Reappearance and Optimization of the FedDyn Algorithm

Zhao Zhang
Sussex Institute of Artificial Intelligence, Zhejiang Gongshang University, Hangzhou, Zhejiang Province, 310018, China

Keywords: FedDyn Algorithm, MNIST, F-MNIST.

Abstract: Federated learning has been widely used as a way to protect data privacy. However, the existing federated learning algorithms have the problems of large differences in data distribution and low efficiency of model updating. Therefore, the FedDyn algorithm was born to solve the above problems. This paper uses Python programming software and FedDyn algorithm to build a simple federated learning framework composed of five major parts: a fully connected neural network model, client model training function, global model aggregation function, global model test function, and main function and carries out model training and performance improvement optimization of the framework, as well as code efficiency and performance. Three aspects of system stability and reliability were optimized, and two different data sets, Modified National Institute of Standards and Technology (MNIST) and Fashion-MNIST (F-MNIST), were used to test whether the algorithm optimization strategy proposed in this paper improved the algorithm performance. Finally, it is concluded that the algorithm can be optimized effectively by adjusting multiple parameters of the experimental model simultaneously.

1 INTRODUCTION

In the era of big data, with the improvement of privacy protection awareness, federated learning that can protect data privacy has attracted wide attention to (Zhao et al. 2023). As a distributed learning paradigm, federated learning allows multiple devices or servers to jointly train models while maintaining data localization, to help preserve data privacy (Mónica & Haris 2024). However, when the client data is not independent and identically distributed (non-IID), that is, the data is not evenly distributed on each client, traditional federated learning algorithms (such as FedAvg) may encounter the problem of the performance performance. This is because the update of the model on different clients may move in different directions, causing the global model to converge to the optimal state (Li et al. 2019).

The study of the FedDyn algorithm is of great significance to the field of federated learning because it directly solves a key challenge in the federated learning system: client data heterogeneity (Yiyang 2022). In practice, there are often significant differences in the distribution of the data held by different clients, which will lead to the deviation of the model locally trained by each client, and affect the performance and generalization ability of the global model. FedDyn By introducing dynamic regularization terms, it effectively reduces the difference in model parameters between clients, thus helping to improve the accuracy of the global model (Wenbo 2023).

The proposal of the FedDyn algorithm is of great research significance for realizing a more efficient and fair federated learning model. First, it addresses the impact of data heterogeneity on model training through dynamic regularization techniques, which is crucial to improving the model performance on real-world diverse datasets. Second, FedDyn helps to improve the global convergence rate of the model, enabling the federated learning system to achieve better learning results in a shorter time. Finally, the design concept of the algorithm has also inspired other studies in the field of federated learning, providing new ideas on how to design more robust distributed learning algorithms.

Although the FedDyn algorithm has achieved significant theoretical progress, and partly achieves the goal of model consistency. However, in practice, especially in large-scale deployment, the algorithm still faces some unresolved problems and challenges to (Acar et al. 2021). For example, the computational efficiency of algorithms, adaptability

Zhang, Z.
Reappearance and Optimization of the FedDyn Algorithm.
DOI: 10.5220/0012839100004547

In *Proceedings of the 1st International Conference on Data Science and Engineering (ICDSE 2024)*, pages 569-576
ISBN: 978-989-758-690-3

to different types of data sets, and performance in environments with limited practical communication still need to be further studied. Moreover, how to reduce the computational and communication burden of clients while maintaining the effect of the algorithm is also an important direction for future research. Finally, it is also a research gap worth exploring in choosing and adjusting the dynamic regularization terms in the FedDyn algorithm to accommodate different learning tasks and network conditions.

This paper uses Python programming software, using the FedDyn algorithm to build a simple federated learning framework, and proposes the framework into three categories, a total of 13 optimization methods. This paper aims to test whether the proposed algorithm optimization strategy indeed improves the algorithm performance with two different datasets. The final study shows that simultaneously adjusting multiple parameters of the experimental model can effectively optimize the experimental conclusion of the algorithm.

2 CODE REPETITION OF FedDyn ALGORITHM

The code for reproducing the FedDyn algorithm implements a simple federated learning scenario, which includes the following main parts:

1. Fully connected neural network model (class SimpleModel):

This is a simple neural network with a fully connected layer used to flatten 28x28 images into a vector and output to a layer with 10 output nodes (corresponding to 10 categories in the Fashion-Modified National Institute of Standards and Technology (F-MNIST) dataset).

