به نام خدا

دانشگاه تهران
پردیس دانشکده‌های فنی
دانشکده برق و کامپیوتر

بهینه سازی توزیع شده و یادگیری
Distributed Optimization and Learning

پروژه 2
Second Project

عرفان میرزایی
Erfan Mirzaei

810199289

دکتر کبریایی
Dr.Kebriaei

دی ماه 1400
December 2021

Contents

Abstract

In this project, we implemented distributed deep learning algorithms as one case study of distributed optimization algorithms both for the data-parallel and data-distributed cases.

We used the CIFAR10 dataset and a convolution neural network to accomplish this. Our first step in training the network was centralized, as is standard in deep learning. Then we implemented the GoSGD algorithm, one of the most famous algorithms in distributed deep learning. In the following, different weight matrices were tested. Then, we assumed that the communication between clients was noisy, and the result was compared with previous cases. Finally, we implemented another gossip-based distributed optimization algorithm, SGP, and compared the result with GoSGD.

In the second part, we consider the case in which the data is distributed between clients, but despite the federated learning, there is no coordinator. For this part, we used a modified version of the GoSGD Algorithm and distributed the data in an iid and uniform manner between clients. Finally, the results are compared with the first project.

Section 1) Problem Definition

In this project, we implemented distributed deep learning algorithms as one case study of distributed optimization algorithms both for the data-parallel and data-distributed cases.

The objective of training a neural network is to find a set of parameters that minimize the cost function over the training set. Usually, these cost functions are not convex, and Stochastic Gradient Descent is one of the most famous algorithms for optimizing this objective. However, to attain good generalization and good performance on the test dataset, you should have a sizeable centralized dataset, which is not the case for many applications. Furthermore, current CNN structures are extremely deep and contain many parameters. Those structures involve heavy gradient computation times, making the training on big datasets very slow.

Therefore, there is so much desire for distributed optimization methods to solve this problem. In the first part, we address the issue of speeding up the training of convolutional neural networks, and then we consider the case in which the data is distributed between clients, and there is no global client. The FedAvg algorithm[1] is one of the simple decentralized optimization algorithms which shows good performance, and as a part of this project, I implemented it.

The formulation of the problem can be derived in a distributed fashion by minimizing the following:

$$\sum_{i=1}^{M} \mathcal{L}(x_i) + \frac{\rho}{2}||x_i - \overline{x}||_2^2$$

with the x_i being the worker's local variables and x(-) the global consensus. We can rewrite this loss in order to exhibit gossip exchanges:

$$\sum_{i=1}^{M} \mathcal{L}(x_i) + \frac{\rho}{4M} \sum_{i}^{M} \sum_{j}^{M} ||x_i - x_j||_2^2$$

Finally, we consider the following equivalent function in our optimization problem introducing A, a random matrix:

$$\sum_{i=1}^{M} \mathbb{E}\left[\ell(x_i, y) + \sum_{j}^{M} a_{ij}||x_i - x_j||_2^2\right]$$

In our gossip method, the terms in the outer sum of the last equation are sampled concurrently by different workers. aij is a random variable controlling exchanges between workers i and j with p and E[aij] = row/4M

With the above formulation, we can address both problems stated above.

Section 2) Network Architecture

In this project, we want to use a convolutional neural network for the classification of images for comparing centralized and distributed versions of deep learning. For this purpose, we used the network in [1] for the cifar10 dataset.

The network has two convolutional layers, two max-pooling layers, and on top of that, there are three fully connected layers for classifying. The below tables show the architecture.

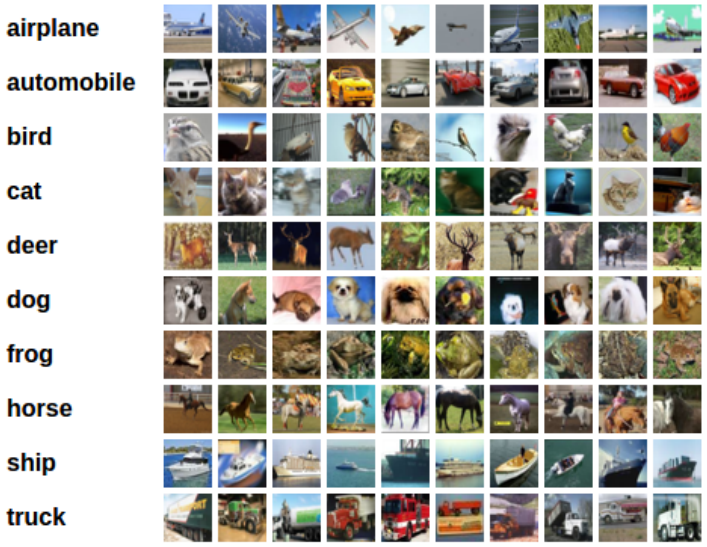
Operation Layer		# filters	Size of Each Filter	Stride value	Padding value
Layer 1	Convolution	64	5 * 5 * 3	1 * 1	-
	ReLU	-	-	-	-
	Max-Pooling[1]	1	2 * 2	2 * 2	-
Layer 2	Convolution	128	5 * 5 * 64	1 * 1	-
	ReLU	-	-	-	-
	Max-Pooling	1	2 * 2	2 * 2	-
Layer3	Convolution	64	3 * 3 * 128	1 * 1	-
	ReLU	-	-	-	-

Layer	Input Dimension	Output Dimension
FC	256	# Classes(10)

[1]

