Going Forward-Forward in Distributed Deep Learning

Ege Aktemur[1,2*], Ege Zorlutuna[1,3], Kaan Bilgili[1,4], Tacettin Emre Bök[1,4], Berrin Yanikoglu[1], and Süha Orhun Mutluergil[1]

[1] Sabanci University, Istanbul, Turkey
{berrin,suha.mutluergil}@sabanciuniv.edu
[2] TU Darmstadt, Darmstadt, Germany
ege.aktemur@stud.tu-darmstadt.de
[3] ETH Zurich, Zurich, Switzerland
ezorlutuna@ethz.ch
[4] TU Berlin, Berlin, Germany
{bilgili,boek}@tu-berlin.de

Abstract. We introduce a new approach in distributed learning, building on Hinton's Forward-Forward (FF) algorithm to speed up the training of neural networks in distributed environments without losing accuracy. Unlike traditional methods that rely on forward and backward passes, the FF algorithm employs a dual forward pass strategy, eliminating the dependency among layers required during the backpropagation period, which prevents efficient parallelization of the training process. Although the original FF algorithm focused on its ability to match the performance of the backpropagation algorithm, this work aims to reduce the training time with pipeline parallelism. We propose three novel pipelined FF algorithms that speed up training 3.75 times on the MNIST dataset while maintaining accuracy when training a four-layer network with four compute nodes. These results show that FF is highly parallelizable and its potential in large-scale distributed/federated systems to enable faster training for larger and more complex models.

Keywords: Forward-Forward Algorithm · Distributed Learning · Pipeline Parallelism

1 Introduction

Training deep neural networks, often consisting of hundreds of layers, is a very time-consuming process that can take several weeks, using the well-known backpropagation [18] as the learning algorithm. However, parallelization of this algorithm presents significant challenges due to the sequential nature of the backpropagation learning algorithm, as illustrated in Figure 1.

* Corresponding author: ege.aktemur@stud.tu-darmstadt.de

During the backward pass in backpropagation-based training methods, the gradient is calculated for all layers, and they depend on the gradient calculated at the next layer. If layers are distributed to different nodes , this dependency necessitates the sequentialization of the computing nodes, as each node must wait for the gradient information to backpropagate from its successor before it can proceed with its calculations. Moreover, the need for constant communication between nodes to transfer gradient and weight information can lead to significant communication overhead. This is particularly problematic in large-scale neural networks, where the volume of data to be transferred can be substantial.

Distributed deep learning has witnessed significant advancements in recent years, driven by the ever-increasing complexity and size of neural networks. Distributed training frameworks such as PyTorch Distributed Data Parallel (DDP) [10], GPipe [6], PipeDream [13] and Flower [2] have emerged as pivotal solutions, enabling the training of massive models by optimizing for scale, speed, cost, and usability. These systems utilize sophisticated methodologies such as data, pipeline, and model parallelism to manage and process large-scale neural network training efficiently across multiple computing nodes.

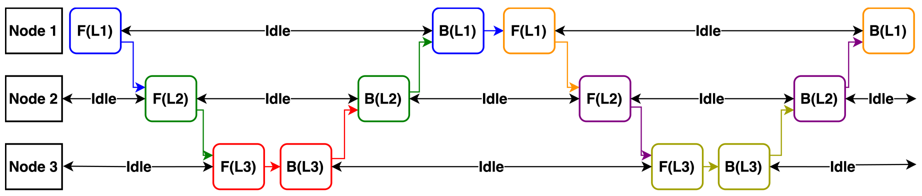


Fig. 1: Challenges in parallelizing traditional backpropagation involve managing the forward (F) and backward (B) passes, both applied to the layer specified by their first parameter.

In addition to the above research geared towards distributed implementations of backpropagation, the Forward-Forward Algorithm (FFA) proposed by Hinton [5] proposes a novel approach to train neural networks. Unlike traditional deep learning algorithms that rely on global forward and backward passes, the FFA only uses local (layer-wise) forward and backward passes. Although the FFA is designed to address the limitations associated with backpropagation without focusing on distribution, the layer-wise training feature of the FFA results in a less dependent architecture in a distributed environment that reduces idle time, communication and synchronization, as demonstrated in Figure 2.

This paper explores the integration of the FFA into distributed deep learning systems, particularly focusing on its potential to improve the efficiency and performance of training large-scale neural networks. We demonstrate the effectiveness of the Pipeline Forward-Forward (PFF) algorithm, which reduces the training duration by almost 5-fold for a 5-layer network while preserving the accuracy of the original algorithm.

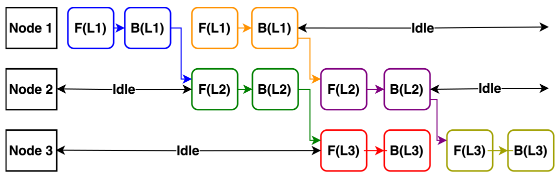


Fig. 2: Distributed training of a FFA network. F and B blocks are representing the layer local forward and backward passes. B blocks compute the gradient by considering the goodness that will be defined later and then update the weights of the corresponding layer.

