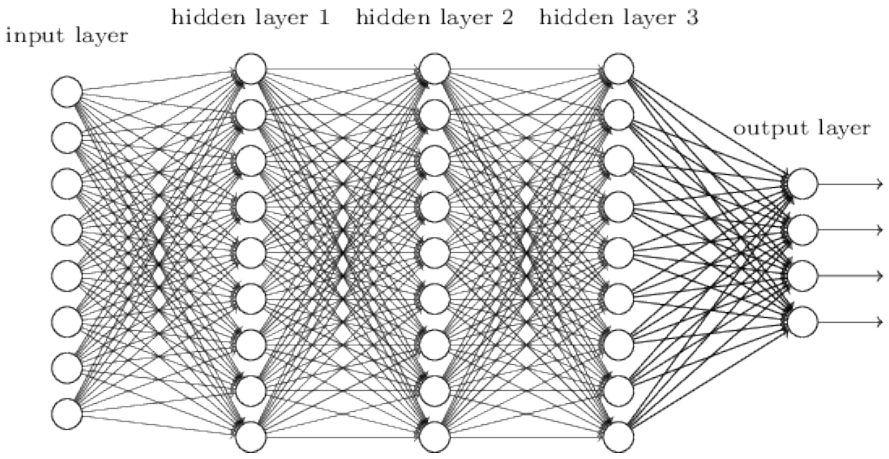


Figure 2.3: A Deep Neural Network topology [2]

2.2.2 The flow of information in Neural Networks

We can categorize ANNs based on the direction of information flow inside them. As mentioned in 2.1, Neurons communicate with other neurons through signals.

Feedforward Neural Networks

Feedforward neural networks have a key distinguishing property: The graph of Neurons is a Directed Acyclic Graph (DAG) [37]. The feedforward neural network was the first and simplest type of artificial neural network created [26]. In this network, the information moves in only one direction, forward, from the input nodes, through the hidden nodes (if any) and finally to the output nodes [37]. Both Figures 2.2 and 2.3 are Feedforward networks.

Recurrent Neural Networks

As opposed to Feedforward Networks the Recurrent Neural Networks (RNNs) have no acyclic constraints. In fact, what makes a Neural Network recurrent is that it contains cycles in it, called feedback loops. These feedback loops create a unique mechanism for these network that allows them to maintain information between inputs. What this means in practice is that a RNN can maintain contextual information,

i.e. Recurrent Neural Networks have memory. See an example Recurrent Neural Network in Figure 2.4.

This memory benefit of RNNs make them fit for solving problems that would be impossible without knowledge of the context. For example accurately predicting the next word the user will type requires not only the information about the last word, but the words before that and, possibly, even the sentiment of the sentences before. Other application areas include translation between human languages, time series prediction, speech recognition, motion picture analysis, just to name a few.

Note that Recurrent Neural Networks can be further divided into many different subcategories but it is out of scope for this thesis to cover them. There is a good overview of them in the *Empirical evaluation of gated recurrent neural networks on sequence modeling* [5].

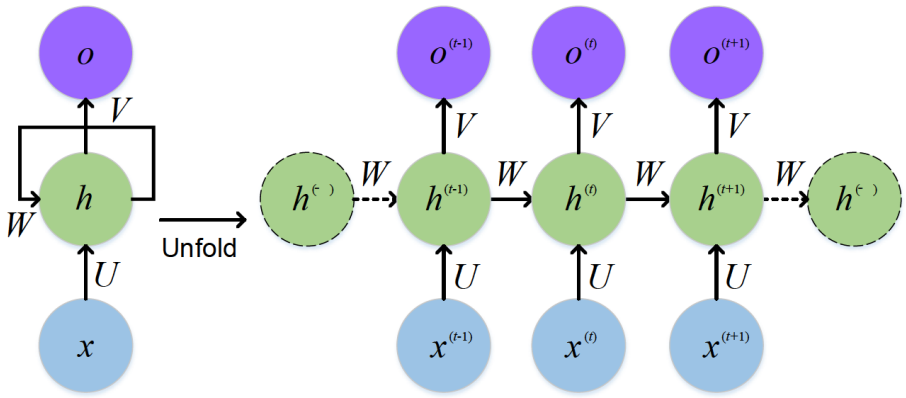


Figure 2.4: A very simple Recurrent Neural Network topology where we only have 1 input neuron, 1 neuron in the single hidden layer and 1 output neuron (left). Note the feedback look in the hidden layer which makes this network recurrent. On the right we can see this loop unfolded. [8]

Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a special type of Deep Feedforward Networks designed based on the Visual Cortex of the human brain in order to mimic it's capabilities at understanding visual imagery using computer vision. The input of

(a) Input matrix and filter matrix (b) Filtering works with matrix multiplication

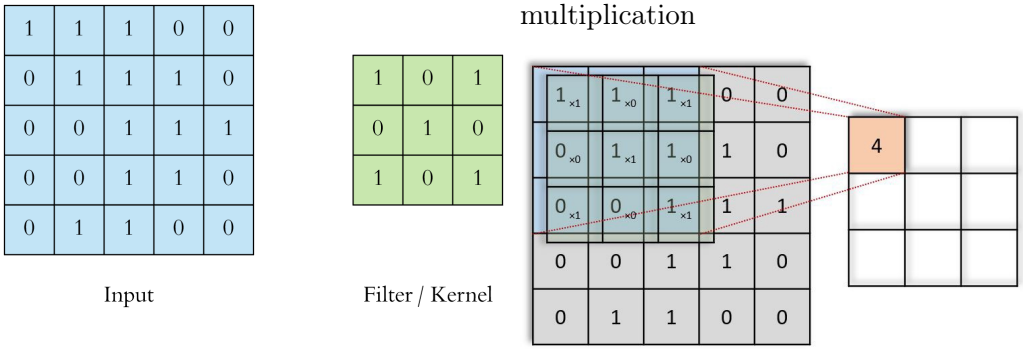


Figure 2.5: Filters example [7]

these networks is usually a 2d image so each neuron in the input layer corresponds to a pixel of the input image. The output of the network is the same as with any other network type.

The main innovation of CNNs lies in the hidden layers where we introduce a new layer type called Convolutional Layer. A Convolutional layer is a set of filters/kernels. Each filter is a small matrix that represents an elementwise multiplication operation as depicted in Figure 2.5. During the training phase these matrix values are modified using the weights to focus on different aspects of the input image like certain shapes or patterns. See Figure 2.6 for an illustration.

The purpose of these Convolutional Layers is to extract certain features of the image. With each added convolutional layer we go from simple features like horizontal or vertical lines to more complex ones like ears and eyes. Based on these higher level features, the classification can be done in the output layer of the CNN.

2.3 Training the network

Mathematically a Neural Network is a function that renders some output to the input and can be defined as $F : X \to Y$ where $X \subset \mathbb{R}^d$ is the input and $Y \subset \mathbb{R}$ is the output of the network. More specifically $f(\mathbf{x})$ is the network which is built up from a composition of neurons $g_i(\mathbf{x})$. Integrating the activation functions we get the

(a) Horizontal Filter (b) Vertical Filter

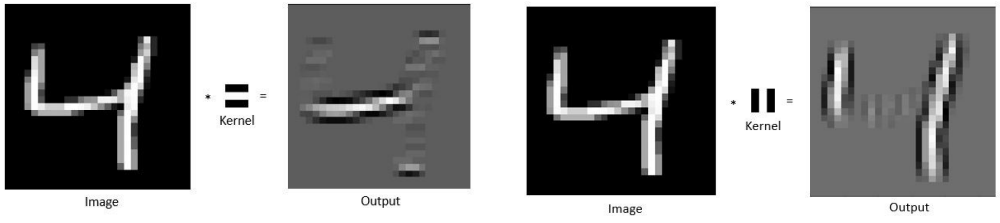


Figure 2.6: Visual outputs of two filters [16] on an example from the MNIST Data [34]

$f(x) = K(\sum_i w_i g_i(x))$ as the Neural Network mathematical formula where w are the weights of the network and K is an activation function. In this thesis I will only consider Supervised Learning scenarios where we have a set of example input-output pairs in the form of $\{\mathbf{x}_i, y_i\}_{i=1}^n$ where $\mathbf{x}_i \in \mathbb{R}^d$ and $y_i \in \mathbb{R}$ while n is the number of input data points (the size of the data set) at hand.

The training of the neural network is the process during which we achieve a state of the network's weights which is considered optimal. Optimality is measured by a cost or loss function that renders a real number to a function $f \in Y^X$. The form of loss function is $C : Y^X \times \mathbb{R}^d \to \mathbb{R}$. A very popular loss function is the Mean Squared Error which is defined as in equation 2.1

$$MSE(f) = \frac{1}{n} \sum_{i=1}^{n} (f(\mathbf{x}_i) - y_i)^2 \tag{2.1}$$

Before the training process can start, we need to initialize the weights. This is usually done by randomizing the weights to some small non-zero numbers.

During the training, in each iteration we evaluate the current f_i network using our loss function. This will indicate how far is f_i from the optimal state. The problem is how do we determine f_{i+1} so that $MSE(f_i) > MSE(f_{i+1})$.

For this we have a family of methods called Backpropagation. These methods calculate the gradient of the loss function and modify the weights based on that. One of these methods is Stohastic Gradient Descent (SGD) which is formalized as

$$w_{ij}(t+1) = w_{ij}(t) - \eta \frac{\partial C}{\partial w_{ij}} \tag{2.2}$$

where η is the learning rate, C is the cost (loss) function.

We have many optimization methods that are based on the gradient descent trick of backpropagation. Stohastic Gradient Descent updates the weights after each training example. The loss function will be fluctuating quite heavily because of this. Reviewing all the different, currently used optimization techniques is outside of the scope of this thesis. For those interested I recommend checking *An overview of gradient descent optimization algorithms* by Sebastian Ruder [24].

Chapter 3

Federated Learning

In October 2016, Jakub Konečnỳ, H Brendan McMahan, Daniel Ramage, and Peter Richtárik published *Federated Optimization: Distributed Machine Learning for On-Device Intelligence* [18], where they introduced the concept of Federated Learning. Because this thesis is largely built upon the ideas introduced in their publication, I will summarize their article in this chapter.