Section 3) Dataset

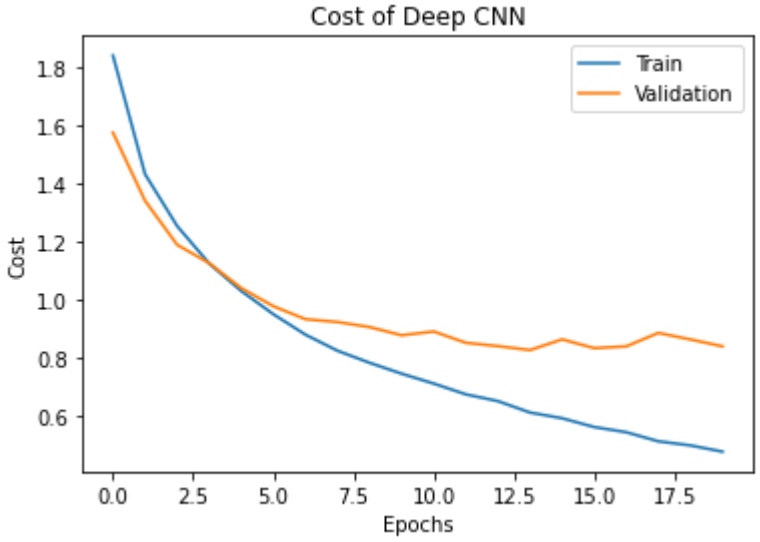
In this project, I used the cifar10 dataset [3]. It has the classes: 'airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck'. The images in CIFAR-10 are of size 3x32x32, i.e., 3-channel color images of 32x32 pixels in size.

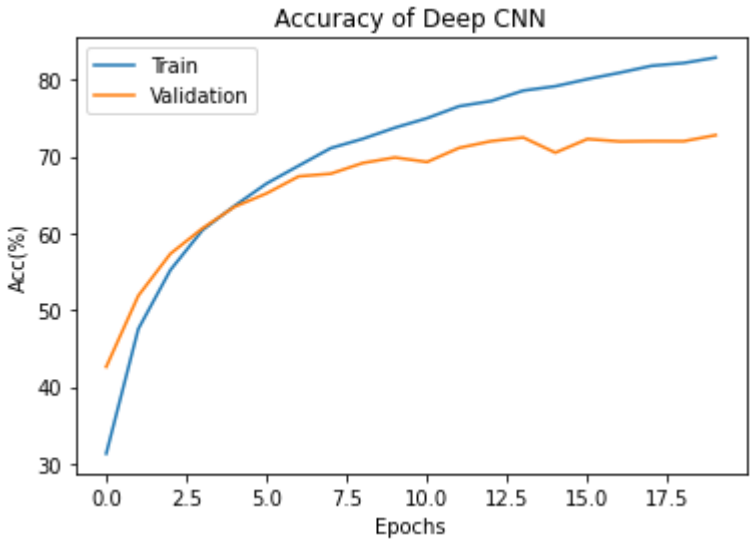


The cifar10 dataset includes 50,000 train images and 10,000 test images. The train images were distributed uniformly with respect to classes. In other words, each class has precisely 5,000 images in the training set. We center-cropped the images of the size 28 * 28 *3, and As before, we re-scale each pixel value to the interval [0,1].

Section 4) Centralized Optimization

In this section, we trained the architecture described in the previous sections on the cifar10 training set in a centralized manner. For training of the network, we passed data as batches with a size of 128, as in the paper[1]. We used the "Cross-Entropy" loss function and also the "SGD" with weight decay(1e-4) as an optimizer function with a learning rate of 0.01, and the network was trained for 20 epochs on the train and validation datasets. The results are as follows:





Section 4) Distributed Optimization with data-parallel

In this section, we assume all workers have all the data, and at each iteration, each worker loads a mini-batch of data from its train loader, and its local network is trained with this mini-batch. The model evaluated on the test set is called the test model. In the section, it is simply the averaging of all workers' model weights.

4.1. GoSGD [Original Paper]

The GoSGD algorithm considers M independent agents called workers. Each hosts a CNN of the same architecture with a set of weights noted x_i for worker i. They are all initialized with the same value. During training, all workers iteratively proceed in two steps; One consisting of local optimization with gradient descent and the other aiming at exchanging information in order to ensure a consensus between workers:

The Pseudo code of this algorithm is as follows:

Algorithm 1 GoSGD: workers Pseudo-code

Input: p: probability of exchange, M: number of threads, η : learning rate
Initialize: x is initialized randomly, $x_i = x$, $\alpha_i = \frac{1}{M}$
repeat
  PROCESSMESSAGES(msg$_i$)
  $x_i \leftarrow x_i - \eta^t v_i^t$
  if $S \sim B(p)$ **then**
    $j = Random(M)$
    PUSHMESSAGE(msg$_j$)
  end if
until Maximum iteration reached
return $\frac{1}{M}\sum_{m=1}^{M} x_m$

Algorithm 2 Gossip update functions

function PUSHMESSAGE(queue msg$_j$)
  $x_i \leftarrow x_i$
  $\alpha_i \leftarrow \frac{\alpha_i}{2}$
  msg$_j$.push((x_i, α_i))
end function
function PROCESSMESSAGES(queue msg$_i$)
  repeat
    $(x_j, \alpha_j) \leftarrow$ msg$_i$.pop()
    $x_i \leftarrow \frac{\alpha_j}{\alpha_i+\alpha_j} x_j + \frac{\alpha_i}{\alpha_i+\alpha_j} x_i$
    $\alpha_i \leftarrow \alpha_j + \alpha_i$
  until msg$_i$.empty()
end function

