Optimization in Federated Learning

Vukasin Felbab, Péter Kiss, and Tomáš Horváth

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

Abstract: Federated learning (FL) is an emerging branch of machine learning (ML) research, that is examining the methods for scenarios, where individual nodes possess parts of the data, and the task is to form a single common model that fits to the whole distribution. In FL, we generally use mini batch gradient descent for optimizing weights of the models that appears to work very well for federated scenarios. For traditional machine learning setups, a number of modifications has been proposed to accelerate the learning process and to help to get over challenges posed by the high dimensionality and non-convexity of search spaces of the parameters. In this paper we present our experiments on applying different popular optimization methods for training neural networks in a federated manner.

1 Federated Learning

Federated learning (FL) [1] is a new paradigm in Machine Learning (ML), that is dealing with an increasingly important distributed optimization setting, that came into view with the spread of small user devices and applications written for them that can profit from ML. The domain of ML models is often the data collected on the devices, thus, to train these models, one should incorporate the knowledge contained into the learning process. The traditional way for this would be to transfer the information gathered at the users to data centers, where the training takes place, then send back the trained models to the users. That, apart from the obvious privacy concerns, can incur a huge communication overhead along with the need for significant storage and computational resources at the place of centralized training.

The idea proposed in [1] is that, instead of moving the training data to centralized location, one could exploit the computational power residing at the user devices and distribute the training process across the participating nodes.

1.1 Distributed Optimization

In ML, the goal is to find a model for the training data that minimizes a loss function f that defines how our learned model distribution differs from the empirical distribution. This measure in general case can be formalized as a negative log likelihood.

$$f = -\mathbb{E}_{\mathbf{x}\sim p_{data}}[log\, p_{model}(\mathbf{x})] \tag{1}$$

That is, if a given example $\mathbf{x}$ is drawn from the training data distribution, what is the probability that it will be present in the same form in the model distribution as well. If the model is for predicting some value(s) $\mathbf{y}$ based on a vector of some attributes $\mathbf{x}$ this can be rewritten as

$$f(\mathbf{x},\mathbf{y},\mathbf{w}) = -\log p(\mathbf{y}|\mathbf{x};\mathbf{w}) \tag{2}$$

,

The problem we want to solve is to minimize the loss function f with respect to model parameters $\mathbf{w}$, that is an aggregation of the losses over all available data point as follows:

$$\min_{\mathbf{w}\in\mathbb{R}^d} f(\mathbf{w}),\ \text{where}\ f(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} f_i(\mathbf{w}), \tag{3}$$

where $f_i(\mathbf{w}) \overset{\text{def}}{=} f(\mathbf{x_i},\mathbf{y_i},\mathbf{w})$ denotes the loss on data point i given the parametrization $\mathbf{w}$.

In the setup of FL the characteristics of data distribution from which our training examples $(\mathbf{x_i},\mathbf{y_i})$ will be drawn, are the following:

1. *Massively Distributed*. Data points are stored across a large number K of nodes. In particular, the number of nodes can be much bigger than the average number of training examples stored on a given node (n/K).

2. *Non-IID*. Data on each node may be drawn from a different distribution, i.e. the data points available locally are far from being a representative sample of the overall distribution.

3. *Unbalanced*. Different nodes may vary by orders of magnitude in the number of training examples they hold.

If we adapt the objective function (see Eq. 3) to these characteristics, our problem can be defined as introduced in the following paragraphs.

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

We can then define the local loss for node as $F_k(w) \overset{\text{def}}{=} \frac{1}{n_k}\sum_{i\in\mathscr{P}_k} f_i(w)$ Thus the problem to be minimized will become:

$$min_{\mathbf{w}\in\mathbb{R}^d} f(\mathbf{w}) = \sum_{k=1}^{K} \frac{n_k}{n} F_k(\mathbf{w}). \tag{4}$$

To solve the problem in (4) the simplest algorithm is FederatedSGD introduced in [2], that is equivalent to minibatch gradient descent over all data, and it is a simple application of distributed synchronous Stochastic Gradient Descent (SGD) [3] for the described setup.