2. Client model training function (train_client_model):

This function is responsible for training the model for the individual clients. It receives the client's model, optimizer, data loader, regularization strength alpha, global model weights, and device information (Central Processing Unit (CPU) or Graphic Processing Units (GPU)). The training process includes forward propagation, loss calculation (including the dynamic regularization term in the FedDyn algorithm), backpropagation, and parameter update.

3. Global model aggregation function (aggregate_global_model):

After all clients complete the training, this function is responsible for aggregating the model updates for all clients into a global model. The aggregation is done by calculating the average of all client updates.

4. Global model test function (test_model):

This function is used to evaluate the performance of the global model on the test set. It calculates the test loss and accuracy.

5. Main function (main):

This is the entry point of the program, it sets the training parameters (such as equipment information, regularization strength alpha, client number, communication rounds, batch size, learning rate, etc.), load and preprocess data set, divides data set to different clients, initialize the global model, and cycle the client model training and global model aggregation, finally test the performance of the global model.

The entire process simulates the process of federated learning, where multiple clients train the model locally and then send the model updates to a central server for aggregation. This process is repeated until the predetermined number of communication rounds is reached. The goal of federated learning is to protect data privacy in this way because the raw data does not need to leave the local client.

Results were obtained by training the fully connected neural network on the MNIST dataset and the FMNIST dataset as shown in Table 1.

The mean values of the initial model loss values were 0.0107 and 0.0183 on the MNIST and FMNIST datasets.

On the MNIST and FMNIST data sets, the proportion of the correct samples with the initial model predictions was (90.49%) and (79.70%).

Table 1: Experimental results of FedDyn algorithm.

Data set	MNIST	FMNIST
Experimental result	Average loss: 0.0107, Accuracy: 9049/10000 (90.49%)	Average loss: 0.0183, Accuracy: 7970/10000 (79.70%)

3 THE SHORTCOMINGS AND IMPROVEMENT SCHEME OF THE REPEATED FedDyn ALGORITHM

There are three main disadvantages of code:

Model training and performance improvement:

I. Single-round training: adjust the number of rounds that each client trains locally. 1. Fixed regularization strength: adjust the regularization strength parameter alpha, which may affect the generalization ability of the model. 2. Model initialization: Using different initialization strategies, may affect the model convergence rate and performance. 3. Client selection: Choosing different clients or different data distributions may affect the accuracy and generalization ability of the model.

II. Code Efficiency and Performance Optimization: Data loader performance: optimize the performance of the data loader, such as through prefetching, multithreaded loading, etc.

III. System Stability and Reliability: 1. Exception handling: Add the exception handling mechanism to ensure the stable operation of the system. 2. Random seeds: Random seeds were set to ensure reproducible experiments. 3. Model complexity: Adjusting model complexity may affect training time and model performance.

In these optimization scenarios, the following scenarios may directly improve the model accuracy: 1. Single-round training: Increasing the number of local training rounds may improve the performance of each client model, thus improving the accuracy of the global model. 2. Fixed regularization strength: finding a suitable regularization strength can help the model to generalize better, which may improve the accuracy. 3. Client selection: If the selected client has more representative data or a more balanced data distribution, the accuracy of the model may be improved (Bo 2020).

Other optimization schemes may indirectly affect the accuracy of the model, such as making more training rounds possible by improving the training efficiency or accelerating the model convergence by improving the model initialization strategy. However, most optimization schemes focus on improving code efficiency, reducing resource consumption, and increasing the stability of the system (Wirth et al. 2023).

4 OPTIMIZATION RESULTS OF THE REPRODUCED FedDyn ALGORITHM

4.1 Fully Connected Neural Network Model (Class SimpleModel)

1. Increase the number of training rounds from 2 to 5,7,10 rounds: The fully connected neural network was trained on the MNIST dataset. The results are shown in Table 2 when the number of training rounds is 5, 7, and 10 rounds. On the MNIST dataset, the new model loss values averaged 0.0095,0.0092 and 0.0090. On the MNIST dataset, the number of samples predicted by the new model was (91.44%), (91.60%) and (91.87%).

