Functional Federated Learning
in Erlang (`ffl-erl`)⋆

Gregor Ulm(✉)[1,2][0000−0001−7848−4883], Emil Gustavsson[1,2][0000−0002−1290−9989],
and Mats Jirstrand[1,2][0000−0002−6612−8037]

[1] Fraunhofer-Chalmers Research Centre for Industrial Mathematics,
Chalmers Science Park, 412 88 Gothenburg, Sweden
[2] Fraunhofer Center for Machine Learning,
Chalmers Science Park, 412 88 Gothenburg, Sweden
{gregor.ulm,emil.gustavsson, mats.jirstrand}@fcc.chalmers.se
http://www.fcc.chalmers.se/

Abstract. The functional programming language Erlang is well-suited for concurrent and distributed applications, but numerical computing is not seen as one of its strengths. Yet, the recent introduction of Federated Learning, which leverages client devices for decentralized machine learning tasks, while a central server updates and distributes a global model, motivated us to explore how well Erlang is suited to that problem. We present the Federated Learning framework `ffl-erl` and evaluate it in two scenarios: one in which the entire system has been written in Erlang, and another in which Erlang is relegated to coordinating client processes that rely on performing numerical computations in the programming language C. There is a concurrent as well as a distributed implementation of each case. We show that Erlang incurs a performance penalty, but for certain use cases this may not be detrimental, considering the trade-off between speed of development (Erlang) versus performance (C). Thus, Erlang may be a viable alternative to C for some practical machine learning tasks.

Keywords: Machine learning · Federated learning · Distributed computing · Functional programming · Erlang

1 Introduction

With the explosion of the amount of data gathered by networked devices, more efficient approaches to distributed data processing are needed. The reason is that it would be infeasible to transfer all data gathered from edge devices to a central data center, process it, and afterwards transfer results back to edge devices via the network. There are several approaches to taming the amount of data received, such as filtering on edge devices, transferring only a representative sample, or performing data processing tasks decentrally. A recently introduced

---

⋆ The final authenticated version is available online at https://doi.org/10.1007/978-3-030-16202-3_10.

example of distributed data analytics is Federated Learning [14]. Its key idea is the distribution of machine learning tasks to a subset of available devices, followed by performing machine learning tasks locally on data that is available on edge devices, and iteratively updating a global model.

In this paper, we present `ffl-erl`, a Federated Learning framework implemented in the functional programming language Erlang.[3] This work was produced in the context of an industrial research project with the goal of exploring and evaluating various approaches to distributed data analytics in the automotive domain. Our contribution consists of the following:

- Creating `ffl-erl`, the first open-source implementation of a framework for Federated Learning in Erlang
- Highlighting the feasibility of functional programming for the aforementioned framework
- Creating a purely functional implementation of an artificial neural network in Erlang
- Comparing the performance of a Federated Learning implementation fully in Erlang with one in which client processes are implemented in C
- Exploring two approaches of integrating C with Erlang: NIFs and C nodes

The remainder of our paper is organized as follows: Section 2 contains background information and describes the motivating use case. Section 3 covers our implementation in detail and presents experimental results. Section 4 gives a brief overview of related work, while Section 5 describes future work. Appendix A contains a mathematical derivation of Federated Stochastic Gradient Descent.

2   Background

This section gives an overview of Federated Learning (2.1) and presents the mathematical foundation of Federated Stochastic Gradient Descent (2.2). It furthermore describes our motivating use case (2.3).

2.1   Federated Learning

Federated Learning is a decentralized approach to machine learning. The general idea is to perform machine learning tasks on a potentially very large number of edge devices, which process data that is only accessible locally. A central server is relegated to assigning tasks and updating the global model based on the local models it receives from edge devices. One iteration of Federated Learning consists of the following steps, following McMahan et al. [14]:

1. Select a subset c of the set of clients C
2. Send the current model from the server to each client $x \in c$

---

[3] Source code artifacts accompanying this paper are available at https://gitlab.com/fraunhofer_chalmers_centre/functional_federated_learning.

3. For each x, update the provided model based on local data by performing iterations of a machine learning algorithm
4. For each x, send the updated model to the server
5. Aggregate all received local models and construct a new global model

There are several motivations behind Federated Learning. First, there is the bandwidth problem in a big data setting. The amount of data generated by local devices is too large to be transferred via the network to a central server for processing. Second, edge devices are getting more and more powerful. Modern smartphones, for instance, have been compared to (old-generation) supercomputers in our pockets in terms of raw computational power [2]. Therefore, it seems prudent to more efficiently use these resources. Third, there are data privacy issues, as some jurisdictions have strict privacy laws. Thus, transmitting data via the network in order to perform central machine learning tasks is frayed with data privacy issues. This is summarized by Chen et al. [3], while Tene et al. point out legal issues [20]. Federated Learning sidesteps potential legal quagmires surrounding data privacy laws and regulations as data is not centrally collected.