2 Related Work

Distributed and parallel neural networks have attracted widespread interest and investigation from researchers, driven by the promise of improved computational efficiency and scalability in handling large-scale data and complex models. Comprehensive surveys from 2019 [1] and 2022 [12] provide extensive information about the recent developments on the parallelism methods including data, model and pipeline that we employ in our methods.

PipeDream [13] is a deep neural network training system that uses backpropagation that efficiently combines data, pipeline, and model parallelism by dividing the model layers into multiple stages containing consecutive layers. Unlike our method, PipeDream inherits the limitations of the classic backpropagation algorithm. Similarly, GPipe[6] is a distributed deep learning library that updates weights synchronously during the backward pass, which differentiates it from PipeDream; however, it also suffers from additional dependencies introduced by the classic backpropagation algorithm. In contrast, PyTorch Distributed Data Parallel [10] achieves near-linear scalability in distributed training by leveraging data parallelism and synchronizing gradients during the backward pass, trying to alleviate the inherent limitations of classic backpropagation.

In addition to the above methods, there are several proposed methods for efficient deep learning training that attempt to overcome the limitations of backpropagation by proposing alternatives to backpropagation. One of them is called *local parallelism* [9]. This method divides the layers into a number of blocks and uses this block's local loss to adjust the parameters of the layers that are within that block. Although this method achieves a significant speedup compared to backpropagation, its effectiveness is limited because the local loss of the blocks is not comparable to the global loss.

Forward-Forward Algorithm (FFA) [5] is a learning method for multilayer networks, proposed by Hinton. This algorithm introduces a novel way of training, which eliminates the backpropagation and the backward pass. Unlike the method proposed in this paper, the original FFA is sequential and is designed to work on a single processing unit. This algorithm forms the backbone of our method along with the split method, which aims to reduce communication between workers and increase parallelism.

FFA has significantly influenced neural network research, leading to innovative integrations and modifications. For example, Ororbia et al. [14] combined FFA with predictive coding, while Scodellaro et al. [19] demonstrated that CNNs trained using FFA can achieve up to 99.16% accuracy, approaching the performance of backpropagation. In parallel, Zhao et al. [20] developed CaFo, an algorithm that constructs cascaded neural blocks with independently trainable layers suitable for parallelization, and [17] proposed a variant of FFA Contrastive Learning that replaces later training stages with dual local updates employing separate loss functions.

Additional studies have extended FFA for various architectures and training paradigms. Gautham et al. [4] investigated the application of FFA in convolutional and recurrent models for multiclass classification and regression tasks, while Hwang et al. [7] introduced a Local Back-Propagation (LBP) algorithm that leverages standard unsupervised models, such as autoencoders, for independent layerwise training in distributed environments. Furthermore, Krutsylo [8] proposed a scalable hybrid blockwise approach that integrates backpropagation within blocks for modern architectures such as ResNet18 and MobileNetV3, and Papachristodoulou et al. [15] addressed FFA's limitations by introducing a novel convolutional design with channel-wise competitive learning via CFSE blocks, which achieved a testing error as low as 0.58% on MNIST.

Distributed Forward-Forward (DFF) [3] is an example of the use of FFA in distributed training system. DFF aims to create a decentralized and distributed training system to allow multiple low-performance devices to train a large model. DFF creates a cluster with one master node and multiple server nodes. Each server is assigned a layer(s) by the master node. These server nodes train their assigned layers using FFA and send their output to other layers.

Park et al. [16] propose FedFwd, a federated learning algorithm that replaces traditional backpropagation with FFA to perform layer-wise updates, thus reducing demands on local clients. The authors aim to tackle the inherent resource constraints in distributed training by eliminating the need to store all intermediate activations during training. Experimental results on datasets such as MNIST and CIFAR-10 show that FedFwd achieves competitive test accuracies and comparable training times. These findings suggest that FFA-based approaches offer a promising alternative for efficient distributed training.

While PipeDream and GPipe aim to parallelize backpropagation, DFF [3] is closest to our work, as it also tries to integrate the FFA into a distributed setting. However, in terms of accuracy, our implementation achieves better results than DFF because our implementation updates the weights more frequently, using minibatches, generates the negative samples adaptively, and uses an additional classifier which all contribute to the performance of the network.

3 Forward-Forward Algorithm

The FFA, proposed by Geoffrey Hinton [5], presents a novel approach to train neural networks that does not involve backpropagation, which is the standard learning algorithm used in training neural networks.

Inspired by the biological processes of the brain, the FFA trains layers individually and sequentially, utilizing two forward passes and a local backward pass to update the layer weights. Those passes are called the positive pass and the negative pass. The positive pass adjusts the weights to increase the "goodness" of the positive or real data, while the negative pass does the opposite for negative data, as defined below.