3.1 Problem setting

When we are talking about Machine Learning problems we usually refer to a mathematical function that can approximate a problem defined with input-output pairs after setting the correct weights of the function. Usually we have a set of input-output pairs $\{\mathbf{x}_i, y_i\}_{i=1}^n$ where $\mathbf{x}_i \in \mathbb{R}^d$ and $y_i \in \mathbb{R}$. Based on a loss function we can iteratively approximate the original f by computing our f_i loss function. Formally, the function we are searching for looks like

$$\min_{w \in \mathbb{R}^d} f(w) \qquad \text{where} \qquad f(w) = \frac{1}{n} \sum_{i=1}^{n} f_i(w) \tag{3.1}$$

As an example, in the case of the famous MNIST Dataset [34], all of the $\mathbf{x}_i$ is a greyscale image of size 28x28 ($\mathbf{x}_i \in \mathbb{R}^{28 \times 28}$). The output y_i is a vector signaling which digit it is on the image ($y_i \in \mathbb{R}^{10}$). The dataset contains 70000 images in total,

so $n = 70000$. We can simply download the data and build a Convolutional Neural Network (CNN) (as explained in 2.2.2) to solve this classification problem.

Solving such problems has undergone heavy research in the past decades and is quite well understood now. However, in a lot of the industry settings the problems are not that simple. The data is often very fragmented and heavily distributed. Users generate a lot of data with their mobile phones and gathering the data from all of them to build a model on them is impossible in most cases as we can't fit this amount of data into a single computer.

Even if users could upload the data to a server, the concern of privacy is raised. In recent years, there are more and more so-called Privacy Aware users who no longer want their data being sent to and stored at a cloud.

A naive idea would be to train the models on the nodes themselves, where the data is. The problem with this is twofold. The first problem is that a node may not contain all the kinds of data we want to handle in our model. For example a dog person would only have photos of his/her dog so we could not build a model of classifying dogs versus cats relying on his phone alone. The second problem is that data volume may vary heavily from node to node. For example, a less active user does not have much data that can be of use. The above explained properties we will call Federated Properties and can be summarized as in the list below.

- **Massively Distributed** Data points are stored across a large number of nodes K. 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.

3.2 Industry Solutions

The usual answer to this problem in the industry is to collect the data from the users - however costly this may be - and solve the problem in a centralized location where it could be shuffled and distributed evenly over proprietary compute nodes. Such distribution could be handled using Spark [35] by evenly distributing data to the nodes and run the single core baseline algorithms over the nodes. This, however, is a very hard constraint and does not conform with the Federated Properties so a heavy data lifting step is required as pre-processing.

Another good solution to the distributed calculation is the CoCoA+ framework, which require minimal changes for distributed calculation compared to the single node version and can use any single node base algorithm to be distributed among the nodes [21] [28]. With CoCoA+ the data can be provided to the framework as-is, so the data preprocessing step can be spared. The downside of CoCoA+ is that it is constrained to the dual form of the optimization problems where $y_i \in \{-1, 1\}$ and the algorithm converges very slowly.

Both of these solutions have in common the need for synchronization in each iteration. This is due to aggregation of partial results from the nodes needed at each iteration step. This is a hard constraint and has a significant impact on the overall performance of the system.

3.3 New approach: Federated Learning

Federated Learning or *Federated Optimization* is a new kind of Distributed Optimization where users or nodes do not send the data they generate to the server, but rather provide part of their computational power to be used to solve the optimization problem. It's purpose is to efficiently solve Machine Learning problems wich have Federated Properties.

The Algorithm proposed in the original paper [18] is a modified version of the Stochastic Variance Reduced Gradient (SVRG) [15, 19] algorithm which itself is

based on the widely acknowledged Stohastic Gradient Descent (SGD) explained in the section 2.3. Let us first examine the original SVRG algorithm, described in Algorithm 1. There are some common variables we will be using in these algorithms that are listed in the below list.

- n — the number of data points / training samples
- $n^j = |\{i \in 1, ..., n : \mathbf{x}_{i_j} \neq 0\}|$ — the number of data points with nonzero j^{th} coordinate
- $\phi^j = n^j/n$ — frequency of appearance of nonzero elements in j^{th} coordinate

Algorithm 1 SVRG

procedure SVRG(m, h) ▷ m number of stochastic steps per epoch, h the step size
  Initialize w^t randomly
  for $s = 0, 1, 2, ...$ **do**
    Compute and store $\nabla f(w^t) = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(w^t)$
    $w = w^t$ ▷ w^t is randomly initialized
    for $t = 1$ to m **do**
      Pick $i \in \{1, 2, ...n\}$, uniformly at random
      $w = w - h(\nabla f_i(w) - \nabla f_i(w^t) + \nabla f(w^t))$
    end for
    $w^{t+1} = w$
  end for
end procedure

The modified version is called Federated Stochastic Variance Reduced Gradient (FSVRG) and can be examined in Algorithm 2. To understand it, we need a few new variables such that:

- K — the number of nodes

- $\mathcal{P}_k$ — set of indices, corresponding to data points stored on device k
- $n_k = |\mathcal{P}_k|$ — the number of data points on node k
- $n_k^j = |\{i \in \mathcal{P}_k : \mathbf{x}_{i_j} \neq 0\}|$ — the number of data points stored on node k with nonzero j^{th} coordinate
- $\phi_k^j = n_k^j / n_k$ — frequency of appearance of nonzero elements in j^{th} coordinate on node k
- $s_k^j = \phi^j / \phi_k^j$ — ratio of global and local appearance frequencies on node k in j^{th} coordinate
- $S_k = Diag(s_k^j)$ — diagonal matrix, composed of s_k^j as j^{th} diagonal element
- $\omega^j = |\{\mathcal{P}_k : n_k^j \neq 0\}|$ — Number of nodes that contain data point with nonzero j^{th} coordinate
- $a^j = K/\omega^j$ — aggregation parameter for coordinate j
- $A = Diag(a^j)$ — diagonal matrix composed of a_j as j^{th} diagonal element

Now let's explore a bit more on why this algorithm works well in the federated setting. Below is a list of some key points of the algorithm:

1. **Introduced local step size ($h_k = h/n_k$)** One of the key problems I named in section 3.1 was that the distribution of data between the nodes can be heavily Unbalanced. We can clearly see that if a node has thousands of data points we require smaller step size in each iteration of data point than for a node where we have only a few data points. The local step size helps us to even out the data size differences and achieve roughly the same magnitude of weight progress on each node regardless of n_k.

Algorithm 2 FSVRG

procedure FSVRG(step size h, data partition $\{\mathcal{P}_k\}_{k=1}^K$,
diagonal matrices A, $S_k \in \mathbb{R}^{d\times d}$ for $k \in \{1, ..., K\}$)
  for $s = 0, 1, 2, ...$ **do** ▷ Overall iterations
    Compute $\nabla f(w^t) = \frac{1}{n}\sum_{i=1}^n \nabla f_i(w^t)$
    for all $k = 1$ to K **do in parallel** over nodes k ▷ Distributed loop
      Initialize: $w_k = w^t$ and $h_k = h/n_k$
      Pick $i \in \{1, 2, ...n\}$, uniformly at random
      Let $\{i_t\}_{t=1}^{n_k}$ be random permutation of $\mathcal{P}_k$
      for $t = 1, ..., n_k$ **do** ▷ Actual update loop
        $w_k = w_k - h_k(S_k[\nabla f_{i_t}(w_k) - \nabla f_{i_t}(w^t)] + \nabla f(w^t))$
      end for
    end for
    $w^{t+1} = w^t + A\sum_{k=1}^K \frac{n_k}{n}(w_k - w^t)$
  end for
end procedure

2. **Aggregation of updates proportional to partition sizes ($\frac{n_k}{n}(w_k - w^t)$)** Intuitively we can understand that an update coming from a node with thousands of data points should have a bigger weight than an update coming from a node with only a few data points. Aggregating weight updated with respect to the size of n_k compared to n solves this issue.

3. **Scaling stochastic gradients by diagonal matrix** S_k In our problem setting (section 3.1) I said that the data could be distributed so that a node does not have an IID local portion of the data. Imagine a scenario where one user has thousands of images of a dog, but none from the other class of the cats. In such scenario, this node should not have a large impact on the gradients of the network that are related to cat classification. This point does that exactly through scaling the weight updates on the node in accordance with the node's data distribution.

4. **Per-coordinate scaling of aggregated updates** $A(w_k - w^t)$ This scaling goes hand in hand with Point 3. In Point 3, we scale down the updates for classes the node does not have on the client side. In Point 4, when we aggregate on the server side we scale the values of nodes according to their class distribution.

3.4 Why would we switch to Federated Learning?

Federated Learning certainly requires the adoption of new viewpoints which requires time and effort from the industry that is already used to the solutions discussed in Section 3.2. These solutions have been working well for the industry so why should they change?

One reason to change is that users are becoming more and more aware of their privacy with each privacy related scandal. This created a new environment in which tech companies focused on privacy have an existing and emerging user demand to supply for.

A good example of this is that at the Google I/O 2019 conference a lot of the emphasis was on distributed Machine Learning. Google CEO Sundar Pichai said that "Gboard is already using federated learning to improve next-word prediction" and Google's Senior Director of Android, Stephanie Cuthbertso said that "On-device machine learning powers everything from these incredible breakthroughs like Live Captions to helpful everyday features like Smart Reply. And it does this with no user input ever leaving the phone, all of which protects user privacy" [6].

Another driving factor could be to decrease the investment required in centralized computational power. This seems reasonable as data is ever growing and the user's phones are idle at most of the time which makes their utilization really poor.

The final driving factor could be political pressure. There could come a time when countries start regulating the privacy requirements of software. If such political changes ever come, Federated Learning will be there in our toolbox to conform with the new regulations.

3.5 Challenges of Federated Learning

Federated Learning is a new area and with every new field of research there is still room for improvement. The below list gives a summary of key weaknesses that are identified by the authors of Federated Learning [18].

1. The proposed algorithm is synchronous and each iteration of the global model requires synchronization with all the nodes. This is the main performance bottleneck of the algorithm as per node wait times can vary heavily based on the amount of data available on the node and the computational power of the node. An Asynchronous research on Federated Learning would be desirable.

2. For non-convex problems like Neural Networks there are no convergence guarantees of the FSVRG algorithm.

3. There is still some privacy leaked from the nodes through the gradients and the

per node class meta data. Both of these are essential for the FSVRG algorithm despite these still giving away some part of the user's privacy.

4. With the FSVRG algorithm we are building a global model. This is good in general but we could improve user experience if the model was biased towards the user's local data. This is certainly an improvement that could be done for users with big amount of local data.