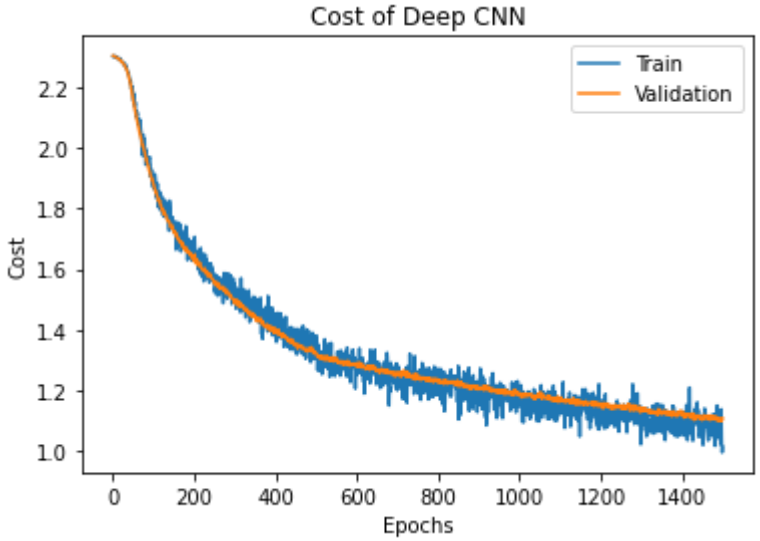
I used the same hyperparameters as in the original paper. Also, in this part, we use the same weight matrix of communication between workers. As we can find out from the pseudocode, the weight matrix is a time-varying stochastic matrix.

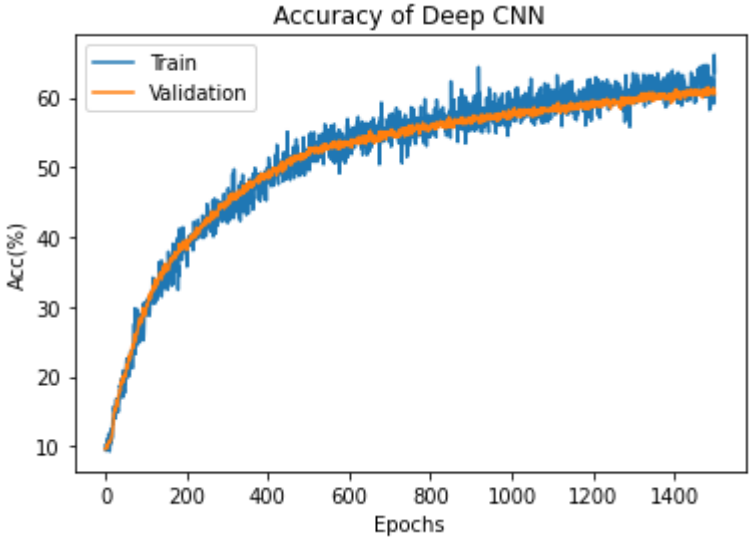
4.1.1. Implementation Results

We trained the previous architecture on the client dataset based on the GoSGD algorithm with the hyperparameters as follows:

During the training, the learning rate is constantly equal to 0.01, and the weight decay to 1e-4. All mini-batches contain 128 images. We use eight workers. The probability p controls the exchange frequency set to 1, as in the original paper.

The network was trained for 1500 iterations on the train and validation datasets. The results are as follows:





As shown in the plots, the performance in 1500 iterations is lower compared to the centralized version. The network's performance could be better if we trained for more iterations, but due to time-limit and for comparison, we trained all versions for 1500 rounds.

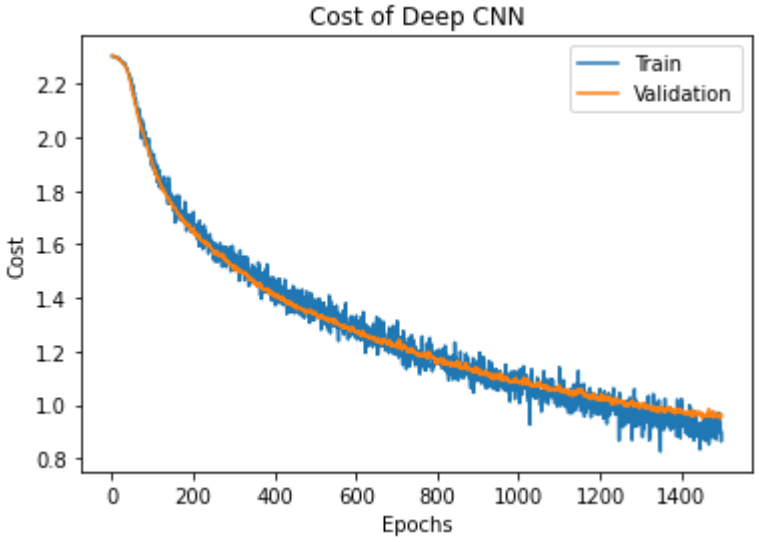
4.2. GoSGD with Doubly Stochastic Weight Matrix

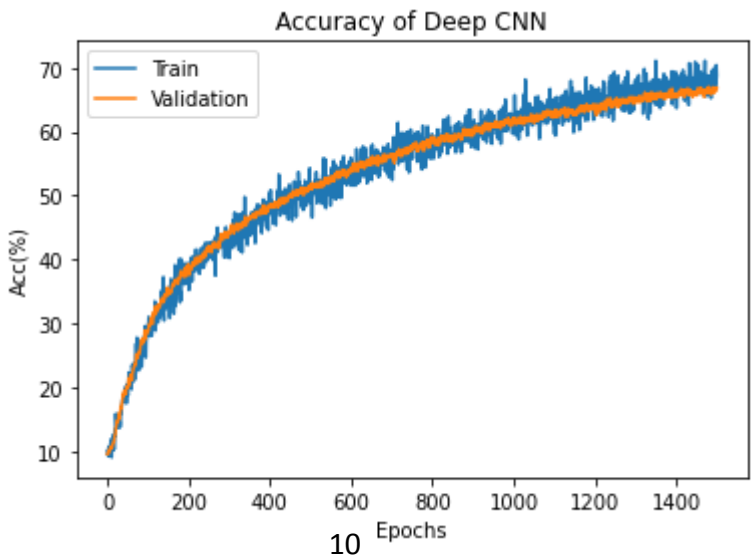
In this section, we just changed the weight matrix between workers. For this section, we consider a fixed doubly stochastic matrix. We used a uniform matrix with all elements equal to 1/M.