2.2   Federated Stochastic Gradient Descent

Federated Stochastic Gradient Descent (Federated SGD) is based on Stochastic Gradient Descent (SGD), which is a well-established method in the field of statistical optimization. We first describe SGD, followed by a presentation of Federated SGD. The latter is based on McMahan et al. [14].

Stochastic Gradient Descent. The aim of SGD is to minimize an objective function F that is defined as the following sum:

$$F(w) = \frac{1}{n} \sum_{i=1}^{n} F_i(w). \tag{1}$$

The goal is to find a value for the parameter vector w that minimizes F. The value F_i represents the contribution of element i of the input data to the objective function. In order to minimize F, the gradient ∇ is computed. The learning rate η is a factor that adjusts how far along the gradient the parameter update step is taken. It modifies the magnitude of change of w between iterations. The parameter w is updated in the following way:

$$w := w - \eta \nabla F(w). \tag{2}$$

This means that the parameter w is updated by computing the gradient of the objective function, evaluated for the previous parameter value, which is subtracted from the previous parameter value w. Since F is a separable function, Eq. 2 can be reformulated as

$$w := w - \frac{\eta}{n} \sum_{i=1}^{n} \nabla F_i(w). \tag{3}$$

As indicated before, the learning rate η is a modifier for slowing down or speeding up the training process. In practice, small positive values in the half-closed interval $(0, 1]$ are used. A common starting value is 0.01. A learning rate that is too high may overshoot a global optimum. A learning rate that is too low, on the other hand, may severely impact the performance of the algorithm.

Federated Stochastic Gradient Descent. Federated SGD is an extension of SGD. It takes into account that there are k partitions P_j of the training data, with j ranging from 1 to k, i.e. there is a bijection between partitions and clients. Consequently, Eq. 3 has to be modified as we need to consider the work performed on each client $\in c$, where c is the chosen subset of all clients C. The objective function is attempted to be minimized for each of the k clients. However, the goal is to optimize the global model, not any of the local models. For Eq. 4, keep in mind that there are n elements in the input data, thus $n = \sum_j |P_j|$.

$$F^j(w) = \frac{1}{|P_j|} \sum_{i \in P_j} F_i(w) \; for \; j = 1, \ldots, k \tag{4}$$

The global objective function is shown in Eq. 5. Its full derivation is provided in Appendix A.

$$F(w) = \frac{1}{n} \sum_{j=1}^{k} |P_j| F^j(w) \tag{5}$$

2.3   Motivating Use Case

Intelligent vehicles generate vast amounts of data. According to recent industry figures, they can generate dozens of gigabytes of data per hour [4]. Considering even a moderately sized fleet of just a few hundred cars, collecting data, transferring data to a central server, processing data on a central server and afterwards sending results to each car is infeasible as we are already in the region of terabytes of data per hour. Yet, even simple tasks like filtering on the client can provide valuable insights. This is an example of a relatively straightforward way of reducing input data to a small fraction of its original volume, which highlights the importance of decentralized data processing.

However, our focus is on a more complex use case in the context of distributed data analytics. We explore training a machine learning model on client devices with local data, while a central server performs supplementary tasks. This relates to a real-world setting in which connected cars [6] are equipped with on-board units that continuously gather data. These on-board units are general-purpose computers with performance metrics comparable to smartphones. For instance, our hardware uses an ARM-based multi-core CPU, similar to those found in a typical mid-range smartphone. On-board units are connected via wireless or 4G broadband networking to a central server, possibly via intermediaries, so-called road-side units. This is by no means a merely theoretical scenario. For instance, a recent large-scale experiment with road-side units was carried out by Lee and Kim [12] in South Korea in 2010.

3   Solution

Our research prototype simulates a distributed system in which a central server interacts with a large number of clients. We first describe the main components of the framework itself (3.1). This is followed by a discussion of a purely functional implementation of an artificial neural network (ANN) in Erlang (3.2). Subsequently, we describe how the skeleton and the ANN can be combined (3.3). Finally, we discuss experimental results (3.4).

3.1   The Skeleton of the Framework

This section illustrates the main ideas behind implementing a distributed machine learning framework. Consequently, we present the main parts of our skeleton, i.e. the client and server processes. The source code in this section leaves some details unspecified, but these can be filled in easily or referenced in the accompanying code repository. It seems appropriate to preface the discussion of our source code by briefly explaining the communication model of Erlang. In Erlang, processes communicate asynchronously by sending messages to each other. Each process has its own mailbox for incoming messages, which are processed in the order they arrive. However, the order in which they arrive is non-deterministic. If process C receives one message each from processes A and B, in this order, there is no guarantee that they were also sent in this order.

The skeleton consists of a client process, which may be instantiated an arbitrary number of times, and a server process. Both are shown in Code Listing 1.1. In the client process, the `receive` clause awaits a tuple tagged with the atom `assignment`. The received model is trained with local data via the function `train`. Examples of such a model are the weights of an ANN or parameters of a linear regression equation. After training has concluded, the updated model is sent to the server process. A tuple that is tagged with the atom `update` is sent to the server, using the operator '!', which is pronounced as *send*. The server is addressed via the process identifier `Server_Pid`. Thus, line 5 has to be read from right to left to trace the execution, i.e. we take a tuple tagged as `update`, containing the process identifier of the current process that is returned when calling the function `self` as well as the new local model, and send it to the server identified by `Server_Pid`. Afterwards, the client function is called recursively, awaiting an updated model.