Algorithm 1 FederatedSGD

procedure SERVER
  initialize $\mathbf{w_0}$
  for $t = 0; 1; 2; ...$ **do**
    for all k in the K nodes in parallel **do**
      $w_{t+1}^k \leftarrow \text{ClientUpdate}(k, \mathbf{w}_t)$
    end for
    $\mathbf{w}_{t+1} = \sum_{k=1}^{K} \frac{n_k}{n} \mathbf{w}_{t+1}^k$
  end for
end procedure
procedure CLIENTUPDATE(k, w)
  $\mathscr{B} \leftarrow$ split $\mathscr{P}_k$ to set of batches
  for all $b \in \mathscr{B}$ **do**
    $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla f(\mathbf{w}, b)$
  end for
  return W
end procedure

In neural network (NN) optimization, due to the non convexity of the loss functions, the most used methods for optimization of network parameters are gradient based, more specifically the versions of SGD [4]. Gradient descent methods take derivatives of loss function according to the parameters of the model, then move the parameter values in the negative of the gradient.

The pure form of SGD samples a random function (e.g a random training data point) $i_t \in 1, 2, ..., n$ in iteration t and performs the update:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \nabla f_{i_t}(\mathbf{w}_t), \tag{5}$$

where η_t denotes the learning rate, which is, in the base case, decaying during the learning to enforce convergence. Intuitively, SGD works because evaluating the gradient at a single training example gives an unbiased estimation of derivative of the error function over all the training examples: $\mathbb{E}[\nabla f_{i_t}(\mathbf{w})] = \nabla f(\mathbf{w})$.

In practice, instead of applying the gradient for $\mathbf{w}$ at each example, usually an average of gradients over b randomly chosen examples is used, that are evaluated at the same $\mathbf{w}$. This method is called minibatch gradient descent (MBGD), that better exploits parallel computational capabilities of the hardware. (MBGD is still commonly referred to as SGD) Though SGD/MBGD in the above form is very popular in optimization, the basic approach can sometimes result in very slow learning. To tackle the challenges incurred by high curvature and noisy gradients of the loss function of NN, a range of method has been proposed based on exponentially decaying average of the gradients or on adapting learning rates. [5]

In this paper, we investigate the effects of these methods on the performance , of federated training of artificial neural networks.

1.2 Momentum techniques

Momentum based techniques [6] use a velocity term during learning, that is an exponentially weighted average over the gradients of the past.

$$\begin{aligned} \mathbf{v} &\leftarrow \beta\mathbf{v} - \nabla_{\mathbf{w}}(\frac{1}{m}\sum_{i=1}^{m} f_i(\mathbf{w})) \\ \mathbf{w} &\leftarrow \mathbf{w} + \mathbf{v} \end{aligned} \tag{6}$$

This term, on one hand, accelerates the learning process and, on the other hand, helps to get over noisy gradient and local minima or flat points of surface defined by the error function f.

A variant of momentum algorithm is introduced in [7] and is based on the Nesterov's accelerated gradient method, that differs from the standard momentum of (6) in the place of the evaluation of the gradient.

$$\begin{aligned} \mathbf{v} &\leftarrow \beta\mathbf{v} - \nabla_{\mathbf{w}}(\frac{1}{m}\sum_{i=1}^{m} f_i(\mathbf{w} + \alpha\mathbf{v})) \\ \mathbf{w} &\leftarrow \mathbf{w} + \mathbf{v} \end{aligned} \tag{7}$$

In Nesterov's momentum, the gradients are evaluated incorporating the velocity. This can be interpreted as adding a correction factor to the standard momentum algorithm [5].

1.3 Adaptive Learning Rates

Setting up learning rates is one of the most important factor in the learning process and deeply influences the performance. Thus, finding methods to adapt the learning rate might yield a substantial increase in speed of the learning. The AdaGrad algorithm [8] adjusts the learning rates individually for each parameter, taking into account the whole history of the parameters, following the assumption, that if the magnitude of the gradients is big than it should be increased:

$$\eta_t = \frac{\eta}{\sqrt{\sum_{\tau=1}^{t-1} g_\tau^2}}, \tag{8}$$

where $g = \frac{\partial f}{\partial \mathbf{w}_j}$ for some parameter $\mathbf{w}_j$, and thus η_t will be the learning rate belonging to $\mathbf{w}_j$ a timestep t.