4.2.1. Implementation Results

We trained the previous architecture on the client dataset based on the GoSGD algorithm with the hyperparameters as follows.

During the training, the learning rate is constantly equal to 0.01, and the weight decay to 1e-4. All mini-batches contain 128 images. We use eight workers. The probability p controls the exchange frequency set to 1, as in the original paper. The network was trained for 1500 iterations on the train and validation datasets. The results are as follows:





As we can see in the plots, the performance in 1500 iterations is lower compared to the centralized version. However, it is better than the previous section. The performance of the network could be better if we trained for more iterations, but due to time-limit and for the sake of comparison, we trained all versions for 1500 rounds.

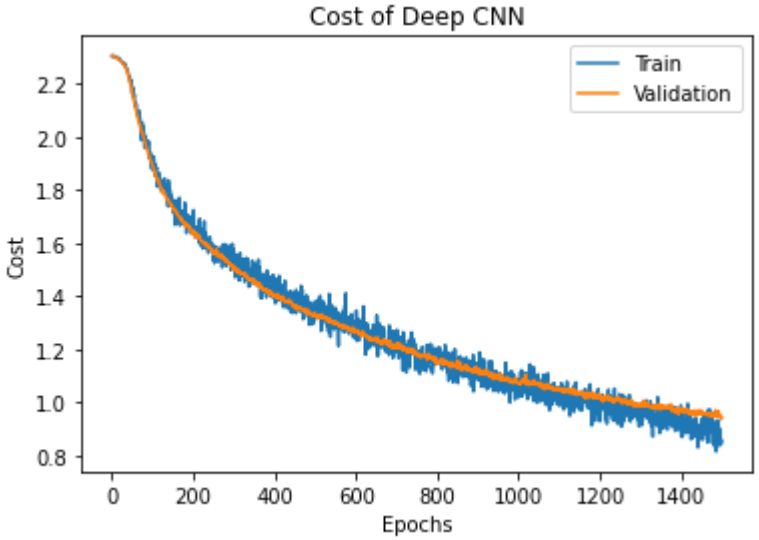
4.3. GoSGD with Stochastic Weight Matrix

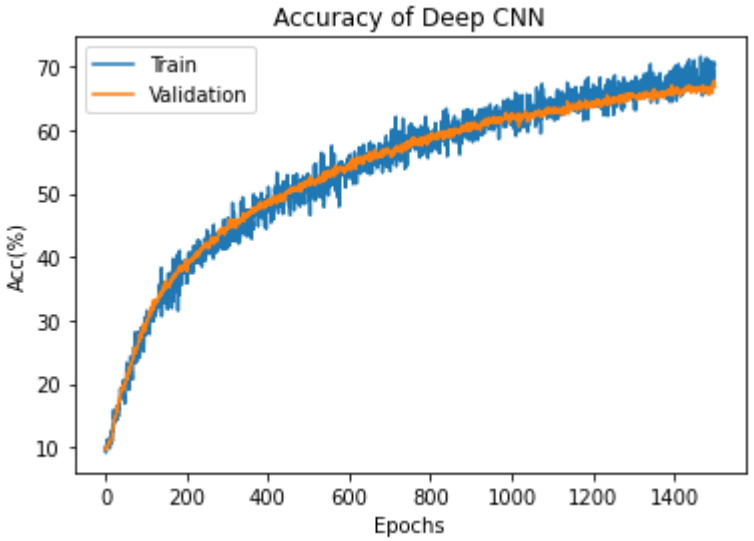
In this section, we just changed the weight matrix between workers. For this section, we consider a fixed stochastic matrix. We used samples from a uniform distribution over [0, 1), and normalized weights of each row to be a stochastic matrix.

4.3.1. Implementation Results

We trained the previous architecture on the client dataset based on the GoSGD algorithm with the hyperparameters as follows:

During the training, the learning rate is constantly equal to 0.01, and the weight decay to 1e-4. All mini-batches contain 128 images. We use eight workers. The probability p controls the exchange frequency set to 1, as in the original paper. The network was trained for 1500 iterations on the train and validation datasets. The results are as follows:





As we can see in the plots, the performance in 1500 iterations is lower compared to the centralized version. However, it is better than the time-varying matrix result and roughly the same as the fixed doubly stochastic matrix. The performance of the network could be better if we trained for more iterations, but due to time-limit and for the sake of comparison, we trained all versions for 1500 rounds.