Chapter 4

Evolutionary Algorithms for Federated Learning

As discussed in 3.4 there is still some privacy related concerns with the FSVRG algorithm, described in algorithm 2. The proposed algorithm exposes class distribution of the nodes and the gradients also contain some information about the actual data on the nodes [18]. In this thesis, I propose a new method for Federated Learning that is completely different from FSVRG but still solves the problem introduced in the section 3.1.

The method I suggest is a modified Evolutionary Algorithm (EA). As we will see, these algorithms do not require any knowledge about the data they are being tested on as, in general, an EA can handle the data as a black box. This eliminates the need to know about data distribution on the node as well as gradients are no longer needed either.

4.1 What are Evolutionary Algorithms?

First let us discuss what are Evolutionary Algorithms (EAs). Back in chapter 2, I mentioned that Artifical Neural Networks are based on nature itself. However, as we have seen in section 2.3, we have constructed our own methodology for training the

network that could be different than how the biological neural networks are trained in living beings.

With Evolutionary Algorithms the programming community has yet again reached out to concepts that we have observed in nature. Evolutionary Algorithms belong to the family of Evolutionary Computation in which we use heuristics and stochastic processes for global optimization.

Evolutionary Algorithms are based on terms we know from Darwinian Evolution. The main terms in biology are fitness, selection, reproduction and mutation. These have very similar names and definitions in our field which are fully explained in the following list:

- **Individual** An individual is an instance of the solution to the problem that we are trying to optimize. Individuals are problem specific. An individual could be for example a vector of values each representing a hyperparameter of a training algorithm, or the weights of an ANN.
- **Population** A list of individuals
- **Generation** The population of a specific iteration of the algorithm
- **Fitness** A real number indicating the goodness of an individual
- **Selection** The process of selecting the best individuals of a generation to for reproduction
- **Crossover** The reproduction process during which we create new individuals - called offsprings - by some form of mixing of selected parents
- **Mutation** The process of randomly changing the offsprings

The most basic form of EA is explained using pseudocode in Algorithm 3. The power of the algorithm is that it is very generic and can be used for solving any optimization problem. To apply to a specific problem we only need to define the representation of our individuals and the functions for operating on them.

Algorithm 3 Evolutionary Algorithm

create population $\mathcal{P}$ randomly
$n = |\mathcal{P}|$
while $\neg$ fitness_goal_is_met($\mathcal{P}$) **do**
    $\{f_i\}_{i=1}^{n} = \{fitness(\mathcal{P}_i)\}_{i=1}^{n}$ ▷ Evaluate fitness of population
    $parents$ = best_individuals($\mathcal{P}$, f) ▷ select the best individuals for breeding
    $offsprings$ = crossover($parents$) ▷ the newly bred individuals
    for $offspring$ **in** $offsprings$ **do** ▷ Mutate the $offsprings$
        $offspring$ = mutate($offspring$)
    end for
    $\mathcal{P} = parents \cup offsprings$
end while

4.2 EAs for Neural Networks

Neuroevolution (NE) is a form of artificial intelligence that uses evolutionary algorithms to optimize the parameters and topology of Artifical Neural Networks (ANNs) [30]. There are certain situations where input-output pairs are not available for training a network using gradient based techniques. Neuroevolution excels in these situations because the Evolutionary Algorithm's fitness function is so generic it can be applied in a lot more cases. Such fields include but not limited to are Evolutionary Robotics, Artificial life and General game playing.

One very famous example of the application of Neuroevolution is an AI developed by Sethbing where he uses NE to train a network that can beat Mario [27]. This is a perfect example of Neuroevolution where gradient based methods are not accessible as we do not have training data. The fitness function here was simply measuring how far did the agent get in the level.

Neuroevolution has many implementations which differ in key features. In the following sections we will review these traits.

4.2.1 Training scope

In the classic gradient based learning we have a fixed network topology where we are only adjusting the weights of the network using backpropagation, as can be seen in the section 2.3.

In the case of Conventional Neuroevolution we are learning the weights of an ANN using Evolutionary Algorithm and not modifying the topology. The problem with this is that without gradients we do not know the slope so the learning is very stochastic. The benefit of Conventional Neuroevolution, however, is that it can avoid being stuck in local minimums with the right Hyperparameters. Such an algorithm is called *Neuro-genetic evolution* by E. Ronald [23].

Another class of Neuroevolution algorithms are Topology and Weight Evolving Artificial Neural Network algorithms (TWEANNs). These algorithms not only optimize the weights of ANNs but also adjust the network topology. This is the main reason for existence of Neuroevolution as this is something that can not be done with regular training methods. The two most famous TWEANNs are *NeuroEvolution of Augmenting Topologies* (NEAT) [31] and *Hypercube-based NeuroEvolution of Augmenting Topologies* (HyperNEAT) [10].

4.2.2 Encoding

Another dividing aspect of Neuroevolution algorithms is the way they encode information about the ANNs, as introduced in the following paragraphs.

Direct Encoding

Direct Encoding is the trivial way of storing network information: we directly store the full network topology and all of the weights in each individual of the population. The issue with this 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 solution to the scaling problem of Direct Encoding can be solved with Indirect Encoding. The way to handle this problem is to compress the information and only store part of it or generative information about it. In this case each individual only stores information about how to generate the network.

It is possible to store only the network topology in the genetic information. The fitness function in this case can calculate some metrics for the individual after training for some epoch using a regular gradient based learning method.

It is also possible to store compressed information about the weights like it is done in HyperNEAT [10].

4.3 Federated Neuroevolution

In this section I propose a solution on how to use Neuroevolution to solve the Federated Learning problem described in the section 3.1. In short, the main idea of this algorithm is that the server generates a new generation of models and sends these to all the clients. The clients evaluate the fitness of each model and send it back to the server for aggregation. The server then generates a new generation of models and the loop starts again.

To use Neuroevolution in a Federated setting, we need not modify the base Evolutionary Algorithm extensively. The only point where we need to interfere is at line 4 of Algorithm 3. The fitness function acts as a black box from the perspective of the main algorithm so no other change is needed. The proposed fitness function is described in Algorithm 4.

We will define a distributed fitness function that sends the models to the nodes. The nodes locally evaluate the models and send back their fitness values as measured on local data. Of course, this can be done parallel on all the nodes.

Because of the Non-IID and Unbalanced properties of the node data distribution I am not considering indirect encoding options, as optimization done on the node

could diverge the model heavily due to local data properties. With direct encoding we are keeping a strong control of the evolution on the server side, thus, clients can not overfit the models too much. The drawback of the direct encoding is the larger search space. Because of this, we will only be learning weights of the network and not its topology.

Algorithm 4 Federated Fitness

procedure FEDERATED_FITNESS(nodes $\{\mathcal{N}_k\}_{k=1}^{K}$, population of models P)
    $n = |\mathcal{P}|$
    for all $k = 1$ to K **do in parallel** over nodes k ▷ Distributed loop
        $\{f_i^k\}_{i=1}^{n} = \{fitness(\mathcal{P}_i)\}_{i=1}^{n}$ ▷ Evaluate fitness of population on node k
        $n^k = |\mathcal{N}_k|$ ▷ Get the number of training examples of the node
    end for
    $\{f_i\}_{i=1}^{n} = \{\frac{\sum_{j=1}^{K} n^j * f_i^j}{\sum_{j=1}^{K} n^j}\}$ ▷ Get a weighted average of the node level fitnesses
    return f
end procedure

Chapter 5

Case Study

In this chapter, I will show how the theory behind Federated Neuroevolution is capable of solving a problem through a case study.

5.1 The EEG Alcohol Dataset

As Federated Learning is a relatively new concept, thus, there are no datasets publicly available that are already split into nodes. The original paper also didn't have such a dataset, but created one by splitting a dataset into virtual nodes based on users posting on Google+ [18]. The splitting was done such that each user represented a node.

The dataset I have used is the EEG Database Data Set [4]. The dataset contains 120 Electroencephalography (EEG) trial data about 122 patients who either belong to the alcoholic or to the control group. In each trial the patients were shown 1 or 2 image 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. A trial contains the following data (an example trial data can be seen in the Appendix 1):

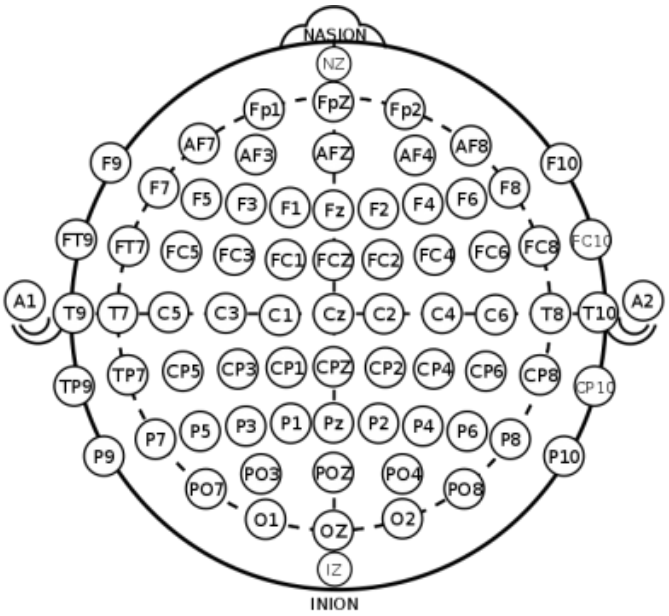


Figure 5.1: EEG Points of measurements on the scalp [17]

- **Stimulus type** Can be one of the following: *S1, S2 match, S2 nomatch*. S1 means that only 1 image were shown. S2 match means that 2 images were shown of the same class, S2 nomatch means that 2 images were shown from different classes.

- **Trial number** The number of the trial

- **Measurement data** Measurement data contains the place of sensor on the scalp, the relative time of measurement and the measured value by the sensor in microvolts.

The dataset itself is very fascinating. It raises the question: Does alcoholism effect brain functionality and, if so, then is it visible in EEG? Before we use machine learning to answer that let us take a look at the data, shown in figures 5.2 and 5.3, to get some intuition. Based on these images we can assume that there is indeed a connection between alcoholism and brain functionality and that this can be observed through EEG.