Negative Data. Negative data are obtained by giving incorrect labels to the samples provided as positive data. In [5], the MNIST dataset is used for the evaluation of the FFA. The existence of black borders in this image data set enabled encoding labels in the top-left corner of the input images (a 10-pixel area on the border denotes the label where a 1-of-C encoding is used). A positive sample is generated by adding the correct label of the digit in the image, whereas a negative sample is produced by encoding an incorrect digit as the label. Other alternatives for creating negative data, including unsupervised methods, are also possible and explored in [5].

Goodness Function. The goodness of a layer explored in [5] is the sum of the squares of the activities of the rectified linear neurons in that layer, and the aim during the learning period is to raise it above some threshold value for positive (real) data and below the threshold for negative data. The goodness function p can be captured by applying the logistic function σ to the sum of squared activities y_j^2 minus a threshold θ:

$$p(data) = \sigma(\sum_j y_j^2 - \theta) \tag{1}$$

Prediction. Two methods have been proposed to predict the label of an input image. In the *Goodness* prediction, the input is run through the network with all possible labels encoded in 1-of-C format so that the label vector consists of all 0s except a single 1 placed in the index representing this class. Then, the activations from all but the first hidden layer of the network are accumulated and the label yielding the maximum goodness is selected as the predicted class.

In the *Softmax* prediction, the raw input image is labeled with a neutral vector consisting of 0.1 values for each class and the activity vectors collected from all except the first hidden layer are fed into a Softmax layer which is trained to predict the class. The softmax layer is trained using backpropagation at the end of the training or during the training. As a remark, the softmax layer is a separate single-layer network, and it does not perform backpropagation to the FF layers. Hence, it can also be trained independently like an FF layer. This prediction approach is faster for inference since it is sufficient to perform a single pass for each datum; on the other hand, it is sub-optimal with respect to the Goodness prediction which better corresponds to how the network was trained.

Sequential Training of the Forward-Forward Algorithm. The layers of the network are trained for several epochs where an epoch is defined as training the network for the entire training dataset once. Since FF does not involve backpropagation, layers can be trained independently, each layer can be trained for more than one epoch waiting the upper layers to finish the previous epochs. For instance, assuming that the network will be trained for 60 epochs, we can train the first layer for 30 epochs, and use this half-trained layer-1 for training the second layer for its 30 epochs and so on.

Moreover, training a layer for the total number of epochs and providing its output to the next layer is not good for accuracy. Training each layer for a small number of epochs allows fine-grained adjustments on this layer and yields better results. So, we divide the training process of each layer into splits, basically the number of consecutive epochs passed from a layer before giving its output to the next layer. Figure 3 shows the difference between the 1 and 100 split values.

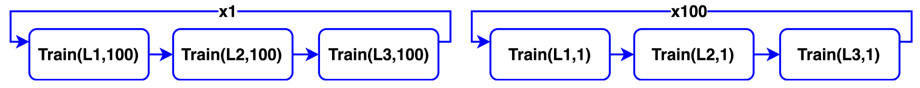


Fig. 3: Comparison of the FFA with split counts 1 (left) and 100 (right) across 3 layers. $Train(L, E)$ in a block denotes training the layer L for E epochs before passing its output to the next layer.

4 PipelineFF: A Pipeline Forward-Forward Algorithm

In this section, we propose three parallel Pipeline Forward Forward (PFF) algorithms: Single-Layer PFF (Section 4.1), All-Layers PFF (Section 4.2), Federated PFF (Section 4.3) and Performance-Optimized Network PFF (Section 4.4).

The Single-Layer PFF employs an architecture in which each node is dedicated to training only one layer, while All-Layers and Federated PFF algorithms distribute the training load more evenly, by asking every node to train all network layers in turn. All models enjoy model and pipeline parallelism, while the federated PFF also takes advantage of data parallelism.

In all proposed PFF models, we divide the total number of E epochs into S splits to facilitate distributed training. The sequential training periods of layers are called chapters, and each chapter takes $C = E/S$ training epochs. Consequently, each node has to process S chapters to complete the training for E epochs. The variable N denotes the number of nodes in the distributed system. The input data set is abstracted as x, which consists of positive and negative samples. $L_i(x)$ is used to denote the output of the i^{th} layer of the neural network after a forward pass with the input data x. Finally, $Train(L_i, n)$ is an abstraction for training the layer L_i for n epochs.

4.1 Single-Layer PFF Algorithm

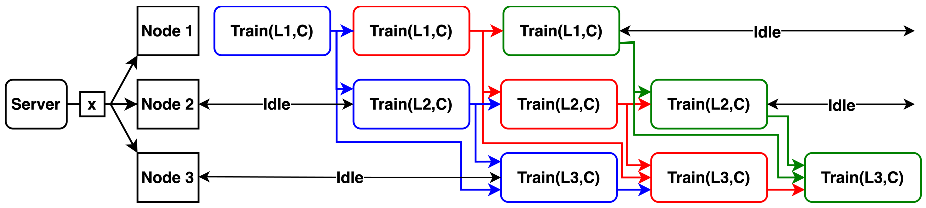


Fig. 4: Single-Layer PFF Algorithm example with three layers and three splits.