Table 2. Experimental results of the optimized algorithm.

Number of training rounds	5	7	10
Training results	Average loss: 0.0095 Accuracy: 9144/10000 (91.44%)	Average loss: 0.0092 Accuracy: 9160/10000 (91.60%)	Average loss: 0.0090 Accuracy: 9187/10000 (91.87%)

Loading part of the data set: replace datasets. MNIST with datasets. Fashion MNIST updates the mean and standard deviation of the data conversion and completes the data set change. Replace the MNIST with the FMNIST dataset. The experimental results are shown in Table 3 when the number of training rounds was 5,7, and 10 rounds. On the FMNIST dataset, the mean values of the new model loss values were 0.0163,0.0158 and 0.0153. On the FMNIST dataset, the number of samples predicted by the new model was (81.76%), (82.52%) and (82.59%).

Table 3: Experimental results of the optimized algorithm.

Number of training rounds	5	7	10
Training results	Average loss: 0.0163 Accuracy: 8176/10000 (81.76%)	Average loss: 0.0158 Accuracy: 8252/10000 (82.52%)	Average loss: 0.0153 Accuracy: 8259/10000 (82.59%)

Experimental findings suggest that increasing the number of training rounds may improve the performance of the model. If the number of training rounds is continuously increased, the model may eventually be overfitting.

2. Adjust the regularization intensity [0.0001, 0.001, 0.01, 0.1, 1]:

The experimental results of the FMNIST dataset are shown in Table 4 [0.0001, 0.001, 0.01, 0.1, 1]. On the FMNIST dataset, the mean values of the new model loss values were 0.0182, 0.0182, 0.0182, 0.0189, and 0.0245. On the FMNIST dataset, the proportion of the total sample number that the new model predicted correctly was (79.87%), (79.83%), (80.01%), (79.53%), and (75.46%).

Table 4: Experimental results of the optimized algorithm.

Alpha price	0.0001	0.001	0.01	0.1	1
Experimental result	Average loss: 0.0182 Accuracy: 7987/10000 (79.87%)	Average loss: 0.0182, Accuracy: 7983/10000 (79.83%)	Average loss: 0.0182, Accuracy: 8001/10000 (80.01%)	Average loss: 0.0189, Accuracy: 7953/10000 (79.53%)	Average loss: 0.0245, Accuracy: 7546/10000 (75.46%)

Furthermore, an alpha_values list can be set to try different alpha values and evaluate model performance using each validation set after training rounds, selecting the best alpha value. In federated learning, the regularization strength alpha is an important hyperparameter that controls the degree to which the local model weights are close to the global model weights (Tivnan et al. 2023). Experiments find that alpha is set too high, so the local model may overfit the weight of the global model, which may ignore the specificity of the local data; Set too low, the local model may overfit the local data, not enough to the global model, which may reduce the generalization performance of the global model.

3. Model initialization:

In the train_client_model function, can directly use global_weights instead of extracting them from global_model every time. In the aggregate_global_model function, can use methods outside of torch. stack to aggregate model updates, such as directly calculating the average, instead of stacking first before averaging. In the test_model function, one can avoid using the division by directly accumulating the losses and the number of correct predictions. These optimizations may have some minor impact on performance but may be more pronounced when training large models or processing large amounts of data. New code is compared to the source code.

When calculating the dynamic regularization term in the train_client_model function, it treats the global_weights [param _ key] differently:

In the new code, global_weights [param _ key] was transferred to the corresponding device. In the source code, global_weights [param _ key] is used directly without device transfer. This change is made to ensure that the model parameters and the global weights are on the same device (CPU or GPU) to avoid operational errors. The calculation of test loss in the test_model function. In the new code, the loss is calculated and multiplied by data. size (0) to get the average loss for the entire data set. In the source code, the loss value is not multiplied by the size of the batch. This change is made to correct the calculation method of the test loss, ensuring that the average loss of the entire test set is obtained.