I think this dataset makes a good candidate for Federated Learning. Let's assume

(a) Alcoholic (b) Control

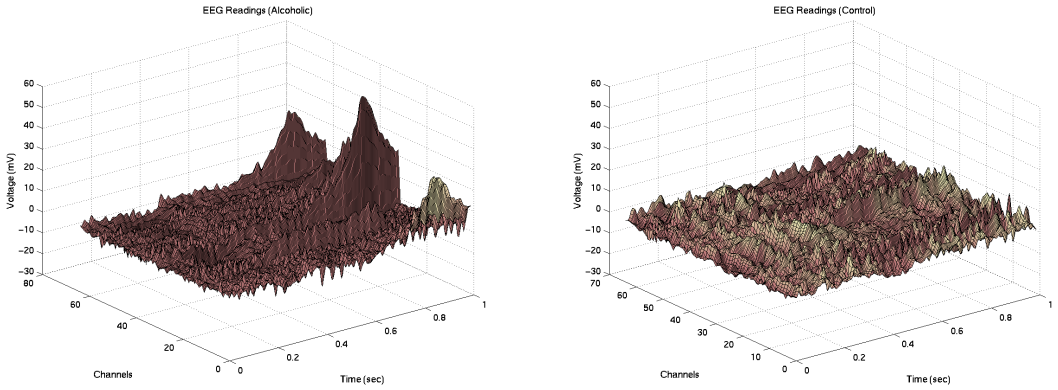


Figure 5.2: Average of 10 trial results for single image stimulus. [4]

(a) Alcoholic (b) Control

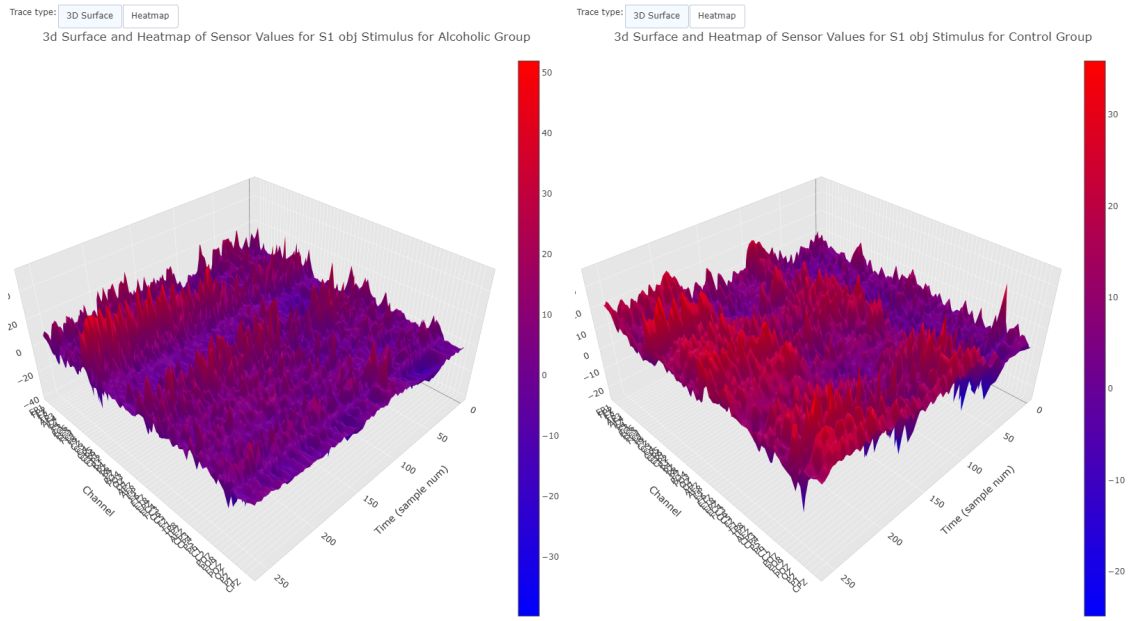


Figure 5.3: Randomly selected trial of single image stimulus. [17]

that this data is hard to come by, and a lot of hospitals have only a few EEG samples from a handful of patients. The hospitals could not train a model based on their small local dataset but combined globally they could form a large enough dataset for training. Also due to GDPR and other regulations regarding medical data it is quite likely that the hospitals are legally prohibited from sharing the medical data with any third party. This is where Federated Learning comes in as a possible solution.

So in my setting the hospitals are the nodes and the data is randomly distributed between them so that all three Federated Properties, listed in section 3.1, are met. See Appendix B.1 for a visualization of the data distribution used for training.

5.2 Implementation

The entire source code related to the thesis can be found on Github[1].

With the Machine Learning boom of recent years came new code libraries to provide reusable code to the community. These libraries were mostly written in Python as it is a very easy to learn and concise language that is even appealing to people not from a software background. By this time Python has become the *de facto* ML language so I have used Python3 for the coding part of my thesis.

There are a lot of libraries for ML in Python for data manipulation, machine learning, neural networks and data visualization that we can use. For the exact libraries I used please refer to Table 5.1.

[1] `https://github.com/VSZM/FederatedNeuroevolution/releases/tag/mscthesis`

Library	Description
Numpy[2]	Providing math support functions. Especially strong in matrix representation and manipulation.
Tensorflow [3]	A library for creating and training Neural Networks on GPUs.
Keras [4]	Simple interface for creating and training Neural Networks using Tensorflow, CNTK or Theano as the underlying backend.
scikit-learn [5]	Providing all kinds of Machine Learning in classification, regression, clustering, preprocessing.
seaborn [6]	High level Data Visualization library built on top of matplotlib.
pandas [7]	Data analysis tool.

Table 5.1: Most relevant libraries used for the case study

5.2.1 Setting up a baseline

To measure the performance of a new methodology we need to setup a baseline to refer to. I have created a model on a single node as a baseline to compare my method's performance against it.

Looking at the visualization of EEG data at Figures 5.2 and 5.3, we can see that if we are to represent the measurements in a 2d heatmap image we could clearly distinguish the two classes from one another. Convolutional Neural Networks are good at classifying images so I have chosen to build a CNN for this problem.

After checking the related research in EEG data classification, I have found that others have successfully used CNNs in this field. The most useful information I learned was that Convolutional Filters should be defined so that each electrode is

[2] `https://www.numpy.org/`
[3] https://www.tensorflow.org/
[4] https://keras.io/
[5] https://scikit-learn.org/stable/
[6] https://seaborn.pydata.org/
[7] https://pandas.pydata.org/

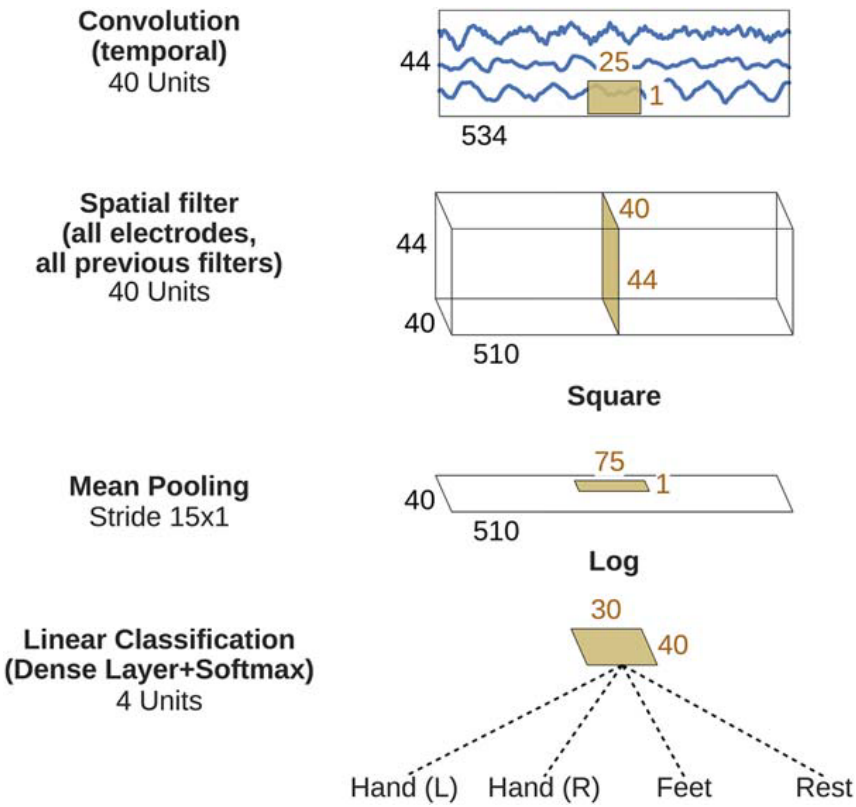


Figure 5.4: Reference network for EEG data classification [25]. Note the kernel size of the first layer where each electrode is handled separately.

measured separately [25]. A reference network that does this can be seen at Figure 5.4.

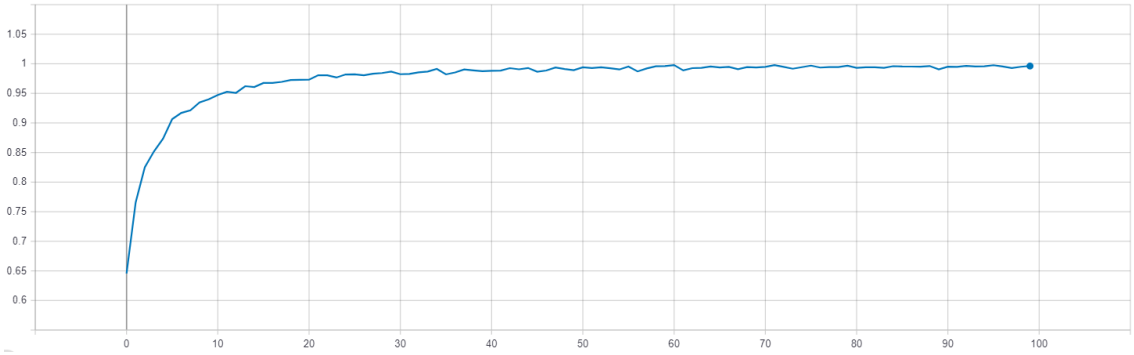
In my implementation, I first had to do some pre-processing where I read in the textual data of the EEG measurements. As I mentioned in Section 5.1, there are 64 electrodes on the scalp and each of them do 256 measurements over 1 second. This results in a 64×256 matrix where each matrix element is a measurement in microvolts.

The CNN I created has 3 Convolutional layers so it can be considered shallow. After the first 2 convolutional layers there is a pooling layer and then comes the 3rd convolutional layer. I inserted 1 batch normalization layer to avoid overfitting and speed up the learning. The activation function I use is sigmoid. I used the Adadelta optimizer [36] which has adaptive learning rate. The loss function I use is Categorical Cross-Entropy [11].