The server process shown in Code Listing 1.1 does not perform computationally intensive tasks. Instead, its role is to maintain a global model, based on updates received from client processes. Our system selects a random subset of all available devices. Sending the current model to the selected subset of client processes can be concisely expressed via mapping over a list or a list comprehension. It is assumed that all devices complete their assignments. This is reflected in the list comprehension in line 14, which blocks until the results of all assignments have been received. The resulting list of values `Vals` contains the updated local models of all client processes, with which a new global model will be constructed. The corresponding function `update_model` is unspecified, however. After updating

```
1  client() ->
2    receive
3    { assignment, Model, Server_Pid } ->
4      Val = train(Model), % computes 'w_j'
5      Server_Pid ! { update, self(), Val },
6      client()
7    end.
8
9  server(Client_Pids, Model) ->
10   Subset = select_subset(Client_Pids),
11   % Send assignment:
12   [ X ! { assignment, Model, self() } || X <- Subset ],
13   % Receive values:
14   Vals = [ receive { update, Pid, Val } -> Val end || Pid <- Subset ],
15   % Update model, i.e. compute global 'w':
16   Model_ = update_model(Model, Vals, length(Client_Pids)),
17   % Note: it is a simplification to use the number of clients; in this
18   % case, each client has the same number of data points to work with
19   server(Client_Pids, Model_).
```

Listing 1.1: Client and Server processes

the model, the server process calls itself recursively. Overall, the preceding code is a textbook case of message passing in Erlang.

3.2   A Neural Network with Backpropagation in Erlang

Artificial Neural Networks. Artificial Neural Networks (ANNs) are a standard method in machine learning for a variety of learning tasks. A prime example is classification based on pattern recognition, for instance tagging images with keywords. The general principle is to minimize an objective function that computes the magnitude of an error. There are normally three steps to deploying an ANN: training, validation, and use in production. First, a labeled data set is used to train an ANN, which has the goal of minimizing the objective function. There is the risk that the ANN has been over-trained, i.e. it has memorized its input. Therefore, a labeled validation set is used to ensure that a data set that is similar to the test set is also correctly classified. If those two steps have been performed satisfactorily, the ANN is ready to be used for real-world data classification tasks.

Figure 1 shows a typical ANN. It consists of two input neurons, three hidden neurons, and two output neurons. The two input neurons on the left are shaded in order to indicate that an ANN is normally not applied to fixed input values but instead applied sequentially to each element of a larger data set. The layer of neurons in the middle is the hidden layer; the layer on the right is the output layer. The edges labeled with their weights represent connections between neurons. The edges leaving the output layer transmit the final output. There are two sets of labeled edges, one set connecting the input layer to the hidden layer and the other connecting the hidden layer to the output layer. Edge weights are initialized to a small random value and updated via training. The goal is to minimize the output error, which is based on the difference between the target values and the values the output layer neurons emit. After a forward pass we can determine how close the values emitted by the output neurons are to the target values. This is followed by adjusting the weights of the ANN with the backpropagation algorithm. Together, these two steps amount to one *epoch*. In the end, the output error is minimized via iterative adjustments of the weights of the ANN.

Using the ANN in Fig. 1 as an example, we first perform a forward pass, which consists of computing the input of each hidden layer neuron by calculating the dot product of the input weights and all edges connecting the input nodes with that hidden layer neuron. For in-

Fig. 1: Artificial neural network

stance, the input of the topmost hidden layer neuron is $0.25 \times 0.05 + 0.70 \times 0.09 = 0.0755$. After applying the *activation function* to that value, the input and output values of the output layer neurons are computed similarly. The activation function computes the output of a node, taking its input as the argument. Afterwards, the difference between target and actual output values can be calculated. This is followed by a backpropagation pass, in which the weights of the ANN are updated: first the weights of the edges from the output layer to the hidden layer, then the weights of the edges from the hidden layer to the input layer. These calculations are similar to the forward pass, except that the gradient, i.e. the derivative of the objective function we want to minimize, is used when calculating the respective dot products. Training an ANN with a batch of input data is done by processing all elements of the provided data, using them one by one as input for the input layer and performing one epoch. After each iteration, the weights are retained as the goal is to train on the entire set of input data.