It has been found empirically that aggregating the gradients from the beginning of the optimization can lead to too

fast decay in the learning rate, that, in some cases, leads to weak performance.

To remedy this problem RMSProp [9] (the same time proposed by the authors of AdaDelta [10]) replaces this aggregation with a decaying average, in the form:

$$\mathbf{v}_t = \rho \mathbf{v}_{t-1} + (1-\rho) g_t^2$$
$$\eta_t = \frac{\eta}{\sqrt{\mathbf{v}_t + \varepsilon}} \tag{9}$$

RMSProp has been proven very effective in non-convex optimization problems of NN, thus, it is the most often used technique in practice.

According to the explanation in [5], AdaGrad is designed to converge rapidly when applied to convex functions. In non-convex cases it should pass many structures before arriving at a convex bowl, and, since it accumulates the entire history of the squared gradient, it can shrink prematurely and eventually vanish. In contrast, discarding the old gradients in the RMSProp case enables learning to proceed rapidly after finding the convex bowl, equivalently as if AdaGrad would have been initialized within that convex area.

1.4 Adam

The name of the Adam algorithm [11] comes from "adaptive momentum", and can be viewed as the combination of adaptive learning rates and momentums.

$$\mathbf{g}_t \leftarrow \nabla_{\mathbf{w}}\left(\frac{1}{m}\sum_{i=1}^{m} f_i(\mathbf{w}_{t-1})\right) \tag{10}$$

$$\mathbf{m}_t \leftarrow \frac{\beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\mathbf{g}_t}{1-\beta_1^t} \tag{11}$$

$$\mathbf{v}_t \leftarrow \frac{\beta_2 \mathbf{v}_{t-1} + (1-\beta_2)\mathbf{g}_t^2}{1-\beta_2^t} \tag{12}$$

$$\mathbf{w}_t \leftarrow \mathbf{w}_{t-1} - \frac{\eta \mathbf{m}_t}{\sqrt{\mathbf{v}_t} + \varepsilon} \text{(element-wise)} \tag{13}$$

In Adam, the weight update is given by applying the RMSProp learning rate (12) on the momentum (11). (In Equation (12) and (11) the denominator is applied bias correction on the estimates.) We are not aware of clear theoretical understanding why this is advantageous, however, it seems to work very well in practice and became a de facto default optimization technique for a lot of ML practitioners.

2 Experimental setup

For analyzing thew performance of the optimizer algorithms, we implemented a simulation environment that trains multiple local NN models that would be aggregated into one common model, according to the Algorithm 1.

Compared to Algorithm 1, we have been varying the CLIENTUPDATE(k,w) method, where the local updates have been calculated. (Except for first experiment, since it describes exactly the MBGD method).

The new CLIENTUPDATE(k,w) method is introduced in the Algorithm 2.

Algorithm 2 ClientUpdate

1: **procedure** CLIENTUPDATE(k,w)
2: $\mathscr{B} \leftarrow$ split $\mathscr{P}_k$ to set of batches
3: **for all** $b \in \mathscr{B}$ **do**
4: $\Delta\mathbf{w} = Optimizer(\mathbf{w}, b)$
5: $\mathbf{w} \leftarrow \mathbf{w} - \Delta\mathbf{w}$
6: **end for**
7: return W
8: **end procedure**

Naturally, all the optimizers have their own hyperparameters which should be tuned to get the best possible result. However, for this experiment we used only the recommended values for them (that are in fact the default values in Keras/TensorFlow libraries).

2.1 Topology

The model we used is a simple multilayer perceptron. The input layer consists of 784 input units that is the flatten representation of the input images of size 28×28 pixels. The input is connected to one hidden layer of 128 neurons with ReLU activation. The output layer corresponds to the 10 output classes, thus it has 10 neurons with softmax activation.

In the implementation of the network, we relied on Keras NN API on a TensorFlow backend.

2.2 Data

For the experiment, we have chosen the Fashion MNIST dataset [12] that was planned to replace the MNIST benchmark database.