The keras code for the network configuration can be seen in Appendix 2 and the network topology summary can be seen in Appendix 3.

The training took 100 epochs and the results are quite satisfying with a maximum validation accuracy of 95%, as can be seen in the figure 5.5.

(a) Training accuracy over 100 epoch



(b) Validation accuracy over 100 epoch

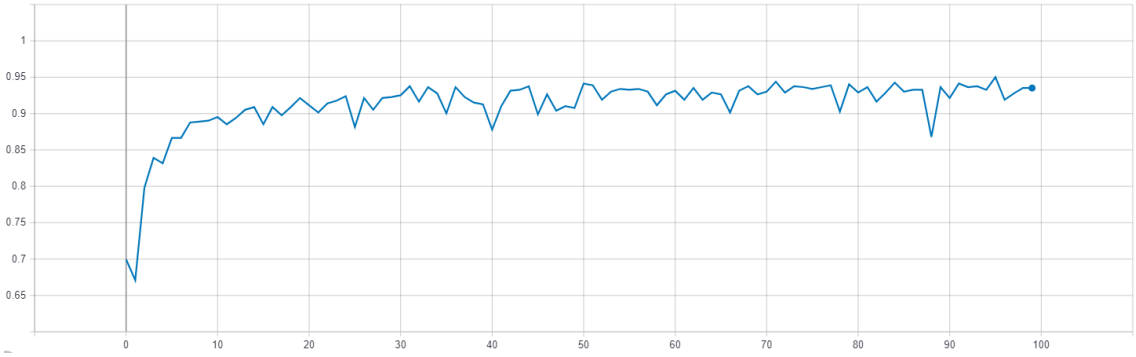


Figure 5.5: Accuracy during baseline training

5.2.2 Federated Neuroevolution Solution

Despite Neuroevolution has been around for more than 2 decades, we do not see any widely used, mature, libraries that would help implementing it. This could be because of Evolutionary Algorithms are so generic that most of the implementation is problem specific and the reusable part is not much hard to implement.

Before creating my own implementation, I looked around to see how others implemented Neuroevolution so far. My initial approach was based on an article I found on "Towards Data Science" [9] which I had to modify extensively in order to work with this problem domain.

Below I go over the implementation in details and what led me to implement Algorithm 3 the way I did it.

Crossover function considerations

One of the most important part of the Evolutionary Algorithm is the creation of offsprings in each iteration. The two main aspects to consider is how many parents should take part in a single crossover and how do we implement the crossover of selected parents.

We can consider choosing multiple parents for the offsprings but the usual way to do is to chose just two parents for each offspring. I also only choose two parents randomly from the pool of parents to produce the required offspring amount. The pool of parents 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. The amount of parents selected for crossover and the pool size is driven by hyper-parameters of the algorithm that can be found in Appendix 4 denoted by *num_parents_mating* and *population_size*, respectively.

After choosing the parents for crossover, we need to do the actual crossover. This can be done in many different ways depending on our model representation. Below is a list of different crossover approaches I considered during my thesis. The first three require us to convert the Artifical Neural Network's layered matrix structure into a flat vector of values. These first three approaches are rather popular in Genetic Algorithms.

- **Halving mix:** In this approach, 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 [9]. Formally:

$$\{offspring_i\}_{i=1}^{n} = \begin{cases} a_i, & \textbf{if } i <= n/2 \\ b_i, & \textbf{if } i > n/2 \end{cases} \tag{5.1}$$

  where n is the length of the model vectors and $\mathbf{a} = (a_1, a_2, \ldots, a_n)$, $\mathbf{b} =$

$(b_1, b_2, \ldots, b_n)$ are the parent vectors.

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

$$\{offspring_i\}_{i=1}^{n} = \begin{cases} a_i, & \textbf{if } i \bmod 2 = 0 \\ b_i, & \textbf{if } i \bmod 2 = 1 \end{cases} \tag{5.2}$$

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

- **Mean mix:** In this approach, 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:

$$\{offspring_i\}_{i=1}^{n} = \{\frac{a_i + b_i}{2}\} \tag{5.3}$$

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

- **Kernelwise mix:** This is the only approach where the layered network structure is not modified. In fact, this one builds on the layered structure of the network. In each layer there are multiple kernels/filters that hold key pattern information. These information portions are kept intact during crossover. The offspring model is created by randomly mixing the kernels inside each layer. See kernelwise mix implemented in Appendix 5.

My initial experiments did not converge using the first three crossover methods. 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, I invented on my own, after taking a look at how genetics work in nature. In nature, the heredity is also a higher level

mixing of genes[8], 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 kernelwise mixing the evolutionary training was converging so I chose this approach.

5.2.3 Mutation function considerations

As the initial population is fully randomized, crossover is not enough guarantee for convergence. Just like in nature, we need mutation. The problem with mutation is that - just like in nature - it is not a heuristic approach, so changes are random by design, meaning that the outcome can either be good or bad from our point of view.

We first need to decide the number of mutated values and the scale of the mutation on these values. The former will be controlled by a 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. These are, again, driven by hyper-parameters of the algorithm that can be found in Appendix 4 denoted by $mutation\_chance$ and $mutation\_rate$, respectively.

There are the following two main approaches I saw for mutating values in a network:

- **Mutate by offset [9]:** Here we add a random value to the selected values. In my implementation, the offset was a random value between $[-mutation\_rate, mutation\_rate]$.
- **Mutate by multiplication [22]:** Here we multiply the selected values with a random value. In my implementation, the multiplication factor was a random value between $[\frac{100-mutation\_rate}{100}, \frac{100+mutation\_rate}{100}]$.

[8] A gene is a region of DNA that encodes a trait or a function

After experimenting, I found lot better convergence rate with the second approach, so I chose that one. The python code implementation can be found in Appendix 6.

5.2.4 Federated fitness function

The Federated fitness function is almost the same as I have laid out in Algorithm 4. Indeed, it needs an actual fitness function that can calculate the node level fitness which was abstracted in Algorithm 4. For this, I have chosen Negative Mean Squared Error which can be written as in the Equation 5.4. The reason for the negating is that the core algorithm is searching for max values when determining which is the fittest individual of a population. The related source code can be seen in Appendix 7.

$$NMSE(f) = -1 * MSE(f) \tag{5.4}$$

where MSE is introduced in Equation 2.1.

Of course, to run Federated calculations, we need to separate the data to nodes. I have spared the technicalities of creating physical nodes and distributing the data, so I have created the `Node` class as an abstraction. The Node class implementation can be found in Appendix 8.

5.2.5 Avoiding overfitting

During my initial runs, where I could achieve convergence, I realized that I did not keep over-fitting in mind. Of course, as with any Supervised Machine Learning technique, the Evolutionary Algorithm is also capable of over-fitting on the training data if we are not careful.

Avoiding this has been studied in Evolutionary Algorithms as well [14, 20, 13, 12]. 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.

Because our setting is a Federated setting, we cannot directly apply these techniques. The problems are that the data is distributed on a multitude of nodes, and, not only is it distributed but is also (most probably) unbalanced. Because we are taking privacy into consideration we want to know as little about the client data on the server as possible. This means that we ideally shouldn't tell the clients which data samples to include in the current generation.

This has led me to the conclusion to do the subset selection at a higher level and treat the nodes as data samples. So, in each generation, the Federated Neuroevolution algorithm selects a subset of the nodes for evaluating the fitness of the current generation. This subset selection is done by iterators at a code level. In my experiments, I have tried the following three approaches:

1. **Random single element for each generation:** Here, I select randomly a node in each generation and ask the node to evaluate the population's fitness on a randomly selected single training sample of it's own. In my experiments, the training did not converge at all with this method.

2. **Random subset for each generation:** Here, I randomly select a subset of the nodes and send the models only to them for evaluation on their full training set. I re-randomize the subset every n generations. In my experiments this was the best approach for Federated Neuroevolution.

3. **Moving window subset for each generation:** Here, I first order the nodes and then select a slice of the list of nodes. This is the window and in every n generation I move the window to the right by 1. The list is a cyclic list so if I run out of indices I start over at the beginning. In experiments, the 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 second and third approach both require two parameters: The subset size and the change interval driven by hyper-parameters of the algorithm that can be found in Appendix 4 denoted by *node_activation_ratio* and *node_subset_change_interval*, respectively.

The second approach of randomly selecting a subset of nodes had the best performance. I programmed these approaches as iterators that return a subset of the list of nodes. The code can be found in Appendix 9.

5.2.6 The main algorithm

Now that we have went over the important parts of the algorithm, the only thing left is its main loop based on Algorithm 3, with the following additions:

- **Validation:** On the server, I retain a validation set and in each generation I 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 local maximums:** Based on the history of validation accuracies, I check the last n entries for a match with the current validation accuracy. If there is a match, I conclude that the evolution has reached a local maximum and 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, I reset the values of the mutation rate and mutation chance to the original values. There is an upper bound on the mutation multiplier. These values are controlled by the hyper-parameters *stuck_check_length*, *stuck_multiplier*, *stuck_evasion_rate* and *stuck_multiplier_max*.
- **Saving best models:** I save the fittest model of each generation.

The developed Federated Neuroevolution approach can be seen in Appendix 10.

5.2.7 Results

Lastly, I present the results of running the evolution for 5000 generations. See Appendix B.2 to see the fitness values and the validation accuracies during the training. We can observe that the convergence was slow but steady, overall.

	Minimum value	**Maximum value**
Validation Accuracy	48.50%	85.28%
Fitness NMSE (Equation 5.4)	−0.3297	−0.0903

Table 5.2: Federated Neuroevolution Performance on EEG Dataset

The stochastic nature of the learning comes from the random selection of node subsets. This can be observed more closely in Appendix B.3.

From a fully random state, my algorithm was able to get to 85% validation accuracy as seen in Table 5.2. This is, of course, a lot less than the baseline but still a good result considering I am using Neuroevolution for training weights which is not the best method for training Neural Networks as mentioned in Subsection 4.2.1. With better hyper-parameters the performance could be increased.

Chapter 6

Conclusion

I have introduced a new Federated Learning algorithm named Federated Neuroevolution, as a modified version of the Neuroevolution algorithm adapted for the Federated setting. In the case study, I have proved the algorithm is capable of solving problems in the Federated setting.