For the sake of brevity, our description of an ANN does not consider common modifications such as setting a specific learning rate or using adaptive behavior based on previous results. We furthermore use a standard activation function, the *sigmoid function*. Practitioners may use different activation functions or resort to various engineering techniques for improving the performance of ANNs as described, for instance, by Orr et al. [16]. As a final note, we would like to highlight that ANNs can approximate any function [8, 9], which is commonly referred to as the universal approximation theorem. Consequently, ANNs are widely used in practice. The example described above, consisting of three layers, is a shallow ANN. Those are versatile, but they are not efficient for large and very complex problems. A particularly noteworthy early breakthrough of shallow ANNs was the successful classification of handwritten digits, which is used by postal services [19]. More recent developments include deep neural networks, often referred to as deep learning. Those are ANNs with multiple hidden neuron layers, consisting of large numbers of neurons.
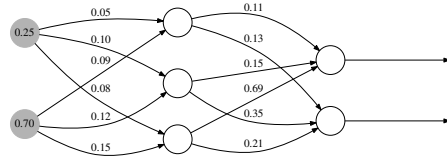
Implementation. In the following, we cover some aspects of an exemplary implementation of a basic ANN in Erlang. We will again leave out some implementation details, and instead focus on the big picture.[4] Code Listing 1.2 shows

---

[4] For illustrative purposes, we chose clear code over computationally more efficient code at some points. For instance, the function `forward` in Code Listing 1.3 constructs a temporary list, which could be avoided by computing the dot product with an accumulator. However, for benchmarking purposes we used more efficient code.

the function `ann`, which models an artificial neural network. The various helper functions it calls are shown in Code Listing 1.3. The input of the function `ann` consists of the values of the input neurons `Input`, the weights of both layers `Weights`, and the target values of the output layer `Targets`.

```erlang
ann(Input, Weights, Targets) ->
  { W_Input, W_Hidden } = Weights,
  % Forward Pass:
  Hidden_In  = forward(Input, W_Input, []),
  Hidden_Out = [ activation_fun(X) || X <- Hidden_In ],
  Output_In  = forward(Hidden_Out, W_Hidden, []),
  Output_Out = [ activation_fun(X) || X <- Output_In ],
  % Target vs. output:
  Delta = lists:zipwith(fun(X, Y) -> X - Y end, Targets, Output_Out),
  % Reverse pass:
  Output_Errors = output_error(Output_Out, Targets),
  % Update weights for output layer:
  W_Hidden_  = backpropagate(Hidden_Out, Output_Errors, W_Hidden, []),
  Hidden_Err = errors_hidden(Hidden_Out, Output_Errors, W_Hidden_, []),
  W_Input_   = backpropagate(Input, Hidden_Err, W_Input, []),
  { Output_Errors, { Input, { W_Input_, W_Hidden_ }, Targets } }.
```

Listing 1.2: The core ANN function

As described earlier, as a first step the ANN computes the input of the hidden layer. The output of the hidden layer is the result of mapping the activation function over the list `Hidden_In`; the corresponding values of the output layer are computed in the exact same way. The list `Delta` contains the differences between the target values and the actual values.[5] The function `forward` computes the dot product of the input values and the weights of the outgoing edges. It is called twice by the function `ann` because there are two transitions between layers, first from the input layer to the hidden layer, and afterwards from the hidden layer to the output layer. Computing the dot product maps nicely to a functional programming style, as the required computation is the element-wise multiplication of two lists, followed by the summation of the results of that computation. The backpropagation pass starts with computing the output error, zipped with a *squashing factor*. In our case, the activation function used for that purpose is a standard sigmoid function, the logistic function $f(x) = \frac{1}{1+e^{-x}}$. The derivative of the logistic function is $f'(x) = f(x)(1 - f(x))$. This makes it possible to efficiently compute gradients as we can use the activations of the hidden layer for computing the total error in the output layer. Computationally, the operations involved, multiplication and subtraction, are less costly than re-evaluating the activation function, which is an exponential function.

The function `backpropagate` performs backpropagation, which computes the adjusted weights of the edges connecting the output layer to the hidden layer, and the adjusted weights of the edges connecting the hidden layer to the input

---

[5] Training normally ends after a given number of iterations or once a predefined error threshold has been met. The latter would make use of the computed error, based on the list `Delta`, but the corresponding code is omitted as it is not conceptually interesting.

```erlang
1  forward(_     , []        , Acc) -> lists:reverse(Acc);
2  forward(Input, [W | Ws], Acc) ->
3    Val = lists:sum(lists:zipwith(fun(X, Y) -> X * Y end, Input, W)),
4    forward(Input, Ws, [Val | Acc]).
5
6  output_error(Vals, Target) ->
7    lists:zipwith(fun(X, Y) -> X * (1.0 - X) * (X - Y) end, Vals, Target).
8
9  backpropagate(_ , []      , []        , Acc) -> lists:reverse(Acc);
10 backpropagate(In, [E|Es], [Ws|Wss], Acc) ->
11   A = lists:zipwith(fun(W, I) -> W - (E * I) end, Ws, In),
12   backpropagate(In, Es, Wss, [A|Acc]).
13
14 errors_hidden([]    , _       , _        , Acc) -> lists:reverse(Acc);
15 errors_hidden([H|Hs], Output_Err, Weights, Acc) ->
16   Outgoing = [ hd(X) || X <- Weights ],
17   % Remaining weights for next iteration:
18   Rest = [ tl(X) || X <- Weights ],
19   % Error of current hidden layer neuron:
20   TMP = lists:zipwith(fun(X, E) -> E * X end, Outgoing, Output_Err),
21   A   = lists:sum(TMP) * H * (1.0 - H),
22   errors_hidden(Hs, Output_Err, Rest, [A|Acc]).
23
24 wrap_ann([]    , Weights, []      , Errors) ->
25   {lists:reverse(Errors), Weights};
26 wrap_ann([I|Is], Weights, [T|Ts], Errors) ->
27   { Error, Weights_ } = ann(I, Weights, T),
28   wrap_ann(Is, Weights_, Ts, [Error | Errors]).
```