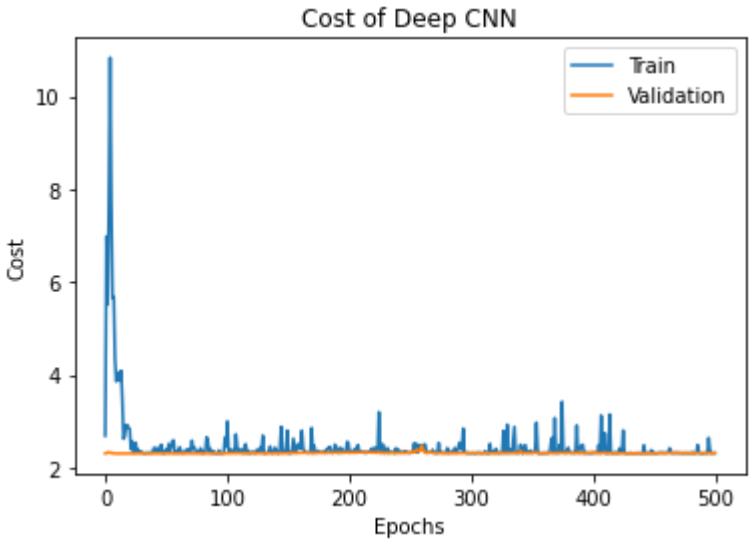
4.4. GoSGD with Stochastic Weight Matrix And Noisy Communication

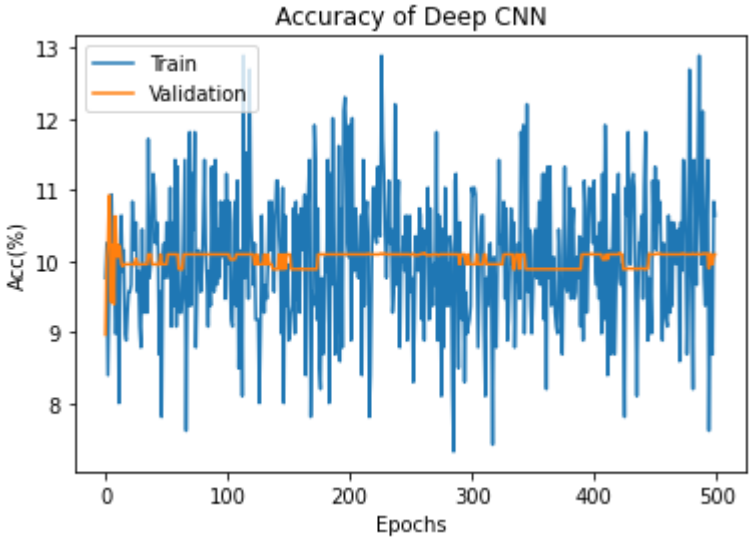
In this section, we changed the weight matrix between workers and also added some gaussian noise in the communication between agents. For this section, we consider a fixed stochastic matrix. We used samples from a uniform distribution over [0, 1), and normalized weights of each row to be a stochastic matrix.

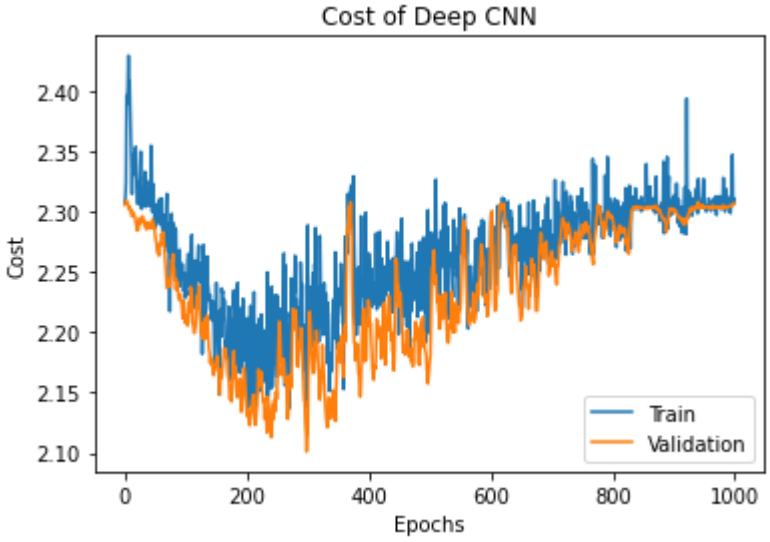
4.4.1. Implementation Results

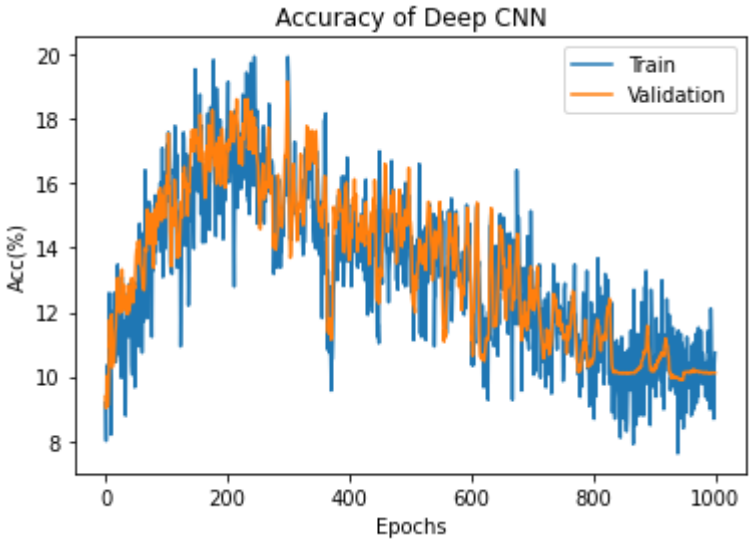
We trained the previous architecture on the client dataset based on the GoSGD algorithm with the hyperparameters as follows:

During the training, the learning rate is constantly equal to 0.01, and the weight decay to 1e-4. All mini-batches contain 128 images. We use eight workers. The probability p controls the exchange frequency set to 1, as in the original paper. The network was trained for 1500 iterations on the train and validation datasets. The results for two different noises are as follows:


Cost of Deep CNN
Train
Validation
Cost
Epochs


Accuracy of Deep CNN
Train
Validation
Acc(%)
Epochs


Cost of Deep CNN
Train
Validation
Cost
Epochs



From the above plots, we can conclude that the GoSGD algorithm is not robust to noise in either case. In the first case, the network did not learn anything because the strength of the noise was heavier. However, in the second case, the network was learning slowly, but it did not go well.