From the characteristics of the FL scenario, in this experiment, we focused on non-iid nature of the data. That is, we have created local datasets of a highly skewed manner. Namely, training data at a given node contains exclusively, or almost exclusively, instances from the same class.

For these experiment, we have not taken into account the unbalanced nature, and each node have been assigned the same amount of data. Our idea here was that if something works in this simple setup then it might work in use cases closer to the real world problem.

Due to the lack of computational resources we also ignored the "massively distributed" condition and set the number of nodes to 10.

The distributions of the local datasets we tried in the experiments are the following:

99% non-iid The training data has been split into two parts in the ratio of 99%-1%, where the parts are independent and identically distributed, as best as possible. The 99% part will be assorted accorded to classes and then one class assigned to one of the nodes. The 1% part will be equally split into 10 parts and then added to the dataset of the particular nodes.

full-non-iid In the second test case, we assorted fully the training data and each node receives a dataset consisting of instances belonging purely to one single class.

2.3 Hyperparameters

To measure general applicability of the examined algorithms on the problem of FL, we executed the learning process multiple times, using different parametrisations. In setting the hyperparameters we followed the Method of GridSearch, that is we defined a set of possible values for each hyperparameter, then run the algorithms with all the combinations. At defining the set of the possible values we tried to include extremities and generally recommended values. In addition to the parameters described in Section 1 we also included experimenting with the decay of the learning rate. Here we only tried tried nevertheless two case at each configuration of the other parameters, the one without decay, and the one with time based decay, where the learning rate a time t will be $\eta_t = \frac{\eta_0}{1+\phi * t}$ with the decay rate parameter $\phi = \frac{\eta_0}{max\{t\}}$.

3 Results

FedSGD - Simple Minibatch Gradient Descent As a baseline we run the experiment with the standard Minibatch Gradient Descent optimiation. The experiment result is shown in Figure 1. It is clear that, as it often happens, the most simple algorithm, MBGD places the baseline rather high for the more sophisticated optimizer algorithm. It performs very well for the 99% non-iid datasets and surprisingly well with the full-non-iid datasets, achieving an accuracy close to 75% in the 30 iterations with the best configuration of hyperparameters ($\eta = 0.001$, no decay).

Moreover, in results it seems like on both distribution too high learning rate without decay leads to a poor performance.

FedSGD + Nesterov momentum In Figure 2, the results using the Stochastic Gradient Descent with Nesterov momentum can be seen. We found that incorporating local momentum into computing the partial directions of the updates has a strong positive effect both on performance and convergence rate of the aggregated model at both data distributions.

The best performing configurations reached in the first couple of iterations the highest accuracy achieved by the baseline during the entire experiment. According to our results the higher the value β (that is the past directions influence stronger the update) is generally the better performance, apparently independently from η and decay rate.

FedSGD + AdaGrad The AdaGrad algorithm yields an even better performance 3, on the 99% non-iid datasets. Using this method results in the fastest convergence until the 70% of the baseline. In the first 30 rounds, though AdaGrad's performarnce drops dramatically full-non-iid datasets, reaching at most a 25% accuracy without obvious perspective further improvement. It might be interesting to check how many random training examples need to be put into the full-non-iid datasets to achieve the very good performance of the AdaGrad which is measured with the 99% non-iid datasets.

FedSGD + RMSProp The RMSProp optimizer is used in the experiment which has produced the statistics in Figure 4. We experienced that this optimizer algorithm apart from the stronger variance of performance seems to approach the accuracy of the AdaGrad and Nesterov momentum methods, and outperforming MBGD baseline as well on the 99% non-iid distribution. One the other hand though the accuracy on the full-non-iid still achieves a significantly worse performance compared to MBGD and Nesterov momentum, however leraning curve is much more promising than the one of AdaGrad. It reaches in the best performing setups about 50% accuracy, and shows an emerging tendency as well.

FedSGD + Adam The last experiment, we were applying Adam. The method is one of the most popular and often default optimizer(11), thanks to fast convergence, high accuracy in the traditional NN learning, and to its robustness to hyperparameter settings. In our experiment however, Adam worked with a very similar effectiveness to RMSProp, and has been definitely outperformed by MBGD and Nesterov momentum (Figure 5) regarding to performance and smoothness of learning on the full-non-iid datasets.