The variable name in the aggregate_global_model function:

In the new code, the update is used as the temporary variable name in the loop. In the source code, client_update is used as the temporary variable name in the loop: This change is just a different choice of variable names and has no impact on the code logic. The variable name of the client data loader is different. In the new code, the dataset is used as the temporary variable name in the loop. In the source code, client_dataset is used as the temporary variable name in the loop:

This change is also just a different choice of variable names and has no impact on the code logic. In conclusion, the main changes are the calculation method of device transfer of the global weights in the train_client_model function and correcting the test loss in the test_model function. Other changes involve only different choices of variable names.

The experimental results are shown in Table 5. On the MNIST data set, 0.3437 was the mean of the new model loss value on the dataset. On the MNIST dataset, the total number of samples as predicted correctly by the new model was (90.45%). On the FMNIST dataset, the average of the new model loss value was 0.5820. On the FMNIST data set, the proportion of the correct samples predicted by the new model was (80.15%).

Table 5: Experimental results of the optimized algorithm.

Data set	MNIST	FMNIST
Experimental result	Average loss: 0.3437, Accuracy: 9045/10000 (90.45%)	Average loss: 0.5820, Accuracy: 8015/10000 (80.15%)

4. Client selection:

A new mechanism is introduced to select a portion of the clients for training instead of training all clients for each round. This approach is often called client sampling in federated learning (Chenyang 2022). Differences between the old and new codes. A new variable client_fraction was added to the main function to specify the proportion of clients selected in each round of training. In each round, a portion of clients were randomly selected by the random. Sample method instead of training all clients. Only the model updates for the selected client are aggregated into the global model. This approach can reduce computational resource consumption and is shown in practice to be able to improve the efficiency of federated learning without significantly reducing model performance.

The experimental results on the FMNIST dataset are shown in Table 11. On the FMNIST data set, the mean value of the new model loss value was 0.0182. On the FMNIST data set, the proportion of correct sample numbers by the new model is (80.08%).

Table 6: Experimental results of the optimized algorithm.

Data set	FMNIST
Experimental result	Average loss: 0.0203, Accuracy: 7799/10000 (77.99%)
	Average loss: 0.0182, Accuracy: 8008/10000 (80.08%)

4.2 Code Efficiency and Performance Optimization

1. Performance of the data loader:

The old and new codes are the same for most of the content, which all define a simple fully connected neural network model, training and aggregation of client models, and testing the performance of the global model. However, there are some differences, mainly focusing on the creation and configuration of the data loader (DataLoader). The specific differences are shown as follows:

Use of the num_workers parameter: In the new code, num _ workers, the parameter, was not used when creating DataLoader in the old code. The num_workers parameter determines the number of child processes used during the data loading process. If set to 0 (or not set, such as old code), all data loading is completed in the main process. If set to a positive integer, the data loading operation is performed in parallel in a specified number of child processes.

Value of the 3num_workers parameter: In the new code, num_workers is set to 4, which means that the data loading process will try to load in parallel the data.

These two differences may have an impact on the speed and efficiency of data loading. Using multiple work processes can speed up data loading, especially when processing large datasets and/or running on a multi-core CPU. However, in some cases, increasing the number of working processes does not always result in a performance improvement, as it may cause overhead in inter-process communication, especially when the number of working processes exceeds the number of CPU cores.

The experimental results on the FMNIST dataset are shown in Table 9. On the FMNIST dataset, the mean of the new model loss value was 0.0183. On the FMNIST data set, the proportion of correct sample numbers by the new model is (80.10%).

Table 7: Experimental results of the optimized algorithm.

Data set	FMNIST
Experimental result	Average loss: 0.0183, Accuracy: 8010/10000 (80.10%)

Device allocation optimization typically involves ensuring that all models and data are on the right device (e.g., GPU or CPU) and reducing unnecessary data transfers when global model weights are shared between multiple clients.

Compared to the source code, the optimization points in the new code include:

1. The weights of the global model (global_weights) are now extracted from the global model at the beginning of each round and used in the whole round, which avoids re-extracting the weights from the global model at each client training, reducing the transmission of data between devices.

2. After the aggregation update, now only update the weight dictionary (global_weights) of the global model, rather than reloading the state of the global model after each client update. This reduces unnecessary state-loading operations.

3. Finally, after all training rounds, only need to load the final global weight into the global model once to test the model performance.