4.5. SGP with Stochastic Weight Matrix

In this section, we study Stochastic Gradient Push (SGP), an algorithm blending parallel SGD and PUSHSUM. SGP enables generic communication topologies that may be directed (asymmetric), sparse, and time-varying. We consider the setting where a network of n nodes cooperates to solve the stochastic consensus optimization problem. Each node has local data following a distribution Di, and the nodes wish to cooperate to find the parameters x of a DNN that minimizes the average loss with respect to their data, where F_i is the loss function at node i. Moreover, the goal codified in the constraints is for the nodes to reach an agreement (i.e., consensus) on the solution they report. We assume that nodes can locally evaluate stochastic gradients, but they must communicate to access information about the objective functions at other nodes. Like section 4.3, we consider a fixed stochastic matrix for this section. We used samples from a uniform distribution over [0, 1), and normalized weights of each row to be a stochastic matrix.

The Pseudo code of this algorithm is as follows:

Algorithm 1 Stochastic Gradient Push (SGP)

Require: Initialize $\gamma > 0$, $\boldsymbol{x}_i^{(0)} = \boldsymbol{z}_i^{(0)} \in \mathbb{R}^d$ and $w_i^{(0)} = 1$ for all nodes $i \in \{1, 2, \ldots, n\}$

1: **for** $k = 0, 1, 2, \cdots, K$, at node i, **do**

2: Sample new mini-batch $\xi_i^{(k)} \sim \mathcal{D}_i$ from local distribution

3: Compute mini-batch gradient at $\boldsymbol{z}_i^{(k)}$: $\nabla \boldsymbol{F}_i(\boldsymbol{z}_i^{(k)}; \xi_i^{(k)})$

4: $\boldsymbol{x}_i^{(k+\frac{1}{2})} = \boldsymbol{x}_i^{(k)} - \gamma \nabla \boldsymbol{F}_i(\boldsymbol{z}_i^{(k)}; \xi_i^{(k)})$

5: Send $\left(p_{j,i}^{(k)} \boldsymbol{x}_i^{(k+\frac{1}{2})}, p_{j,i}^{(k)} w_i^{(k)}\right)$ to out-neighbors;
receive $\left(p_{i,j}^{(k)} \boldsymbol{x}_j^{(k+\frac{1}{2})}, p_{i,j}^{(k)} w_j^{(k)}\right)$ from in-neighbors

6: $\boldsymbol{x}_i^{(k+1)} = \sum_j p_{i,j}^{(k)} \boldsymbol{x}_j^{(k+\frac{1}{2})}$

7: $w_i^{(k+1)} = \sum_j p_{i,j}^{(k)} w_j^{(k)}$

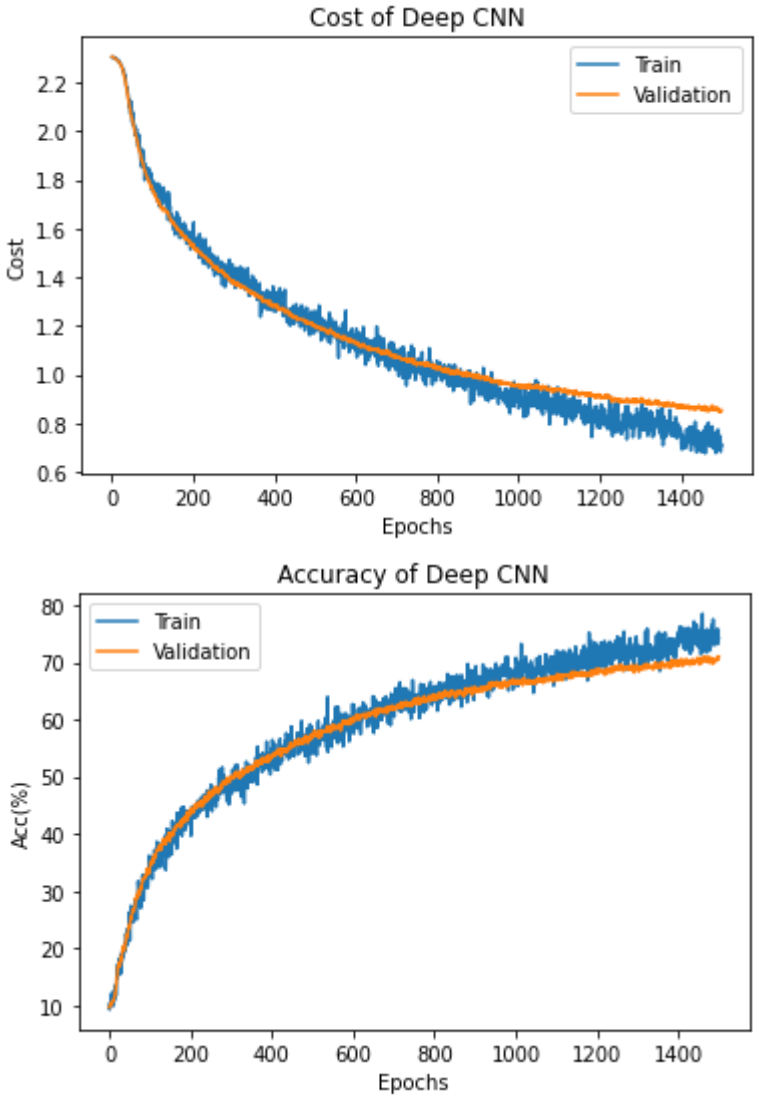
8: $\boldsymbol{z}_i^{(k+1)} = \boldsymbol{x}_i^{(k+1)} / w_i^{(k+1)}$