4 Conclusion

In our experiment we found, that the best performing optimizer algorithms for both the distribution are Minibatch Gradient Descent without and with Nesterov momentum, whilst Adadelta and RMSProp is promising despite their poor performance on fully non-IID datasets. As one could have assumed, the presence of the non-iid part of the training data has a very strong regularizing effect even if its weight seems to negligible compared to the dominating class.

SGD

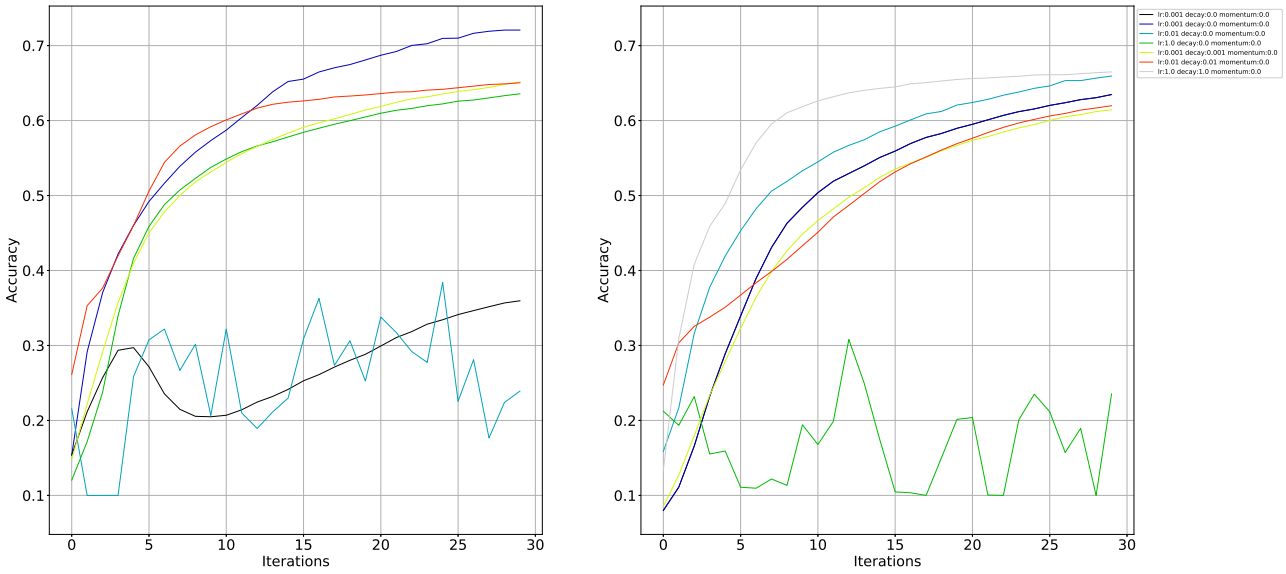

Figure 1: FedSGD baseline(simple minibatch updates)