Listing 1.3: ANN helper functions

layer. The new weights are computed by adding the product of the error and the input to each weight. The computation of the errors of the hidden layer is slightly trickier, due to using the list data structure. The weights assigned to the edges connecting the hidden layer with the output layer are specified as a list of lists in which each list contains the incoming weights of one of the output neurons. In the backpropagation pass, however, we need to traverse the ANN the opposite way, so the edges connecting the hidden layer to the output layer need a representation that considers all edges that point from the output layer to each node in the hidden layer. This is achieved by recursively taking the heads of the list of lists of the weights before performing the error calculation. Lastly, performing training on the entire input, so-called batch training, can be elegantly expressed in a functional style, shown by the function **wrap_ann**. Its arguments are, in order, the list of inputs that constitute the training set, the weights, the target values associated with the input data, and an accumulator **Errors** that collects the output error for each element of the input set. The weights are continually updated so that every invocation of the function **ann** uses the weights of the preceding invocation.

3.3   The Combined Framework

The parts introduced earlier can be combined to build a distributed system for Federated Learning. It boils down to using the skeleton introduced in Section 3.1 and adding code for an artificial network to the client process, similar to what we have shown in Section 3.2, as well as further program logic. What has not been covered is, for instance, code for input/output handling. Our assumption is that each client process operates on data that is only locally available. The

client process needs to be adjusted correspondingly, so that the available data is processed for batch-training with the ANN. Likewise, the server process needs to process the incoming models from the clients to update the centrally maintained global model, for instance via averaging.

While the description of our implementation is exclusively in Erlang, an alternative approach consists of a C implementation of the ANN. From a user perspective, there is no difference with regards to the output. Of course, internally the client trains an ANN in C instead of Erlang. However, in order to fairly compare how well an implementation solely in Erlang compares against one in which the computationally heavy lifting is performed by C, it is necessary to take the respective idiosyncrasies of two common approaches to interoperability with C into account. The older and more established way of calling C from Erlang is via so-called Natively Implemented Functions (NIFs), which are an improvement over using ports to communicate with C. A more recent addition to Erlang are C nodes, which have the advantage that they can be interfaced with the same way as regular Erlang nodes. Overall, for the purposes of simulating the framework, concurrent execution is adequate. However, distributed execution, in which messages are sent back and forth between nodes, more closely relates to real-world use cases (cf. Section 2.3).

3.4   Evaluation

Setup. We created four versions of our combined Federated Learning framework: (1) a concurrent implementation, fully in Erlang, as well as (2) one in which the clients are implemented in C as NIFs. Furthermore, we implemented (3) a distributed version fully in Erlang as well as (4) a variant of it in which the clients are C nodes. By default, all Erlang nodes in a distributed system are fully connected. As this is neither practical nor desirable for our use case, this behavior was disabled with the flag `-hidden`. Erlang source code has also been compiled to native code, which has been made possible due to the HiPE project [10,17].

The motivation behind benchmarking a distributed system, as opposed to the simpler case of a concurrent system, is that this mirrors the real-world scenario of performing distributed data analytics tasks on a network with many edge devices and a central server. On the other hand, a concurrent system is more straightforward to design and execute. In both the distributed and the concurrent use case, we did not create our own implementation of a neural network in C. Instead, we chose Nissen's widely used Fast Artificial Neural Network (FANN) library [15] with the option `FANN_TRAIN_BATCH`, which uses gradient descent with backpropagation. This corresponds to our Erlang code. Our ANN implementation in Erlang mirrors the chosen architecture of the ANN in FANN, i.e. there are two input nodes, three hidden nodes, and two output nodes. Furthermore, there is one bias node each, connecting to the hidden and the output layer, respectively. Error computation is done via computing the mean squared error (MSE) in both implementations. The Erlang code does not use a learning rate η, which implies that $\eta = 1$. In FANN, η was explicitly set to 1 in order to override the

default value of 0.7. Both implementations use the sigmoid activation function (cf. Section 3.2). The corresponding setting in FANN is `FANN_SIGMOID`.

Hardware and Software. We used a PC with an Intel Core i7-7700K CPU clocked at 4.2 GHz. This is a quad-core CPU that supports hyper-threading with 8 threads. Our code was executed in Ubuntu Linux 16.04 LTS on a VirtualBox 5.1.22 virtual machine hosted by Windows 10 Pro (build 1703). The total amount of RAM available on the host machine was 32 GB, of which 12 GB were dedicated to VirtualBox. We used Erlang/OTP 20.2.2 and, for C, GCC 5.4.0.