Single-Layer PFF employs an architecture in which each node is dedicated to training only one layer, as shown in Figure 4. This procedure ensures that while node i is training the layer L_i for the j^{th} split, the previous node can train the layer L_{i-1} concurrently on the split number $j' > j$. Already with this simple architecture, we achieve parallelism among the nodes, unlike the sequential FFA.

The distributed training example provided in Figure 4 illustrates a scenario with three compute nodes and a three-layer neural network. The number of nodes, N, is selected to be equal to the number of layers L and each node trains the corresponding layer of the network. In the following, we detail the algorithm with a pseudo code and the training process.

Node Initialization: Each node $i \in [1, N]$ is initialized with training data and layer parameters and hyperparameters.

Initialization and Training of Node 1: Node 1 initializes neurons of L_1 randomly and trains this layer for C epochs. Then, it sends L_1's weight and biases to Node $i' > 1$ and starts training L_1 for C more epochs and repeats.

Other Nodes: To train L_i for the j^{th} split, Node i needs to prepare the output of L_{i-1} (trained for $j' > j$ splits). This is done by getting the layers L_k where $k < i$ and doing a forward pass until L_i. Then, it trains L_i for C epochs and sends its weight and bias values to each Node $i' > i$ and repeats.

Termination: Each node iterates for S chapters and E epochs in total. After each iteration, a node might update the negative examples based on a protocol.

Transferring large layer outputs, which grow with dataset size, between computing nodes leads to a communication bottleneck. To enhance time efficiency, our approach prioritizes the transmission of layer parameters over output tensors. By sending the parameters and executing the forward pass on each node, we drastically reduce the amount of data transferred, resulting in a faster overall process. This optimization is represented in Figure 4 and the loop within Algorithm 1, Lines 3-5.

Algorithm 1 Single-Layer Training Procedure

```
1: for $chapter = 1$ to $split$ do
2:     $x_{pos}, x_{neg} \leftarrow X_{POS}, X_{NEG}$
3:     for $layerIndex = 1$ to $currentClientIndex$ do
4:         $layer \leftarrow getLayer(layerIndex, chapter)$
5:         $x_{pos}, x_{neg} \leftarrow layer(x_{pos}, x_{neg})$
6:     for $miniEpoch = 1$ to $epochs/split$ do
7:         $learningRateCooldown(chapter, miniEpoch)$
8:         for each $batch$ in $x_{pos}, x_{neg}$ do
9:             $trainLayer(currentClientIndex, batch)$
10:    $PublishLayer(chapter, layerIndex)$
11:    $UpdateXNEG(publish = False)$
```

4.2 All-Layers PFF Algorithm

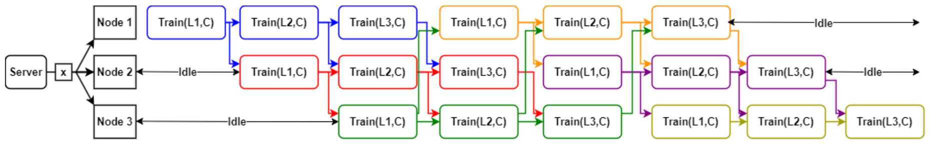


Fig. 5: All-Layers PFF Algorithm example with three layers and two splits.

The All-Layers PFF model builds upon the Single-Layer approach by allowing each compute node to train the entire network, affording a more balanced load among the nodes.

Figure 5 illustrates the approach in the context of a system with three compute nodes and a three-layer neural network, but with two splits for clarity.

Node Initialization: Each node is initialized with the dataset and the required hyperparameters.

Initial Training: $Node_1$ trains the entire network with randomly initialized layers for C epochs.

Rest of the training: Concurrently, each $Node_i$ trains the layer L_{k-1} immediately after $Node_{i-1}$ has completed its C epochs of training for the layer L_{k-2}. Each $Node_i$ sends its last trained layer to $Node_{i+1(mod(N))}$ and receives the layer to be trained from $Node_{i-1(mod(N))}$.

Iteration and Convergence: The process repeats for S/N splits but not S since in this approach every node trains each layer. The algorithm converges when the entire network has been trained for a total of E epochs.