Nesterov

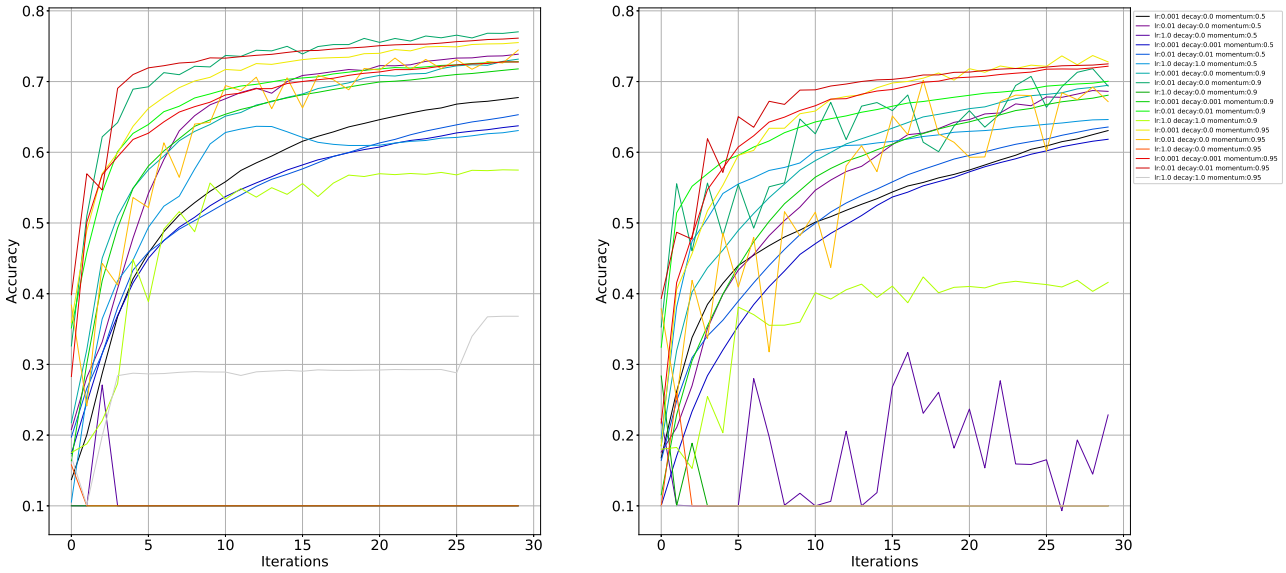

Figure 2: FedSGD + Nesterov momentum

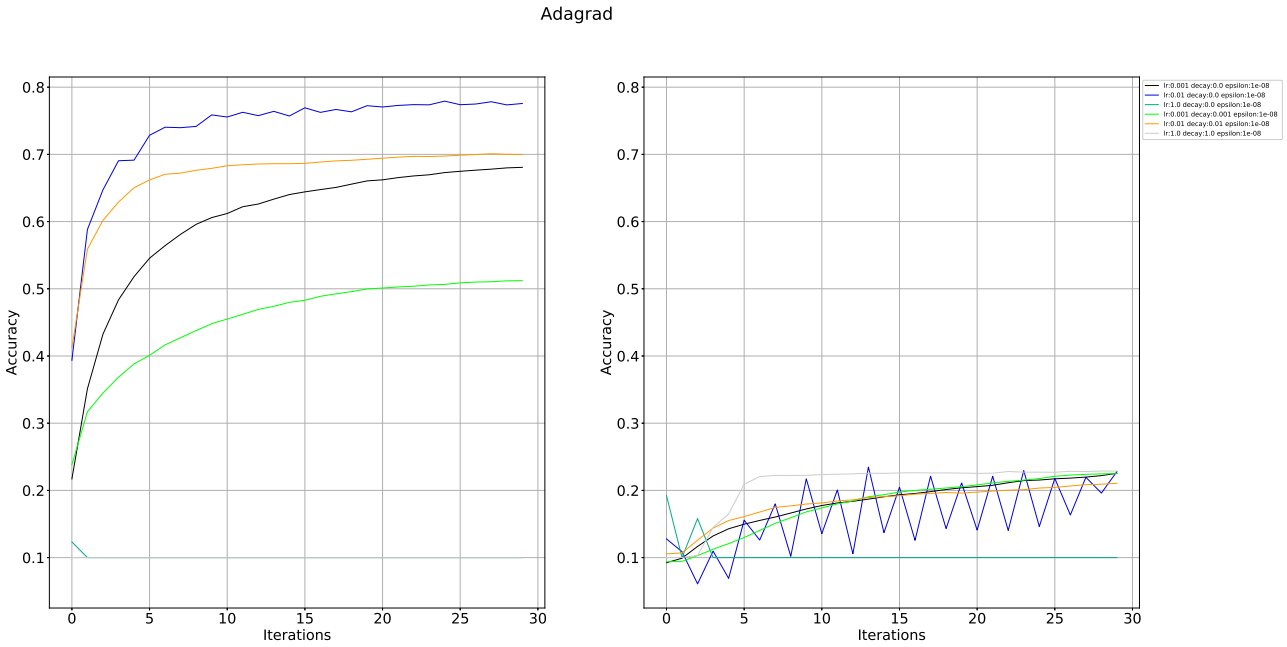

Figure 3: FedSGD with AdaGrad

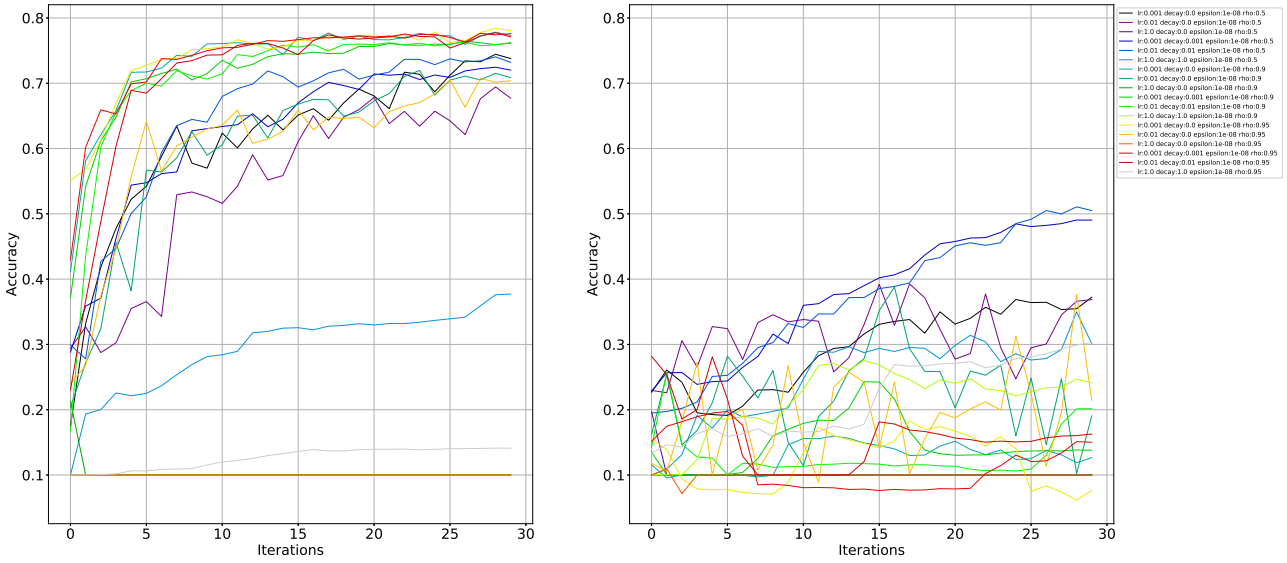

Figure 4: FedSGD with RMSProp

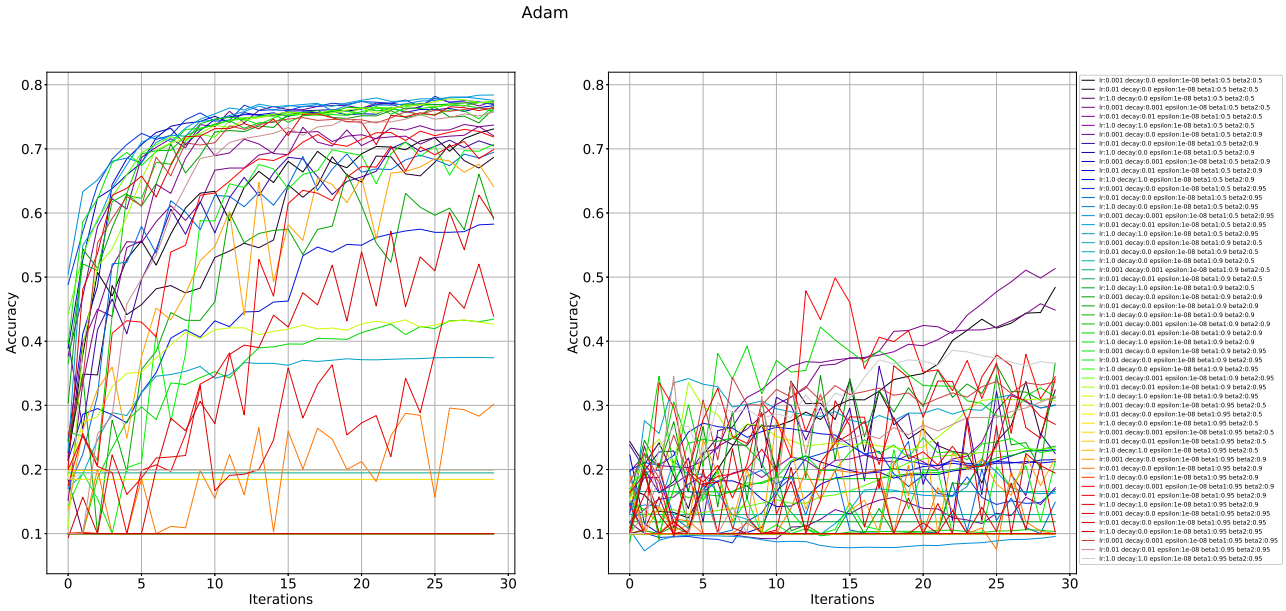

Figure 5: Federated Averaging - Adam

In general we experienced that methods that are intended to reduce the variance of the gradient direction works actually quite well for our specific scenario (1). This can be because momentum techniques can be seen as an averaging over the subsequent gradients, leading to a less and less biased estimate of the optimal update direction. The fact that strong momentum (high β) seems to help in the big majority of configurations of the other parameters supports this idea.

On the other hand methods that aim at adapting the magnitude of the gradients seem to harm the learning process (2) in the full-non-iid case. The reason behind this phenomenon is most probably, that the local optimisers update their inner state – which here corresponds to η learning rate – based on the local gradients. In each local training round according to our intuition the magnitude of gradients start growing, since the aggregation of the local models moved the model away from the locally optimal model, but as we approaching again the local optima, they start shrinking again, resulting in slower start (smaller η) of optimization in the next iteration.