Experiment. For benchmarking novel machine learning methods, standard data sets are often used. These include the Iris data set [7], which contains observational data of the petal length of various iris species. A more ambitious data set is the MNIST handwritten digits database [11]. However, our goal was to directly compare the performance of two pairs of systems, so it seemed more appropriate to generate an artificial data set. Our data set is based on the mathematical function $f(x,y) = (\sqrt{xy}, \sqrt[4]{xy})$, where $\{x, y \in \mathbb{R} \mid x, y \in [0, 1)\}$. The ANN consists of two input nodes, three hidden nodes, and two output nodes. The training data consists of tuples (x, y). Each client randomly generated 250 such tuples prior to each round of training. As the relationship between input and output is known, it is trivial to generate an arbitrary amount of data. We performed five 500-second test runs with each of the four combined frameworks, recording time, number of executed iterations of the ANN on each client node, and total error. We used 10 client processes and hard-coded the initial weights for the sake of easy reproducibility. An alternative approach would have been to create the initial weights with the same random seed. In practice, the difference is insignificant.

The number of clients may seem small. However, the client part of our system will eventually be executed on separate hardware, such as the aforementioned on-board units in connected vehicles, which would each represent one single client node. Consequently, the focus is not on how the performance of `ffl-erl` scales when adding increasing numbers of client nodes to one machine. Given the recent interest in deep learning, one may also question the choice of using a shallow ANN. Shallow ANNs are still viable, however. In our case, the specified function is approximated successfully. On a related note, an interesting recent example of using a shallow ANN for computationally challenging work was presented by Cuccu et. al [5]. They show that a shallow ANN can, in some tasks, compete with deep neural networks.

Results. Experimental results are shown in Fig. 2 below. The x-axis indicates the running time in seconds, while the y-axis shows the number of epochs, i.e. the number of iterations of the server-side ANN. Clients perform batch training on 250 data points per epoch. Each training pass consists of a constant amount of work, so the expectation was that the final result would be linear. The plotted data is the average of five test runs, which yielded virtually identical results.

In the concurrent use case, the Erlang-only implementation, compiled to the BEAM virtual machine, executes ∼192,000 epochs in 500 seconds. This value increases to ∼286,000 epochs when compiling Erlang to native code. In comparison, the result with NIFs is ∼386,000 epochs. NIFs cannot be used together with natively compiled Erlang code, which is why a corresponding plot is missing. The performance difference between using NIFs and Erlang code compiled to BEAM amounts to a constant factor of 2.01. With native execution, the speedup compared to execution on the BEAM virtual machine amounts to 49.0%. Comparing that performance to NIFs, the resulting difference shrinks to a factor of 1.35.

An Erlang-only distributed implementation running on the BEAM virtual machine is able to compute ∼128,000 epochs in 500 seconds, which increases to ∼250,000 epochs (+95.3%) with native code. On the other hand, with C nodes, the resulting performance is ∼522,000 epochs on the BEAM virtual machine as opposed to ∼643,000 epochs (+23.4%) per client when compiling Erlang to native code. The performance difference between a pure Erlang implementation and one that uses C nodes amounts to a constant factor of 4.1 on the BEAM virtual machine, which shrinks to 2.6 with HiPE.

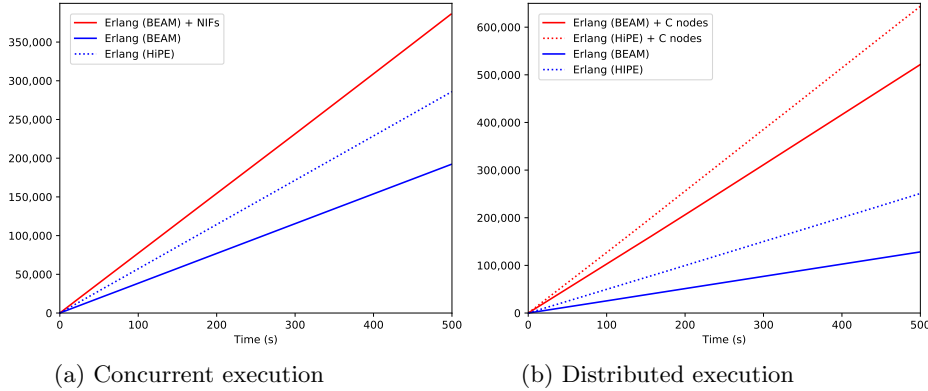(a) Concurrent execution   (b) Distributed execution

Fig. 2: In (a), Erlang (HiPE) reaches 74.1% of the performance of Erlang code that uses NIFs. In (b), Erlang (BEAM) reaches 24.5% of the performance of Erlang with C nodes; Erlang (HiPE) reaches 38.9% of the corresponding performance.