4.3 Federated PFF Algorithm

In contrast to the All-Layers PFF Algorithm, the Federated PFF Algorithm employs the same training procedure but differentiates in the data it utilizes

Algorithm 2 All-Layers Training Procedure

```
1:  for $chapter = 1$ to $split$ do
2:      $x_{pos}, x_{neg} \leftarrow X_{POS}, X_{NEG}$
3:      for $layerIndex = 1$ to $numLayers$ do
4:          if not $(firstClient$ and $chapter = 0)$ then
5:              $getLayer(layerIndex, chapter)$
6:          for $miniEpoch = 1$ to $epochs/split$ do
7:              $learningRateCooldown(chapter, miniEpoch)$
8:              for each $batch$ in $x_{pos}, x_{neg}$ do
9:                  $trainLayer(layerIndex, batch)$
10:         $PublishLayer(chapter, layerIndex)$
11:         $x_{pos}, x_{neg} \leftarrow layer(x_{pos}, x_{neg})$
12:     $UpdateXNEG(publish = False)$
```

for training. Each node trains on its own local dataset rather than a shared one. This preserves data privacy, as there is no need to share raw data with other nodes or a centralized server. While the data remain localized, the nodes exchange model updates. This allows nodes to benefit from learning carried out by their counterparts, which iteratively improves the global model that benefits from diverse data without central aggregation. This federated structure is crucial not only for scenarios requiring data privacy but also enables distributed training over networks where central data aggregation is impractical or undesired.

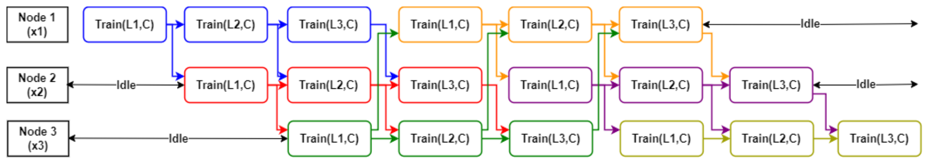
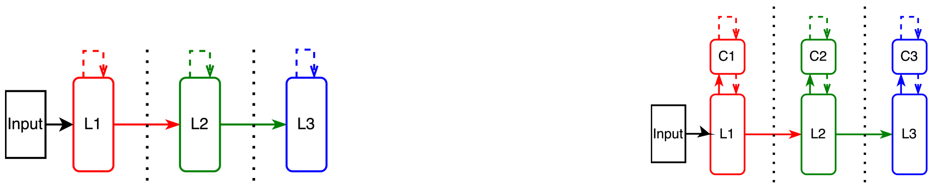


Fig. 6: Federated PFF Algorithm example with 3 layers and 6 split.

4.4 PFF with A New Goodness Function

The FFA [5] states that different goodness functions can be used while training each layer individually. In this section, we propose a new PFF algorithm called Performance-Optimized Network (PON) that uses a goodness function based on classification accuracy. Specifically, we add a softmax activated layer to each network layer and update only the weights of these two layers via backpropagation (as illustrated in Figure 7b), training each layer to maximize classification accuracy and thereby making this approach suitable only for supervised learning problems. Furthermore, the rest of the training is performed exactly in accordance with the FFA except that there are no negative data.



(a) Original FFA (b) PON

Fig. 7: Comparison of forward and backward passes between FFA and PON.

5 Experimental Evaluation

We obtain and compare several models by modifying the distributed model (Single-Layer or All-Layers), negative data selection (Adaptive, Fixed, or Random), classification strategies (Goodness and Softmax) and goodness function.

The two variants of the PFF (Single-Layer and All-Layers models, as explained in Sections 4.1 and 4.2, respectively) are compared with the sequential implementation (node count $N = 1$), which is equivalent to the original FFA and can be used to compare models using the same code base.

To select negative samples, we implemented AdaptiveNEG, the method proposed in [5], which selects the most predicted incorrect label as the negative label. The second approach is FixedNEG, which involves choosing random incorrect labels for each instance in the beginning of the training. The third approach, called RandomNEG, selects the random incorrect labels at the end of each chapter. The last two are proposed to speed up the training.

For the classification approach, we implemented both approaches explained in Section 3 proposed by Hinton in [5]. In the Goodness based approach, a sample is classified into the class that results in the highest goodness score (considering all but the first layer). In the Softmax approach, a single layer classification head is added to the network with softmax activation.

Finally, we evaluated the PON proposed in 4.4 in supervised learning scenarios.

5.1 Implementation Details

Our chosen architecture is the same feedforward neural network as in [5]. Its configuration is [784, 2000, 2000, 2000, 2000], where the numbers indicate the number of nodes in each layer. Thus, the input, which is 784-dimensional, is followed by four hidden layers with 2000 nodes each. Each of these layers employs the ReLU activation function.

The network is trained and tested using the MNIST dataset, using 60,000 training instances with mini-batches of size 64 and testing on a separate test set comprising 10,000 instances for 100 epochs and 100 splits. The Adam optimizer is used for both the FF Layers and the Softmax layer (trained using Backpropagation) is set initially at 0.01 and 0.0001 respectively and cooldowns after the 50th epoch. The threshold coefficient θ in Eq. 1 is set to 0.01, as in [5].