Federated Neuroevolution's advantage, compared to the FSVRG algorithm proposed by the original paper[18], is that it is exposing even less client data to the server. FSVRG exposes the client side data distribution and the gradients during learning. Federated Neuroevolution only expose the amount of datapoints of the clients and an abstract fitness number of the model.

The disadvantages are that the convergence is a lot slower. I needed 5000 iterations of the algorithm to get to an 85% accuracy which is still less then the baseline's 95%. However, this could be improved with more thorough hyper-parameter search for the algorithm.

In summary, I am trading off 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 other use cases, like medical institutions, the rounds of communication is not important at all while privacy aspect takes precedence instead.

Further work includes learning the network topology based on already developed algorithms for topology learning with Neuroevolution in the single node setting [32].

Bibliography

Books

[3] Isaac Asimov. *Foundation Series*. 1942–1993.

[37] Andreas Zell. *Simulation Neuronaler Netze*. Addison-Wesley, 1994. ISBN: 3893195548.

Online Sources

[1] Google Photos application. URL: https://en.wikipedia.org/wiki/Google_Photos (Last accessed on 05/03/2019).

[2] Illustration of a Deep Convolutional Neural Network. URL: http://neuralnetworksanddeep.com/chap6.html (Last accessed on 05/07/2019).

[4] Henri Begleiter. EEG Database Data Set. URL: https://archive.ics.uci.edu/ml/datasets/eeg+database (Last accessed on 05/13/2019).

[6] Josh Constine. Facebook talked privacy, Google actually built it. URL: https://techcrunch.com/2019/05/07/show-dont-tell/ (Last accessed on 05/11/2019).

[7] Arden Dertat. Applied Deep Learning - Part 4: Convolutional Neural Networks. URL: https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2 (Last accessed on 05/08/2019).

[9] Ahmed Gad. Artificial Neural Networks Optimization using Genetic Algorithm with Python. 2019. URL: https://towardsdatascience.com/artificial-neural-networks-optimization-using-genetic-algorithm-with-python-1fe8ed17733e (Last accessed on 05/15/2019).

[11] Raúl Gómez. Understanding Categorical Cross-Entropy Loss, Binary Cross-Entropy Loss, Softmax Loss, Logistic Loss, Focal Loss and all those confusing names. 2018. URL: https://gombru.github.io/2018/05/23/cross_entropy_loss/ (Last accessed on 05/14/2019).

[16] Prateek Karkare. Convolutional Neural Networks - Simplified. URL: https://medium.com/x8-the-ai-community/cnn-9c5e63703c3f (Last accessed on 05/08/2019).

[17] Ruslan Klymentiev. EEG Data Analysis. 2019. URL: https://www.kaggle.com/ruslankl/eeg-data-analysis (Last accessed on 05/13/2019).

[27] Sethbin. MarI/O — A Neural Network capable of beating the classical game Mario. URL: https://youtu.be/qv6UVOQ0F44 (Last accessed on 05/13/2019).

[33] *The AI Revolution: The Road to Superintelligence*. Jan. 22, 2015. URL: https://waitbutwhy.com/2015/01/artificial-intelligence-revolution-1.html (Last accessed on 05/03/2019).

[34] *THE MNIST DATABASE of handwritten digits*. URL: http://yann.lecun.com/exdb/mnist/ (Last accessed on 05/05/2019).

Scientific Articles

[5] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. "Empirical evaluation of gated recurrent neural networks on sequence modeling". In: *arXiv preprint arXiv:1412.3555* (2014).

[18] Jakub Konečnỳ, H Brendan McMahan, Daniel Ramage, and Peter Richtárik. "Federated optimization: Distributed machine learning for on-device intelligence". In: *arXiv preprint arXiv:1610.02527* (2016).

[19] Jakub Konečnỳ and Peter Richtárik. "Semi-stochastic gradient descent methods". In: *arXiv preprint arXiv:1312.1666* (2013).

[20] WB Langdon. "Minimising testing in genetic programming". In: *RN* 11.10 (2011), p. 1.

[21] Chenxin Ma, Jakub Konečnỳ, Martin Jaggi, Virginia Smith, Michael I Jordan, Peter Richtárik, and Martin Takáč. "Distributed optimization with arbitrary local solvers". In: *Optimization Methods and Software* 32.4 (2017), pp. 813–848.

[24] Sebastian Ruder. "An overview of gradient descent optimization algorithms". In: *arXiv preprint arXiv:1609.04747* (2016).

[25] Robin Tibor Schirrmeister, Jost Tobias Springenberg, Lukas Dominique Josef Fiederer, Martin Glasstetter, Katharina Eggensperger, Michael Tangermann, Frank Hutter, Wolfram Burgard, and Tonio Ball. "Deep learning with convolutional neural networks for EEG decoding and visualization". In: *Human brain mapping* 38.11 (2017), pp. 5391–5420.

[26] Jürgen Schmidhuber. "Deep learning in neural networks: An overview". In: *Neural networks* 61 (2015), pp. 85–117.

[28] Virginia Smith, Simone Forte, Ma Chenxin, Martin Takáč, Michael I Jordan, and Martin Jaggi. "Cocoa: A general framework for communication-efficient distributed optimization". In: *Journal of Machine Learning Research* 18 (2018), p. 230.

[29] Joan G Snodgrass and Mary Vanderwart. "A standardized set of 260 pictures: norms for name agreement, image agreement, familiarity, and visual complexity." In: *Journal of experimental psychology: Human learning and memory* 6.2 (1980), p. 174.

[32] Kenneth O Stanley and Risto Miikkulainen. "Evolving neural networks through augmenting topologies". In: *Evolutionary computation* 10.2 (2002), pp. 99–127.

[35] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. "Spark: Cluster computing with working sets." In: *HotCloud* 10.10-10 (2010), p. 95.

[36] Matthew D Zeiler. "ADADELTA: an adaptive learning rate method". In: *arXiv preprint arXiv:1212.5701* (2012).

Glossary

Artificial life Artificial life is a field of study wherein researchers examine systems related to natural life, its processes, and its evolution, through the use of simulations with computer models, robotics, and biochemistry.
See https://en.wikipedia.org/wiki/Artificial_life. 22

Control Flow In computer science, control flow (or flow of control) is the order in which individual statements, instructions or function calls of an imperative program are executed or evaluated.
See https://en.wikipedia.org/wiki/Control_flow. 3

Convolutional Neural Network (CNN) In deep learning, a convolutional neural network (CNN, or ConvNet) is a class of deep neural networks, most commonly applied to analyzing visual imagery.
See https://en.wikipedia.org/wiki/Convolutional_neural_network. 12

Darwinian Evolution Darwinism is a theory of biological evolution developed by the English naturalist Charles Darwin (1809–1882) and others, stating that all species of organisms arise and develop through the natural selection of small, inherited variations that increase the individual's ability to compete, survive, and reproduce.
See https://en.wikipedia.org/wiki/Darwinism. 21

Directed Acyclic Graph (DAG) In graph theory a directed acyclic graph (DAG), is a finite directed graph with no directed cycles. That is, it consists of finitely

many vertices and edges, with each edge directed from one vertex to another, such that there is no way to start at any vertex v and follow a consistently-directed sequence of edges that eventually loops back to v again.
See https://en.wikipedia.org/wiki/Directed_acyclic_graph. 6

Directed Graph In graph theory, a directed graph (or digraph) is a graph that is made up of a set of vertices connected by edges, where the edges have a direction associated with them.
See https://en.wikipedia.org/wiki/Directed_graph. 4

Distributed Optimization Distributed optimization is a method where we try to optimize a function f without having the data on a single central location. 13

Electroencephalography (EEG) Electroencephalography (EEG) is an electrophysiological monitoring method to record electrical activity of the brain. It is typically noninvasive, with the electrodes placed along the scalp, although invasive electrodes are sometimes used, as in electrocorticography. EEG measures voltage fluctuations resulting from ionic current within the neurons of the brain.
See https://en.wikipedia.org/wiki/Electroencephalography. 26

Evolutionary Algorithm Evolutionary Algorithm is a subset of evolutionary computation, a generic population-based metaheuristic optimization algorithm. An Evolutionary Algorithm uses mechanisms inspired by biological evolution, such as reproduction, mutation, recombination, and selection.
See https://en.wikipedia.org/wiki/Evolutionary_algorithm. 2

Evolutionary Robotics Evolutionary Robotics (ER) is a methodology that uses evolutionary computation to develop controllers and/or hardware for autonomous robots.
See https://en.wikipedia.org/wiki/Evolutionary_robotics. 22

Fitness Function A fitness function is a particular type of objective function that is used to summarise, as a single figure of merit, how close a given design

solution is to achieving the set aims.
See https://en.wikipedia.org/wiki/Fitness_function. 2

Fraud Detection Fraud Detection is a financial background process during malicious or fraudalent activities are detected in a financial system. This process can be automated with Artificial Intelligence.
See https://en.wikipedia.org/wiki/Fraud#Detection. 5

General game playing General game playing (GGP) is the design of artificial intelligence programs to be able to play more than one game successfully.
See https://en.wikipedia.org/wiki/General_game_playing. 22

Gradient Gradients are calculated during backpropagation to update the weights of an ANN.
See https://en.wikipedia.org/wiki/Backpropagation. 2

Hyperparameter In machine learning, a hyperparameter is a parameter whose value is set before the learning process begins. By contrast, the values of other parameters are derived via training. Examples are stepsize, learning rate.
See https://en.wikipedia.org/wiki/Hyperparameter_(machine_learning). 23

Kodkod Kodkods are the smallest wild cats living in South America.
https://en.wikipedia.org/wiki/Kodkod. 1

MRI Magnetic resonance imaging (MRI) is a medical imaging technique used in radiology to form pictures of the anatomy and the physiological processes of the body in both health and disease.
See https://en.wikipedia.org/wiki/Magnetic_resonance_imaging. 3

Natural Language Processing Natural Language Processing is a subfield of Artificial Intelligence which is concerned with analyzing and understanding naturally spoken or written human languages.
See https://en.wikipedia.org/wiki/Natural_language_processing. 5

Neuroevolution Neuroevolution is a form of artificial intelligence that uses evolutionary algorithms to generate artificial neural networks, parameters, topology and rules.
See https://en.wikipedia.org/wiki/Neuroevolution. 2

Positronic Brain The artificially created brains of the robots in Isaac Asimov's Robot universe. 1