Discussion. It is perhaps surprising that an implementation that relies on Erlang for numerical computations is fairly competitive with C, with the observed difference amounting to a modest constant factor. In particular, the performance of natively compiled Erlang code is commendable. In the concurrent use case in particular, HiPE performs remarkably well. These are significant results for a number of reasons. From the perspective of programmer productivity, the relative

conciseness of Erlang code, compared to C, is worth pointing out. For instance, the line count of our C code that merely interfaces with the FANN library slightly exceeds the line count of the Erlang implementation of our entire ANN. Writing the former was more time-consuming than the latter. That being said, the tool Nifty [13], which automates the generation of NIF libraries based on C header files, may have simplified this task. However, as we wanted to limit external dependencies, this was not a viable option.

The main argument for using C is its high performance. Yet, a downside is that it is a low-level programming language. In particular, manual memory allocation and garbage collection are an abundant source of programming errors. In terms of programmer productivity, C therefore does not compare favorably with Erlang. As there are use cases where performance is not of the topmost priority, Erlang may be a viable alternative as it leads to a much shorter turnaround time between design, implementation, and execution.

The performance comparison between Erlang and C is arguably lopsided, due to using the open-source C library FANN. It originally appeared in 2003 and has been actively maintained for over a decade, even though development activity seems to have slowed down recently. On the other hand, we developed our Erlang implementation of an ANN relatively quickly and without the benefit of extensively using it in real-world situations. Because FANN has been much more optimized than our code, the true performance difference between the competing programming languages may be less than what our numbers indicate.

C nodes work very well as they can essentially be addressed like Erlang nodes. NIFs, on the other hand, have serious drawbacks.[6] They are executed as native extensions of the Erlang VM. Thus, a NIF that crashes will crash the Erlang VM. Furthermore, NIFs can cause state inefficiencies, which may lead to crashes or unexpected behaviors. Lastly, there is the issue of *lengthy work*: a NIF that takes too long to return may negatively affect the responsiveness of the Erlang VM. In the Erlang version we were using, a well-behaving NIF has to return within one millisecond. In exploratory benchmarking with data sets not much larger than the one we eventually used, we measured calls to NIFs that took longer than that. Consequently, we think it is too risky to use NIFs in a more taxing environment.

4 Related Work

There has been some preceding work in academia related to using functional programming languages for tackling machine learning tasks. About a decade ago, Allison explored using Haskell for defining various machine learning and statistical learning models [1]. Yet, that work was of a theoretical nature. Going back even further, Yu and Clack presented a system for polymorphic genetic programming in Haskell [22]. Likewise, this was from a theoretical perspective. More recently, Sher [18] did extensive work on modeling evolutionary computations. A central

---

[6] Refer to the section on Natively Implemented Functions (NIFs) in the official Erlang documentation for further details: http://erlang.org/doc/man/erl_nif.html (accessed on June 28, 2018).

part of his contribution is an ANN implemented in Erlang. However, his fairly complex system could only have been used as the starting point of our work with substantial modifications. One key difference is that individual nodes of the ANN are modeled as independent processes, and so are sensors and actuators. A related ANN implementation in Erlang is `yanni`,[7] which follows Sher's approach of using message passing, albeit only between layers.

5   Future Work

The `ffl-erl` project has influenced ongoing work in our research lab on a real-world system for distributed data analytics for the automotive industry [21]. In that system, Erlang is used for distributing assignments to clients, which operate on local data. Those clients can execute code written in an arbitrary programming language. Federated Learning is one of its use cases.

References

1. Allison, L.: Models for Machine Learning and Data Mining in Functional Programming. Journal of Functional Programming **15**(1), 15–32 (2005)
2. Bauer, H., Goh, Y., Schlink, S., Thomas, C.: The Supercomputer in Your Pocket. McKinsey on Semiconductors pp. 14–27 (2012)
3. Chen, D., Zhao, H.: Data Security and Privacy Protection Issues in Cloud Computing. In: Proceedings of the 2012 International Conference on Computer Science and Electronics Engineering (ICCSEE). vol. 1, pp. 647–651. IEEE (2012)
4. Coppola, R., Morisio, M.: Connected Car: Technologies, Issues, Future Trends. ACM Computing Surveys (CSUR) **49**(3), 1 – 36 (2016)
5. Cuccu, G., Togelius, J., Cudre-Mauroux, P.: Playing Atari with Six Neurons. arXiv preprint arXiv:1806.01363 (2018)
6. Evans-Pughe, C.: The Connected Car. IEE Review **51**(1), 42–46 (2005)
7. Fisher, R., Marshall, M.: Iris Data Set. UC Irvine Machine Learning Repository (1936)
8. Gybenko, G.: Approximation by Superposition of Sigmoidal Functions. Mathematics of Control, Signals and Systems **2**(4), 303–314 (1989)

---

[7] The corresponding code repository is located at https://bitbucket.org/nato/yanni (accessed on August 6, 2018).