5.2 Comparison of FF, DFF and PFF Models

We first compare the accuracy and speed of the PFF variants using the Goodness approach with the Matlab implementation of the original Forward-Forward paper's results [5] and another distributed implementation [3] of the Forward-Forward algorithm. The results are given in Table 1.

The AdaptiveNEG approach, which selects negative data based on the network's performance at each chapter, outperforms both RandomNEG and FixedNEG models in terms of accuracy. Remarkably, despite shorter training times (7,178 vs 11,190 seconds), RandomNEG performs close to AdaptiveNEG in terms of accuracy (98.33 vs 98.52 in Sequential), making these models more accurate than FixedNEG in every implementation.

The AdaptiveNEG Goodness model with Sequential implementation is basically the same as [5] and it indeed achieves almost the same accuracy as Hinton's Matlab implementation [11]. It enables us to compare the training time of the model with our Python implementation.

Furthermore, AdaptiveNEG with All-Layers implementation matches the highest accuracy of its sequential version (98.51% vs. 98.52%) but in about a quarter of the time (2,980 vs. 11,190 seconds), using four compute nodes. This finding showcases the PFF Algorithm's efficiency in distributing the training, achieving 94% utilization (3.75/4), significantly speeding up the process. The All-Layers method speeds up training for AdaptiveNEG by efficiently distributing tasks between nodes, allowing each to compute its own negative samples after every chapter, unlike the Single-Layer approach, where the last node generates and publishes the generated labels.

Table 1: Original, DFF and PFF comparison. Baseline is highlighted as bold.

Model	**Setup**	**Training Time (s)**	**Test Accuracy (%)**
Hinton's Matlab Code [11]			98.53
DFF (1000 epochs)[3]			93.15
AdaptiveNEG-Goodness	Sequential	**11,190.72**	**98.52**
	Single-Layer	5,254.87	98.43
	All-Layers	2,980.76	98.51
RandomNEG-Goodness	Sequential	7,178.71	98.33
	Single-Layer	1,974.10	98.26
	All-Layers	2,008.25	98.17
FixedNEG-Goodness	Sequential	7,143.28	97.95
	Single-Layer	1,920.80	97.94
	All-Layers	1,978.21	97.89

5.3 Classifier Mode Comparison for AdaptiveNEG

In the first evaluation, we compared PFF variants that used the Goodness approach for classification and established the AdaptiveNEG Goodness model as the proposed baseline. The second experiment compares the performance of this model with that using the Softmax approach. As mentioned before, the goodness method is the first method provided in [5] and uses only the FF layers. The Softmax method on the other hand, is presented as a faster alternative, predicting the output using a Softmax layer that gets the activations of the hidden FF layers (except the first). The corresponding results can be found in Table 2.

The results show that the AdaptiveNEG Softmax model trains faster than the AdaptiveNEG-Goodness model, for all different implementations (e.g. 1,886 versus 2,980 secs). This accelerated training is due to the single-step prediction in the softmax approach, rather than predicting across 10 different classes. However, the accuracies all slightly decrease compared to comparable Goodness models.

The advantage of the All-Layers PFF Algorithm on the AdaptiveNEG explained in Section 5.2 is still very observable for the Softmax methods.

Table 2: Classification mode comparison for AdaptiveNEG. The first three row results indicated in italic denote the baseline model from Table 1. Bold results indicate suggested models for each classification mode.

Model	**Setup**	**Training Time (s)**	**Test Accuracy (%)**
AdaptiveNEG-Goodness	Sequential	11,190.72	98.52
	Single-Layer	5,254.87	98.43
	All-Layers	**2,980.76**	**98.51**
AdaptiveNEG-Softmax	Sequential	8,365.96	98.38
	Single-Layer	2,471.27	98.31
	All-Layers	**1,886.42**	**98.30**

5.4 Classifier Mode Comparison for RandomNEG

Based on the previous findings in section 5.2, we observed that RandomNEG closely rivals AdaptiveNEG in accuracy, while significantly reducing training time. This observation leads us to further investigate the impact of employing softmax within the RandomNEG framework. This section thus compares the effects of softmax and Goodness prediction on both FixedNEG and RandomNEG strategies. The results of this comparison are detailed in Table 3, providing insight into how these elements influence the effectiveness and efficiency of the methods under review.

In terms of accuracy, the Softmax approach outperforms the Goodness approach for prediction. For instance, while the Sequential RandomNEG Goodness model performed 98.33 the softmax version performed 98.48. This level of accuracy of the softmax version is very close to the performance of our baseline model

(98.51 Accuracy using AdaptiveNEG Goodness). Although the accuracy levels are very close, we observe that there is a significant speed-up using RandomNEG (8,104 versus 11,190).

In terms of training speed, on the other hand, training a model with the additional Softmax layer takes longer in the Sequential model as expected. However, Single-Layer implementation is not affected by this additional complexity. This is due to the fact that Softmax Layer is easier/faster to train than the FF Layers, and due to the pipeline architecture this only adds a small delay.