9: **end for**

4.5.1. Implementation Results

We trained the previous architecture on the client dataset based on the SGP algorithm with the hyperparameters as follows:

During the training, the learning rate is constantly equal to 0.01, and the weight decay to 1e-4. All mini-batches contain 128 images. We use eight workers. The probability p controls the exchange frequency set to 1, as in the original paper. The network was trained for 1500 iterations on the train and validation datasets. The results are as follows:



As we can see in the plots, the performance in 1500 iterations is roughly the same as the centralized version. However, it is better than the fixed stochastic matrix result. The performance of the network could be better if we trained for more iterations, but due to time-limit and for the sake of comparison, we trained all versions for 1500 rounds.

Section 5) Distributed Optimization with data-distributed

The assumption that the clients have all the data is discarded in this section. However, the other assumption from the previous section is still valid here. Gossip-based distributed deep learning and federated learning are implemented for this part. For this section, like section 4.3, we consider a fixed stochastic matrix. We used samples from a uniform distribution over [0, 1), and normalized weights of each row to be a stochastic matrix. We use the same federated learning setting as the last project on a different network.

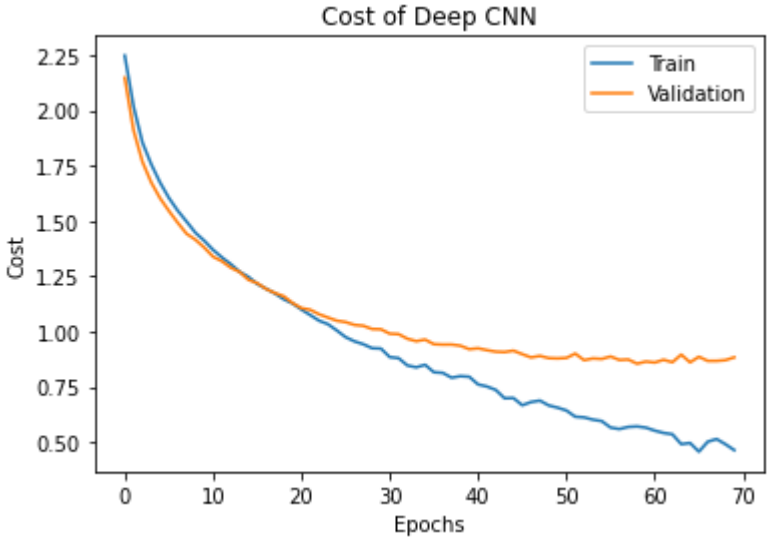
5.1. Splitting the Data

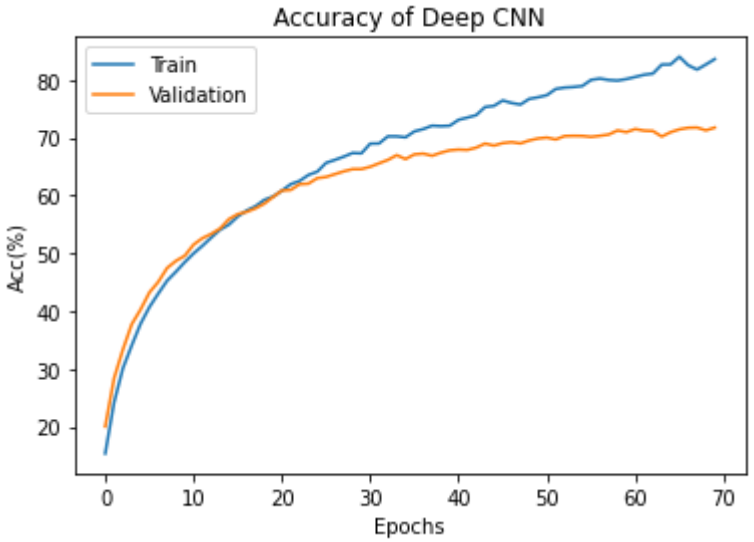
There are 50,000 train images in the cifar10 dataset and ten classes, each with 5,000 train images. Also, I assumed that there were eight clients to be compatible with the paper. Additionally, for this section, the assumption is that client data are i.i.d. Thus, I split the data equally between clients, and each client has 6250 images from all ten classes uniformly. In other words, each client has 625 images from each class.

5.2. Implementation Results

We trained the previous architecture on the client dataset based on the GoSGD algorithm. In addition, with a minor change that at each iteration, a worker updates its hosted network's weights with stochastic gradient descent on all mini-batches to be more compatible with federated learning in this algorithm with the hyperparameters as follows:

During the training, the learning rate is constantly equal to 0.01, and the weight decay to 1e-4. All mini-batches contain 128 images. We use eight workers. The probability p controls the exchange frequency, which is set to 1, as in the original paper. The network was trained for 70 epochs on the train and validation datasets. The results are as follows:





To solve this decentralized optimization, we implemented the FedAvg algorithm, which was first introduced in [4]. The Pseudo code of this algorithm is as follows:

Algorithm 1 `FederatedAveraging`. The K clients are indexed by k; B is the local minibatch size, E is the number of local epochs, and η is the learning rate.