Privacy Privacy is the ability of an individual or group to seclude themselves, or information about themselves, and thereby express themselves selectively. https://en.wikipedia.org/wiki/Privacy. 2

Privacy Aware A person is called Privacy Aware if he/she is cautious about how his/her data is being handled by services and third parties.. 12

Pudú Pudús are the smallest deers and can be found in South America. https://en.wikipedia.org/wiki/Pud%C3%BA. 1

Visual Cortex The visual cortex of the brain is that part of the cerebral cortex which processes visual information. It is located in the occipital lobe. Visual nerves run straight from the eye to the primary visual cortex to the Visual Association cortex.
See https://en.wikipedia.org/wiki/Visual_cortex. 7

Acronyms

AI Artificial Intelligence. 1, 2

ANN Artifical Neural Network. 2, 4, 21, 23, 34, 48

ANNs Artifical Neural Networks. 2–6, 20, 22, 23, 66, 67

CNN Convolutional Neural Network. 8, 30, 32

CNNs Convolutional Neural Networks. 7, 8, 30

DNN Deep Neural Network. 5

EA Evolutionary Algorithm. 20–24, 34, 37

EAs Evolutionary Algorithms. 20, 21, 33, 37

FSVRG Federated Stochastic Variance Reduced Gradient. 14, 20

GA Genetic Algorithm. 34

ML Machine Learning. 1, 13, 29, 30

NE Neuroevolution. 22–24, 33

RNN Recurrent Neural Network. 6, 7

RNNs Recurrent Neural Networks. 6, 7

SGD Stohastic Gradient Descent. 9, 10, 14

SVRG Stochastic Variance Reduced Gradient. 13, 14

TWEANNs Topology and Weight Evolving Artificial Neural Network algorithms. 23

Appendix A

Source Codes, Text files, Listings

```
1  import numpy as np
2
3
4  def individual_fitness_nmse(keras_model, X, y):
5      y_pred = keras_model.predict(X, batch_size=512)
6
7      try:
8          return -1.0 * mean_squared_error(y, y_pred)
9      except:
10         return -100000
11
12 def fitness_of_model_for_nodes(nodes, model_weights,
   ↪ individual_fitness):
13     keras_model = create_keras_model(model_weights)
14     weights_and_scores = np.array([node.evaluate_model(keras_model,
       ↪ individual_fitness) for node in nodes]).transpose()
15
16     return np.average(weights_and_scores[1],
       ↪ weights=weights_and_scores[0], axis = 0)
```

```
17
18
19  # Iteration is based on models because of massive overhead of keras
    ↪ model creation.
20  def federated_population_fitness_model_based(nodes,
    ↪ individual_fitness, population_of_models):
21      fitness_scores = [fitness_of_model_for_nodes(nodes, model,
        ↪ individual_fitness) for model in population_of_models]
22
23      return fitness_scores
```

Listing 7: Federated fitness calculation in Federated Neuroevolution

```
1  import numpy as np
2
3  class Node:
4
5      _id = 1
6
7      def __init__(self, id, X, y):
8          if id == None:
9              id = Node._id
10             Node._id += 1
11         self.id = id
12         self.X = X
13         self.y = y
14
15     def evaluate_model(self, model, individual_fitness):
16         return len(self.y), individual_fitness(model, self.X,
           ↪  self.y)
17
18
19     def evaluate_multiple_models(self, models, individual_fitness):
20         return len(self.y), [individual_fitness(model, self.X,
           ↪  self.y) for model in models]
21
22     def __eq__(self, other):
23             if other == None:
24                 return False
25
26             myId = self.id
```

```
27              otherId = other.id
28              if myId == otherId:
29                  return True
30              else:
31                  return False
32
33      def __str__(self):
34          # Non alcoholic: [1, 0], Alcoholic: [0, 1]
35          argmaxed = np.argmax(self.y, axis = 1)
36          alcoholic_count = np.count_nonzero(argmaxed == 1)
37          non_alcoholic_count = np.count_nonzero(argmaxed == 0)
38
39          return "Node(id = |%d|, Sample count = |%d|, Alcoholic
            ↪  samples = |%d|, Non-Alcoholic samples = |%d|)" %\
40              (self.id, len(self.y), alcoholic_count,
                ↪  non_alcoholic_count)
41
42      __repr__ = __str__
```

Listing 8: The Node abstraction class

```
from abc import ABC, abstractmethod
import numpy as np

class NodeIteratorBase(ABC):

    def __init__(self, X, y, change_interval):
        self.change_interval = change_interval
        self.nodes = NodeIteratorBase.split_nodes(X, y)
        self._access_nr = 0
        self.current_subset = []

    @staticmethod
    def split_nodes(X, y):
        """
            We will split the data into 1 order of magnitude less nodes than the actual length of data.
            This ensures the 'Massively Distributed' federated property.
        """
        node_count = int(len(y) / 10)
        log.info('Splitting |%d| data into |%d| nodes', len(y), node_count)
        split_indices = np.append([0, len(X)], np.random.choice(range(1, len(X)), node_count - 1, replace=False))
        split_indices.sort()
        nodes = [Node(None, X[start:end], y[start:end]) for start, end in zip(split_indices[:-1], split_indices[1:])]
```

```
23          for node in nodes:
24              log.info(node)
25
26          return nodes
27
28      @abstractmethod
29      def update(self):
30          pass
31
32      def __iter__(self):
33          while(True):
34              yield self.__next__()
35
36      def __next__(self):
37          if self._access_nr % self.change_interval == 0:
38              self.update()
39
40          self._access_nr += 1
41          return self.current_subset
42
43  class NodeIteratorRandomSubset(NodeIteratorBase):
44
45      def __init__(self, X, y, change_interval, subset_ratio):
46          super().__init__(X, y, change_interval)
47          self.subset_ratio = subset_ratio
48
49      def update(self):
50          # Choosing a random set of nodes.
```

```
51          self.current_subset = np.random.choice(self.nodes,
            ↪  int(len(self.nodes) * self.subset_ratio))
```

Listing 9: Node iterator base class and Random Subset variant used in Federated Neuroevolution

```
1  import numpy as np
2  from keras import backend as K
3
4  def run_federated_evolution(*, nodes_iterator, X_validate,
   ↪  y_validate,\
5                      num_parents_mating, num_generations,
                       ↪  mutation_chance,mutation_rate,\
6                      best_fitness_of_each_generation,
                       ↪  best_accuracy_of_each_generation,
                       ↪  best_model_of_each_generation,
                       ↪  population_weights,\
7                      stuck_multiplier, stuck_multiplier_max,
                       ↪  stuck_evasion_rate, stuck_check_length):
8      y_validate_argmax = np.argmax(y_validate, axis = 1)
9
10     for generation in range(num_generations):
11         # Measuring the fitness of each individual in the
           ↪  population.
12         fitness_scores =
           ↪  federated_population_fitness_model_based(next(nodes_iterator),
           ↪  individual_fitness_nmse, population_weights)
13
14         best_fitness_of_each_generation.append(max(fitness_scores))
15         best_model_keras =
           ↪  create_keras_model(population_weights[np.argmax(fitness_scores)])
16
           ↪  best_accuracy_of_each_generation.append(individual_accuracy(best_mod
           ↪  X_validate, y_validate_argmax))
```

```
17
        ↪ best_model_of_each_generation.append(population_weights[np.argmax(fi
18      # Selecting the best parents in the population for mating.
19      parents =
        ↪ fittest_parents_of_generation(population_weights.copy(),
        ↪ fitness_scores, num_parents_mating)
20      # Generating next generation using crossover.
21      offsprings = crossover(parents.copy(),
        ↪ len(population_weights) - num_parents_mating)
22      stuck_multiplier_value = min(stuck_multiplier,
        ↪ stuck_multiplier_max)
23      # Adding some variations to the offsrping using mutation.
24      offsprings = mutation(offsprings,
        ↪ mutation_chance=mutation_chance *
        ↪ np.sqrt(stuck_multiplier_value),
        ↪ mutation_rate=mutation_rate * stuck_multiplier_value)
25      # Creating the new generation based on the parents and
        ↪ offspring.
26      population_weights = []
27      population_weights.extend(parents)
28      population_weights.extend(offsprings)
29      # If our accuracy is not increasing we try and speed up
        ↪ mutation
30      if generation > 0 and
        ↪ best_accuracy_of_each_generation[generation] in
        ↪ best_accuracy_of_each_generation[generation-stuck_check_length:gener
31          stuck_multiplier *= stuck_evasion_rate
32      else:
33          stuck_multiplier = 1
```

```
34          #cleanup resources
35          K.clear_session()
36
37      return best_fitness_of_each_generation,
        ↪ best_accuracy_of_each_generation,
        ↪ best_model_of_each_generation, population_weights
```

Listing 10: Federated Neuroevolution Algorithm

Listing 1 EEG Trial raw data

```
# co2a0000364.rd

# 120 trials, 64 chans, 416 samples 368 post_stim samples

# 3.906000 msecs uV

# S1 obj , trial 0

# FP1 chan 0

0 FP1 0 -8.921

0 FP1 1 -8.433

...

0 FP1 253 4.262

0 FP1 254 5.727

0 FP1 255 8.169

# FP2 chan 1

0 FP2 0 0.834

0 FP2 1 3.276

0 FP2 2 5.717

...

0 Y 250 3.153

0 Y 251 6.571

0 Y 252 12.431

0 Y 253 15.849

0 Y 254 16.337

0 Y 255 14.872
```

Listing 2 Baseline configuration

```
input_shape = (64, 256, 1)
num_classes = 2
batch_size=64
epochs=100

model = Sequential()
model.add(Conv2D(30, kernel_size=(1, 25),
                 input_shape=input_shape))
model.add(Conv2D(10, kernel_size=(64, 1)))
model.add(Lambda(lambda x: x ** 2))
model.add(AveragePooling2D(pool_size=(1, 15), strides=(1, 1)))
model.add(Lambda(lambda x: safe_log(x)))
model.add(Conv2D(2, kernel_size=(1, 8), dilation_rate=(15, 1)))
model.add(BatchNormalization(momentum=0.1))
model.add(Flatten())
model.add(Dense(num_classes, activation='sigmoid'))

model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),
              metrics=['accuracy'])
```