Surprisingly, Softmax models give faster results compared to same Goodness models for All-Layers implementation. This again stems from the fact that training Softmax is faster than training FF Layers, and distributing this job between 5 nodes speeds up the overall training.

Table 3: Classification mode comparison for RandomNEG. The first three row results indicated in italic are from Table 1. Bold text shows the proposed RandomNEG model balancing accuracy and speed.

Model	**Setup**	**Training Time (s)**	**Test Accuracy (%)**
RandomNEG-Goodness	Sequential	7,178.71	98.33
	Single-Layer	1,974.15	98.26
	All-Layers	2,008.25	98.17
RandomNEG-Softmax	Sequential	8,104.96	98.48
	Single-Layer	1,891.86	98.31
	All-Layers	**1,786.30**	**98.33**

5.5 Evaluation of the Performance-Optimized Network

This section explores the evaluation of our Performance-Optimized Network (PON) that is created by incorporating a new goodness function into the Forward-Forward algorithm. Our analysis shows that while the Performance-Optimized Model performs slightly worse than the best-performing AdaptiveNEG Goodness model, there is a notable reduction in training times, indicating a significant step towards efficient computing without significantly sacrificing performance.

In the MNIST dataset, the results presented in Table 4 highlight the effectiveness of the new goodness function proposed when integrated with the Forward-Forward algorithm. The highest test accuracy is achieved by the AdaptiveNEG Goodness model, reaching 98.52%, which is slightly superior to the 98.48% accuracy of the RandomNEG Softmax model. It should be noted that the Performance-Optimized Network, whether using only the last layer or all layers, shows a small decrease in accuracy. Specifically, using all layers slightly improves the accuracy to 98.38% compared to 98.30% when only the last layer is used. These results underscore the robustness of the proposed goodness function, particularly when all layers are optimized, despite a considerable reduction in training time (4219.97 seconds) compared to the AdaptiveNEG Goodness model (11190.72 seconds).

Table 4: MNIST evaluation of PON with the baseline models from Table 1.

Model	**Training Time (s)**	**Test Accuracy (%)**
AdaptiveNEG-Goodness	11,190.72	98.52
RandomNEG-Softmax	8,104.96	98.48
PON (only last layer)	4,219.97	98.30
PON (using all layers)	4,219.97	98.38

5.6 Experiments with CIFAR-10

To extend our analysis to a more complex scenario, we replicated the experiments described in Sections 5.2 through 5.5 using the CIFAR-10 dataset. CIFAR-10, known for its higher variability and complexity compared to MNIST, serves as a challenging benchmark to test the robustness and effectiveness of our models.

Table 5 presents the detailed results of these experiments. Performance-Optimized Network (PON), whether it uses only the last layer or all layers, demonstrated a superior ability to handle the complexities of CIFAR-10, with the prediction using all of the layers achieving the highest accuracy of 53.50%. This was closely followed by the RandomNEG Softmax model, which achieved 52.18% accuracy. Furthermore, the PON models show relatively efficient training durations compared to the other sequential PFF Models. Surprisingly, the AdaptiveNEG Goodness model, which performed exceptionally well in MNIST, significantly underperformed in CIFAR-10, with an accuracy of only 11.10%. This difference suggests a potential mismatch between the model's capabilities and the dataset's requirements.

It should be noted that while state-of-the-art models have achieved above 99% accuracy on CIFAR-10, the performance of our models is similar to Hinton's original FFA, which reported an accuracy of 56% using CNNs [5].

Table 5: CIFAR-10 Results

Model	**Training Time (s)**	**Test Accuracy (%)**
PON (using all layers)	**4,920.97**	**53.50**
PON (only last layer)	4,920.97	53.11
FixedNEG-Softmax	8,021.15	50.89
RandomNEG-Softmax	7,636.99	52.18
AdaptiveNEG-Goodness	10,148.23	11.10

6 Conclusion and Future Work

Our work in this paper presents Pipeline Forward-Forward Algorithm (PFF), a novel way to train distributed neural networks using Forward-Forward Algorithm. Compared to the classic implementations with backpropagation and pipeline parallelism [6] [13], PFF is inherently different as it does not enforce backpropagation dependency on the system, thus achieving higher utilization of computational units with fewer bubbles and idle time. Experiments done with PFF show that the PFF Algorithm achieves the same accuracy as the standard FF implementation [5] with a 4x speed up. Comparison of PFF with an existing distributed implementation of Forward-Forward (DFF [3]) shows even greater improvements, as PFF achieves %5 more accuracy in 10 times less epochs. This improvement in accuracy is mainly because PFF splits the data into batches and feeds them to the network batch by batch, unlike DFF which feeds the data as a whole. In addition, the data that are exchanged between layers in PFF is a lot less than DFF since PFF sends the layer information (weights and biases), whereas DFF sends the whole output of the data. This results in less communication overhead compared to DFF.