These optimizations reduce memory operations and data transfer, especially on GPU devices, which may lead to performance improvements. In the new code, controling the proportion of clients participating in each round by the client_fraction parameter, rather than having all clients trained. This can reduce the consumption of computational resources and may improve the generality of the model. The experimental results on the FMNIST dataset are shown in Table 7. On the FMNIST data set, the mean value of the new model loss value was 0.0133. On the FMNIST data set, the proportion of the correct samples predicted by the new model is (73.25%).

Table 8: Experimental results of the optimized algorithm.

Data set	FMNIST
Experimental result	Average loss: 0.0133, Accuracy: 7325/10000 (73.25%)

4.3 System Stability and Reliability

1. Exception handling:

To optimize the code and add exception handling, make changes to ensure that the code can handle exceptions when it encounters problems and provide useful feedback. Capture exceptions that can occur when the data loads, such as network problems that can occur when downloading the MNIST dataset. Capture errors that may occur during model training, such as gradient explosion or gradient vanishing. Ensure that device-related abnormalities are captured when using devices (e.g., Compute Unified Device Architecture (CUDA)). When the aggregate model updates, ensure that the update is valid and that there are no numerical issues. When testing the model, ensure the data loads correctly and the evaluation process is not an error.

In the modified code, several tries...except blocks were added to capture and process exceptions that may occur. This way if the code encounters any problems during execution it will print out an error message and terminate or skip the error part instead of causing the entire program to crash. In addition, KeyboardInterrupt abnormalities were captured so that the training process could be interrupted by pressing Ctrl + C.

2. Random seeds:

The new code adds the following parts to set the random seed rather than the code without the random seed: A function to set random seeds is defined, set _ random _ places, which accepts a parameter seed_value with a default value of 42. The function of this function is to ensure that the random number generators of NumPy and PyTorch use the same seeds such that the same is generated for each run of the code, guaranteeing the reproducibility of the experiment. If using CUDA, it also sets random seeds for all CUDA devices. In the main function, the set_random_seeds function is called.

In this way, the random seed is set up before the model is trained.

Except for the above, the rest of the code should be the same as the code without a random seed. The random seed is mainly set to ensure that each time, the initialized weight, the segmentation of the data set, and other operations relying on the random number generator can get the same results, thus making the experimental results reproducible. The

experimental results on the FMNIST dataset are shown in Table 8. On the FMNIST data set, the mean value of the new model loss value was 0.0183. On the FMNIST data set, the number of samples predicted by the new model was (79.74%).

Table 9: Experimental results of the optimized algorithm.

Data set	FMNIST
Experimental result	Average loss: 0.0183, Accuracy: 7974/10000 (79.74%)

3. Model complexity:

Two hidden layers and the ReLU activation functions were added. The first hidden layer has 512 neurons, and the second hidden layer has 256 neurons. Use the ReLU as the activation function. The last layer is a linear layer, used for classification. The main difference between the old and new codes lies in the complexity of the neural network model used. In the new code, a more complex fully connected neural network model, ComplexModel, includes three fully connected layers and the ReLU activation function. While in the old code, a simple fully connected neural network model, SimpleModel, is used, which only contains a fully connected layer with no activation function.

Here are the specific differences:

1. Different class definitions of the neural network model.

ComplexModel Contains three fully connected layers (fc 1, fc 2, fc 3) and the ReLU activation function.SimpleModel It contains only one fully connected layer (fc).

2. In the main function main (), the classes of initialized global and local models are different:

In the new code, use the ComplexModel.In the old code, use the SimpleModel.

Other parts of the code, including data loading, client-side model training, model aggregation, and testing, remain unchanged. This means that the two codes implement the same federated learning framework, but the complexity of the neural network model trained on the client side varies.

The experimental results are shown in Table 10. On the FMNIST data set, the mean value of the new model loss value was 0.0182. On the FMNIST data set, the proportion of correct sample numbers by the new model is (80.04%).

Table 10: Experimental results of the optimized algorithm.

Data set	FMNIST
Experimental result	Test set: Average loss: 0.0202, Accuracy: 7806/10000 (78.06%) Test set: Average loss: 0.0182, Accuracy: 8004/10000 (80.04%)

Through experiments, after optimizing the code, it is sometimes found that theoretically, the accuracy of the code will increase, but in fact, the accuracy does not change significantly, even decreasing rather than rising. The following is the reason analysis:

1. Learning rate: If the learning rate is not set appropriately, the model may fail to converge to the best solution. Too high a learning rate may cause the model to oscillate near the optimal solution, while too low a learning rate may lead to too slow convergence.