The extremely poor performance of AdaGrad on the full-non-iid dataset can support this intuition, since it prevents even the described fluctuation of the learning rate, instead it decreases it continuously.

The good performance of these algorithms on the 99% non-iid might be explainable with the presence of gradients of really big magnitude in the decaying average that controls the learning rate keeping it at an effective level.

Another interesting phenomenon is that in case of Adam – where momentum and adaptive learning rates are both applied – the strong deccelerating effect of learning rate adaption apparently overrides the help of the momentum. However looking at magnitude of performance differences it might be understandable.

Although according to our results these optimizers are not clearly beneficial in perspective of finding the best global model, they still could be useful for optimizing the global model at clients. One can argue, that in the end the goal of the entire federated optimization is to provide clients with a model performs well on their own data.

Acknowledgements EFOP-3.6.3-VEKOP-16-2017-00001: Talent Management in Autonomous Vehicle Control Technologies - The Project is supported by the Hungarian Government and co-financed by the European Social Fund.

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

References

[1] J. Konečný, H. B. McMahan, D. Ramage, and P. Richtárik, "Federated optimization: Distributed machine learning for on-device intelligence," *arXiv preprint arXiv:1610.02527*, 2016.

[2] H. B. McMahan, E. Moore, D. Ramage, S. Hampson *et al.*, "Communication-efficient learning of deep networks from decentralized data," *arXiv preprint arXiv:1602.05629*, 2016.

[3] J. Chen, X. Pan, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting distributed synchronous sgd," *arXiv preprint arXiv:1604.00981*, 2016.

[4] L. Bottou, "Online learning and stochastic approximations," *On-line learning in neural networks*, vol. 17, no. 9, p. 142, 1998.

[5] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[6] B. T. Polyak, "Some methods of speeding up the convergence of iteration methods," *USSR Computational Mathematics and Mathematical Physics*, vol. 4, no. 5, pp. 1–17, 1964.

[7] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *International conference on machine learning*, 2013, pp. 1139–1147.

[8] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2121–2159, 2011.

[9] G. Hinton, N. Srivastava, and K. Swersky, "Neural networks for machine learning," *Coursera, video lectures*, vol. 264, 2012.

[10] M. D. Zeiler, "Adadelta: an adaptive learning rate method," *arXiv preprint arXiv:1212.5701*, 2012.

[11] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[12] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," *arXiv preprint arXiv:1708.07747*, 2017.