Besides the exciting results that PFF produced, we believe that our work paves the way for a brand new path in the area of Distributed training of Neural Networks. Thus, there are plenty of different ways that can improve PFF, with some of these approaches outlined below.

– **Exchanging parameters after each batch:** In the current implementation of PFF, the exchange of parameters between different layers takes place after each chapter. Making this exchange after each batch is worth trying since it could better tune the weights and produce higher accuracy. However, it could potentially increase the communication overhead.
– **Different Setups:** In the experiments of this work, we used sockets to establish communication between different nodes. This brings additional communication overhead, as the data is sent through the network. In a setup where computational units of the PFF are closer and can access to a shared resource (Multi GPU architectures) the time of training networks can decrease drastically.
– **Generating Negative Samples Differently:** The way negative samples are generated is a very important aspect of the Forward-Forward Algorithm as it directly affects the learning of the network. Thus, discovering new and better ways to generate negative samples would definitely result in a better performing system.
– **Forming an Innovative Framework:** A general and efficient framework can be implemented for training large neural networks following and improving the novel ideas presented in this work.

References

1. Ben-Nun, T., Hoefler, T.: Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. ACM Computing Surveys **52**(4), Article 65 (2019). https://doi.org/10.1145/3320060
2. Beutel, D.J., Topal, T., Mathur, A., Qiu, X., Fernandez-Marques, J., Gao, Y., Sani, L., Li, K.H., Parcollet, T., de Gusmao, P.P.B., Lane, N.D.: Flower: A friendly federated learning research framework (2022), https://arxiv.org/abs/2007.14390
3. Deng, Q., et al.: Dff: Distributed forward-forward algorithm for large-scale model in low-performance devices. In: 2023 IEEE 6th International Conference on Pattern Recognition and Artificial Intelligence (PRAI). IEEE (2023)
4. Gautham, S.R., Nair, S., Jamadagni, S., Khurana, M., Assadi, M.: Exploring the feasibility of forward forward algorithm in neural networks. In: 2024 International Conference on Advances in Modern Age Technologies for Health and Engineering Science (AMATHE). pp. 1–6 (2024). https://doi.org/10.1109/AMATHE61652.2024.10582053
5. Hinton, G.: The forward-forward algorithm: Some preliminary investigations. arXiv preprint arXiv:2212.13345 (2022)
6. Huang, Y., et al.: Gpipe: Efficient training of giant neural networks using pipeline parallelism. In: Advances in neural information processing systems. vol. 32 (2019)
7. Hwang, T., Seo, H., Jung, S.: Local back-propagation: Layer-wise unsupervised learning in forward-forward algorithms (12 2024). https://doi.org/10.21203/rs.3.rs-5695830/v1
8. Krutsylo, A.: Scalable forward-forward algorithm (2025), https://arxiv.org/abs/2501.03176
9. Laskin, M., et al.: Parallel training of deep networks with local updates (2021)
10. Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., Paszke, A., Smith, J., Vaughan, B., Damania, P., Chintala, S.: Pytorch distributed: Experiences on accelerating data parallel training (2020), https://arxiv.org/abs/2006.15704
11. Löwe, S., Baichuan: Forward-forward. https://github.com/loeweX/Forward-Forward (2023)
12. Mayer, R., et al.: Scalable deep learning on distributed infrastructures: Challenges, techniques, and tools. ACM Computing Surveys **53**(1), Article 3 (2020). https://doi.org/10.1145/3363554
13. Narayanan, D., Harlap, A., et al.: Pipedream: generalized pipeline parallelism for dnn training. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles. pp. 1–15. SOSP '19, New York, NY, USA (2019). https://doi.org/10.1145/3341301.3359646, https://doi.org/10.1145/3341301.3359646
14. Ororbia, A., Mali, A.: The predictive forward-forward algorithm (2023)
15. Papachristodoulou, A., Kyrkou, C., Timotheou, S., Theocharides, T.: Convolutional channel-wise competitive learning for the forward-forward algorithm. Proceedings of the AAAI Conference on Artificial Intelligence **38**(13), 14536–14544 (Mar 2024). https://doi.org/10.1609/aaai.v38i13.29369, https://ojs.aaai.org/index.php/AAAI/article/view/29369
16. Park, S., Shin, D., Chung, J., Lee, N.: Fedfwd: Federated learning without backpropagation (2023), https://arxiv.org/abs/2309.01150
17. R, G.: Improved forward-forward contrastive learning (2024), https://arxiv.org/abs/2405.03432

18. Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning representations by back-propagating errors. Nature **323**(6088), 533–536 (1986). https://doi.org/10.1038/323533a0
19. Scodellaro, R., et al.: Training convolutional neural networks with the forward-forward algorithm. arXiv preprint arXiv:2312.14924 (2023)
20. Zhao, G., et al.: The cascaded forward algorithm for neural network training (2023)