9. Hornik, K.: Approximation Capabilities of Multilayer Feedforward Networks. Neural networks **4**(2), 251–257 (1991)

10. Johansson, E., Pettersson, M., Sagonas, K.: A High Performance Erlang System. In: Proceedings of the 2nd ACM SIGPLAN international conference on Principles and practice of declarative programming. pp. 32–43. ACM (2000)

11. LeCun, Y., Cortes, C., Burges, C.J.: Mnist Handwritten Digit Database. AT&T Labs [Online]. Available: http://yann.lecun.com/exdb/mnist (2010)

12. Lee, J., Kim, C.: A Roadside Unit Placement Scheme for Vehicular Telematics Networks. In: Advances in Computer Science and Information Technology. pp. 196–202. Springer (2010)

13. Löscher, A., Sagonas, K.: The Nifty Way to Call Hell from Heaven. In: Proceedings of the 15th International Workshop on Erlang. pp. 1–11. ACM (2016)

14. McMahan, H.B., Moore, E., Ramage, D., Hampson, S., et al.: Communication-efficient Learning of Deep Networks from Decentralized Data. arXiv preprint arXiv:1602.05629 (2016)

15. Nissen, S.: Implementation of a Fast Artificial Neural Network Library (FANN). Report, Department of Computer Science University of Copenhagen (DIKU) **31**, 29 (2003)

16. Orr, G.B., Müller, K.R.: Neural Networks: Tricks of the Trade. Springer (2003)

17. Sagonas, K., Pettersson, M., Carlsson, R., Gustafsson, P., Lindahl, T.: All You Wanted to Know About the HiPE Compiler (But Might Have Been Afraid to Ask). In: Proceedings of the 2003 ACM SIGPLAN workshop on Erlang. pp. 36–42. ACM (2003)

18. Sher, G.I.: Handbook of Neuroevolution Through Erlang. Springer Science & Business Media (2012)

19. Srihari, S.N., Kuebert, E.J.: Integration of Hand-Written Address Interpretation Technology into the United States Postal Service Remote Computer Reader System. In: Proceedings of the Fourth International Conference on Document Analysis and Recognition. vol. 2, pp. 892–896. IEEE (1997)

20. Tene, O., Polonetsky, J.: Privacy in the Age of Big Data: A Time for Big Decisions. Stan. L. Rev. Online **64**, 63–69 (2011)

21. Ulm, G., Gustavsson, E., Jirstrand, M.: OODIDA: On-board/Off-board Distributed Data Analytics for Connected Vehicles. arXiv preprint arXiv:1902.00319 (2019)

22. Yu, T., Clack, C.: Polygp: A Polymorphic Genetic Programming System in Haskell. Genetic Programming **98** (1998)

A   Mathematical Derivation of Federated Stochastic Gradient Descent

In Section 2.2 we briefly describe Federated Stochastic Gradient Descent. In the current section, we present the complete derivation. As a reminder, we stated that in Stochastic Gradient Descent, weights are updated this way:

$$w := w - \frac{\eta}{n} \sum_{i=1}^{n} \nabla F_i(w). \tag{6}$$

Furthermore, we started with the following equation, which is the objective function we would like to minimize:

$$F(w) = \frac{1}{n} \sum_{j=1}^{k} |P_j| F^j(w). \tag{7}$$

The gradient of F^j is expressed in the following formula:

$$\nabla F^j(w) = \frac{1}{|P_j|} \sum_{i \in P_j} \nabla F_i(w), j = 1, \ldots, k. \tag{8}$$

To continue from here, each client updates the weights of the machine learning model the following way:

$$w_j = w - \frac{\eta}{|P_j|} \sum_{i \in P_j} \nabla F_i(w). \tag{9}$$

On the server, the weights of the global model are updated. The original equation can be reformulated in a few steps:

$$w := \frac{1}{n} \left(\sum_{j=1}^{k} w_j |P_j| \right) \tag{10}$$

$$= \frac{1}{n} \sum_{j=1}^{k} \left(w - \frac{\eta}{|P_j|} \sum_{i \in P_j} \nabla F_i(w) \right) |P_j| \tag{11}$$

$$= \frac{1}{n} \sum_{j=1}^{k} |P_j| w - \frac{1}{n} \eta \sum_{j=1}^{k} \sum_{i \in P_j} \nabla F_i(w) \tag{12}$$

$$= w - \frac{\eta}{n} \sum_{i=1}^{n} \nabla F_i(w). \tag{13}$$

The reformulation in the last line is equivalent to Equation 6 above. In case the transformation between Eq. 12 and Eq. 13 is unclear, consider that the summand simplifies to

$$\frac{1}{n} \sum_{j=1}^{k} |P_j| w = \frac{1}{n} nw = w. \tag{14}$$

The second summand in Eq. 12 can be simplified as follows:

$$\frac{1}{n} \eta \sum_{j=1}^{k} \sum_{i \in P_j} \nabla F_i(w) = \frac{1}{n} \eta \sum_{i=1}^{n} \nabla F_i(w) = \frac{\eta}{n} \sum_{i=1}^{n} \nabla F_i(w). \tag{15}$$