2. Regularized Strength: In the code, the alpha parameter controls the intensity of the regularization term. If this parameter is set too high, it may lead to over-penalizing the model weights, thus hindering the learning process. If too low, it may not adequately prevent overfitting.

3. Client data distribution: If the data distribution between clients is uneven (i.e., there is a Non-IID), the model may perform better on some clients and perform poorly on other clients, which will affect the accuracy of the global model.

4. Model update aggregation: Aggregation methods may affect the performance of the global model. For example, simple averaging may not be the optimal aggregation method, especially in cases where the client data is evenly distributed.

5. Number of training rounds: If the number of training rounds is too small, the model may not have enough time to learn the characteristics of the data.

6. Batch size: Batch size affects the accuracy of gradient estimation and the stability of the training process. Batch processing too small can cause too much noise, while batch processing too large can cause memory problems or slow training.

7. Code change: Any code change can cause unexpected side effects. For example, if data loading or preprocessing steps are optimized, data leakage

may have been accidentally introduced or data distribution changed.

8. Randchasticity (Randomness): Due to the randomness of initialization weights, data shuffling and other factors, multiple runs may get different results even under the same setting.

Therefore, in practical application, the optimization of one model should take into account the influence of one parameter on another parameter. When optimizing the model, it is best to adjust multiple parameters at the same time, and the experimental effect will be better than adjusting only a single parameter.

5 CONCLUSION

It is worth noting that this study has some limitations. This experiment only constructed a simple algorithm model, which is only used as a code repetition of FedDyn algorithm theory. However, in practice, the FedDyn algorithm model is much more complex, and more factors and challenges need to be considered. For example, issues such as data distribution, communication delays between clients, privacy protection, and security all need to be fully considered. Moreover, to improve the performance and generalization capabilities of the models, more complex neural network structures, optimization algorithms, and regularization techniques may be required to be used. Alternatively, the datasets in practical application may be larger and more complex than the datasets are used in this study. This requires more computational resources and more efficient algorithm design to handle large-scale datasets. In addition, the FedDyn algorithm may also need to be integrated with other techniques and methods in practical applications. For example, problems such as model aggregation, client selection, and task scheduling in federated learning all need to be considered comprehensively.

Future studies could further explore the following aspects:

1. Algorithm optimization: In practical application, the performance and efficiency of the FedDyn algorithm can be further optimized. For example, researchers can explore more efficient model aggregation methods, client-side selection strategies, and task scheduling algorithms to improve the convergence speed and accuracy of the model.

2. Privacy protection: Privacy protection in federal learning is an important issue. Future studies could explore how to train and update models while protecting user privacy effectively. This may involve further research in areas such as differential privacy technology, encryption computing, and secure multi-party computing.

In conclusion, future studies can further explore and improve various aspects of the FedDyn algorithm to improve its performance, privacy protection, and scalability and apply it to a wider range of practical scenarios. This will bring greater development and innovation to the field of federated learning and distributed machine learning.

REFERENCES

X. Zhao, Z. Mao, H. Li, et al. (2023, March). “Big data security risk control model based on federated learning algorithm”. In Second International Conference on Green Communication, Network, and Internet of Things, Vol. 12586, (CNIoT, 2022), pp. 178-183.

R. Mónica, V. Haris. Pattern Recognition 148: 110122, (2024).

X. Li, K. Huang, W. Yang, et al. arXiv preprint arXiv: 1907. 02189, (2019).

L. Yiyang. Donghua University, (2022).

Z. Wenbo. Applied Sciences 13.9, (2023).

D. A. E. Acar, Y. Zhao, R. M. Navarro, et al. arXiv preprint arXiv:2111.04263, (2021).

L. Bo. Huazhong University of Science and Technology, (2020).

G. Wirth, P. A. Alves, R. D. Silva. Fluctuation and Noise Letters, 2350027, (2023).

M. Tivnan, G. J. Gang, W. Wang, et al. Journal of Medical Imaging, 10(3), 033501-033501, (2023).

L. Chenyang. Computer Science, 49(09): 183-193, (2022).