Listing 3 Baseline summary

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 64, 232, 30)       780
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 1, 232, 10)        19210
_________________________________________________________________
lambda_1 (Lambda)            (None, 1, 232, 10)        0
_________________________________________________________________
average_pooling2d_1 (Average (None, 1, 218, 10)        0
_________________________________________________________________
lambda_2 (Lambda)            (None, 1, 218, 10)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 1, 211, 2)         162
_________________________________________________________________
batch_normalization_1 (Batch (None, 1, 211, 2)         8
_________________________________________________________________
flatten_1 (Flatten)          (None, 422)               0
_________________________________________________________________
dense_1 (Dense)              (None, 2)                 846
=================================================================
Total params: 21,006
Trainable params: 21,002
Non-trainable params: 4
_________________________________________________________________
```

Listing 4 Final hyperparameters of the Federated Neuroevolution Algorithm

```
node_activation_ratio = 0.10

node_subset_change_interval = 10

population_size = 50

num_parents_mating = 8

num_generations = 5000

mutation_chance = 0.01

mutation_rate = 3

stuck_multiplier = 1

stuck_evasion_rate = 1.25

stuck_multiplier_max = 5

stuck_check_length = 30
```

Listing 5 Kernelwise mixing of 2 ANNs' weights

```
import numpy as np

# Mixing too models by keeping their kernel weights intact
def kernelwise_mix(model_a, model_b):
    mix = []
    for i in range(len(model_a)):
        layer_a = model_a[i]
        layer_b = model_b[i]
        # choosing kernels
        choice = np.random.randint(2, size = int(layer_a.size /
        ↪  layer_a.shape[-1])).reshape(layer_a.shape[:-1]).astype(bool)
        # extending the chosen kernel bools to the level of single
        ↪  values
        choice = np.repeat(choice,
        ↪  layer_a.shape[-1]).reshape(layer_a.shape)

        layer_mix = np.where(choice, layer_a, layer_b)
        mix.append(layer_mix)

    return mix
```

Listing 6 Mutating the weights of an ANNs

```
import numpy as np

def mutation(offsprings, mutation_chance=0.1, mutation_rate=1):

    for offspring in offsprings:
        for layer in offspring:
            trues = np.full(int(layer.size * mutation_chance), True)
            falses = np.full(layer.size - trues.size, False)
            mutation_indices = np.append(trues, falses)
            np.random.shuffle(mutation_indices)
            mutation_indices = mutation_indices.reshape(layer.shape)

            # The random value to be added to the gene.
            mutation_multiplier = np.random.normal(loc=0.0,
             ↪  scale=0.01 * mutation_rate, size=1)
            layer[mutation_indices] = layer[mutation_indices] +
             ↪  layer[mutation_indices] * mutation_multiplier

    return offsprings
```

Appendix B

Visualizations

Figure B.1: Data distribution among nodes. Total node count is 721, but only first 100 node is displayed here.

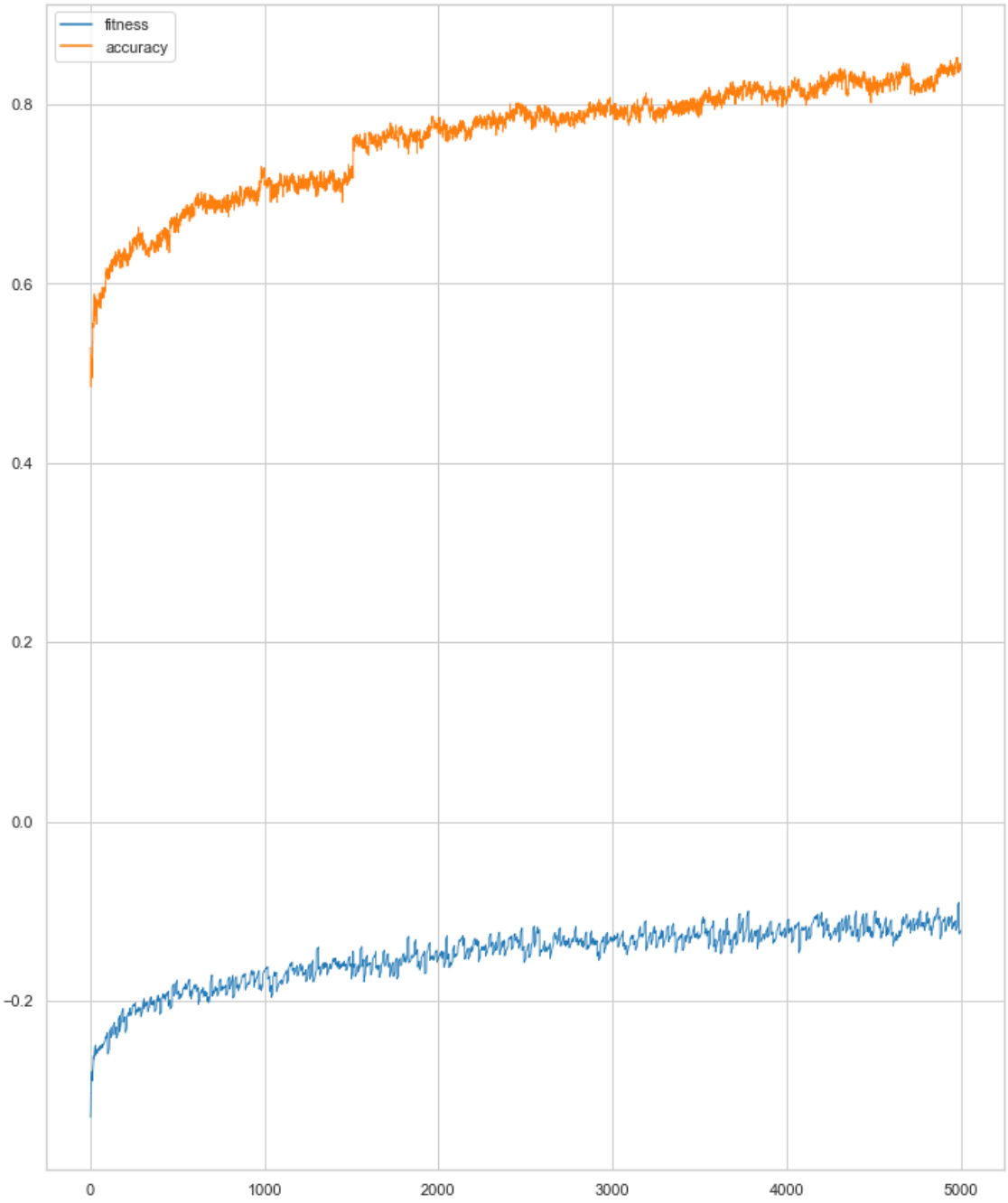


Figure B.2: Running Federated Neuroevolution on the EEG dataset for 5000 generations. Fitness is NMSE 5.4, Accuracy is validation accuracy.

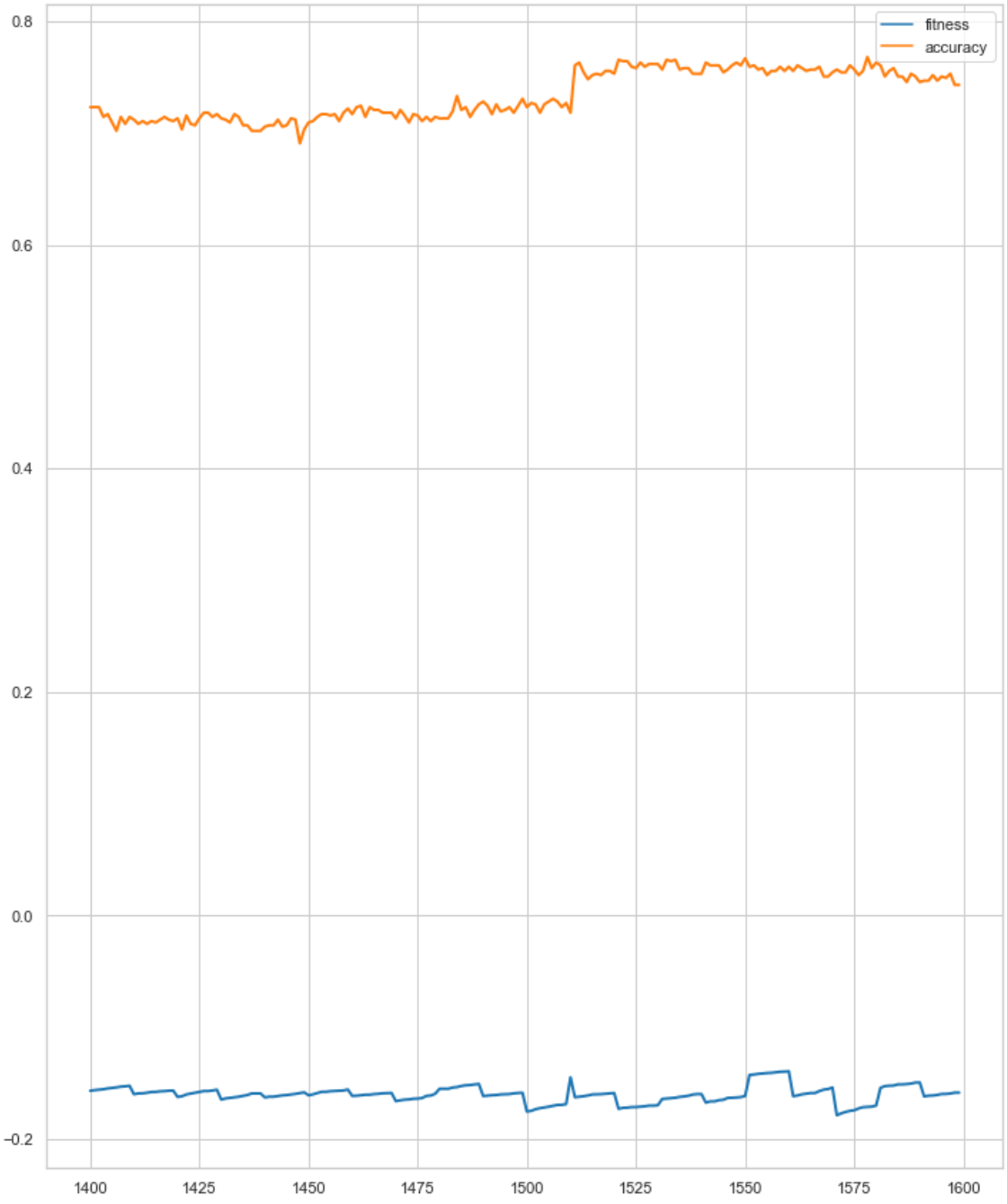


Figure B.3: Zooming in on the training for a 200 generation span.