Server executes:

initialize w_0

for each round $t = 1, 2, \dots$ **do**

$m \leftarrow \max(C \cdot K, 1)$

$S_t \leftarrow$ (random set of m clients)

for each client $k \in S_t$ **in parallel do**

$w_{t+1}^k \leftarrow \text{ClientUpdate}(k, w_t)$

$w_{t+1} \leftarrow \sum_{k=1}^{K} \frac{n_k}{n} w_{t+1}^k$

ClientUpdate(k, w): *// Run on client k*

$\mathcal{B} \leftarrow$ (split $\mathcal{P}_k$ into batches of size B)

for each local epoch i from 1 to E **do**

for batch $b \in \mathcal{B}$ **do**

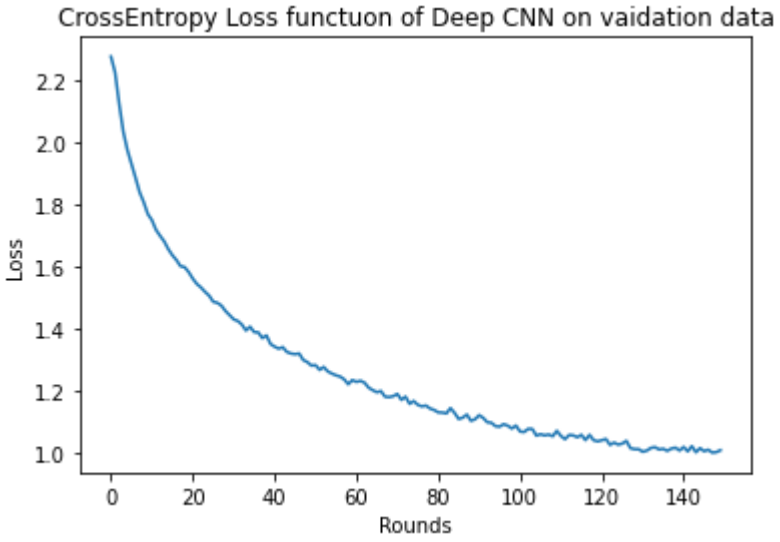
$w \leftarrow w - \eta \nabla \ell(w; b)$

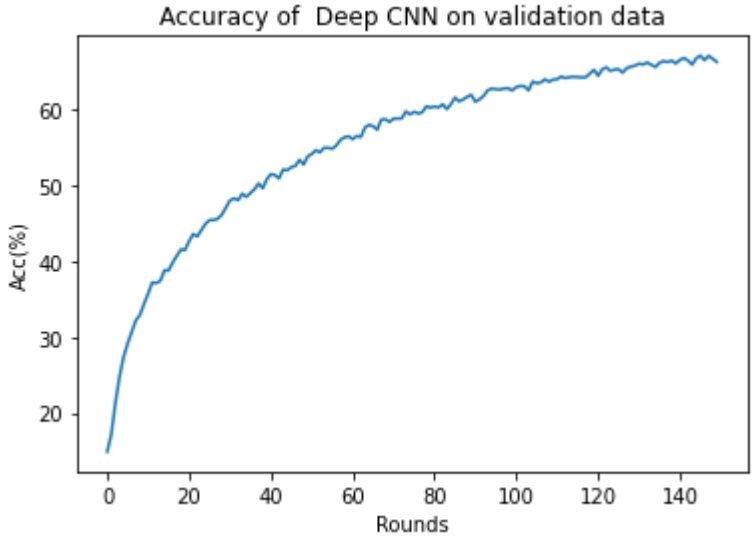
return w to server

We used the same hyperparameters as in the original paper.

Hyperparameter	E	K	C	B
Value	5	100	0.1	50

We trained the architecture on the client dataset based on the FedAvg algorithm with the mentioned hyperparameters. We used the "Cross-Entropy" loss function and the "SGD" as an optimizer function with a learning rate of 0.01. The network was trained for 150 rounds on the train and validation datasets. The results are as follows:





As we can see in the plots, the performance of the distributed optimization was the same as the centralized version. However, it is better than the federated learning result. Furthermore, distributed optimization needed much less communication and was two times faster than the federated learning algorithm.

Section 6) Conclusion

In this project, We implemented a convolutional neural network on the cifar10 dataset in 3 three different scenarios:

1) When a global client has access to all of the training samples(Classic centralized Version)
2) When the clients use parallel data, exploiting gossip-based distributed optimization with different weight matrices and in the presence of noise.
3) When the data had been splitted in a i.i.d. manner through the clients, both with and without a global client.

For this purpose, we implemented the GoSGD and the SGP algorithms. In conclusion, we can say that in the second part, we saw that GoSGD is not robust with respect to noise in communication. As stated in the paper, the fixed stochastic matrix had a bit better performance than the default time-varying weight matrix but roughly the same as the doubly stochastic weight matrix. However, the SGP algorithm outperformed GoSGD with the same fixed stochastic weight matrix manner and reached the centralized performance.

In the second part, we discarded the assumption that all clients(workers) had all the data and compared the result of distributed deep learning versus decentralized deep learning.

References:

[1] Blot, Michael, et al. "Gossip training for deep learning." arXiv preprint arXiv:1611.09726 (2016).

[2] Assran, Mahmoud, et al. "Stochastic gradient push for distributed deep learning." International Conference on Machine Learning. PMLR, 2019.

[3] Krizhevsky, Alex, Vinod Nair, and Geoffrey Hinton. "The cifar-10 dataset." *online: http://www. cs. toronto. edu/kriz/cifar. html* 55.5 (2014).

[4] McMahan, Brendan, et al. "Communication-efficient learning of deep networks from decentralized data." *Artificial intelligence and statistics*. PMLR, 2017.