SwarmAvg: A Novel Approach to Fully Distributed Machine Learning

Josh Pattman[1]

Abstract— **Federated learning is a technique that allows a machine learning model to be trained on data distributed across multiple data islands. This approach protects privacy by keeping the data decentralized, meaning that sensitive data does not need to ever be shared. Swarm learning is a similar technique that eliminates the need for a central server, ensuring that not only the data, but also the communication, is completely decentralized. Current Swarm Learning algorithms rely on blockchain to distribute the shared global model, however this may be a poor choice for certain scenarios closely associated with swarms. In this paper, a novel swarm learning technique called SwarmAvg is presented which operates without a blockchain. The algorithm is validated against federated learning in various scenarios. The benefits and drawbacks of operating Swarm Learning without a blockchain are also discussed, exemplifying some interesting reasons why one might choose to use SwarmAvg over other distributed machine learning techniques.**

I. BACKGROUND

Machine learning is becoming an exceedingly vital tool for our society to progress. However, many modern machine learning algorithms require large volumes of diverse data to achieve optimal performance. In the ideal world, this data would be stored in a single location close to a very powerful computer for training. Unfortunately, real-world data is often distributed among multiple nodes that are unable to share the data with each other or a central location, due to privacy regulations such as GDPR [1]. Accessing a super computer for training is also a luxury that many cannot afford. The reduction in data volume available to a single machine can negatively impact the post training performance [2], and the use of a slower computer means that training may not be able to be performed as fast as needed. One possible remedy to this problem is the distribution of both data and computation amongst multiple nodes.

A. Federated Learning

Federated Learning (FL) [3] is a technique in machine learning that aims to train a single model using all available data across nodes without requiring any data to be shared among them.

There are a multitude of published FL frameworks [4], each with different merits and drawbacks for certain use cases. Federated Averaging (FedAvg) [5] is a commonly used yet simple framework, which splits training into iterations where three steps take place:

1) A copy of the current model is sent to each node from the central server.
2) Each node performs some training with their copy of the model and their own private data.
3) The trained models from each node are sent back to the server to aggregate into the new server model.

The server model is improved over time, beyond what could be achieved by simply training on a the data stored on a single node.

As it does not require data to be shared between nodes, FL is naturally beneficial for privacy sensitive tasks compared to conventional machine learning where the data is aggregated in a central location [6]. Additionally, as FL performs training on multiple nodes in parallel, it can make better use of available training resources in situations where processing power is not only distributed among multiple nodes but also limited on each node, such as Internet of Things (IoT) [7].

FL has been adapted into several variations which eliminate the need for a single central server. One such variant is Multi-Center Federated Learning [8], which involves multiple central servers, each connected to different clusters of nodes. Another approach is to use leader election to select a node to function as the server [9]. This method can improve network resilience because, if the server fails, a new server is elected.

B. Swarm Learning

Swarm Learning (SL) is a subcategory of FL which operates in a completely distributed and decentralised manner. SL enables the collaboration of nodes to learn a shared global model, however in contrast to FL, a central server is never used. SL also does not use leader election, so all nodes on the network are given equal importance.

In SL, the model on which new training is performed is known as the global model. However, unlike FL where the global model is stored in a central location, the global model in SL does not materially exist, but is instead a concept which is agreed upon by the nodes in the network. An illustration of the different algorithms is shown in Figure 1.

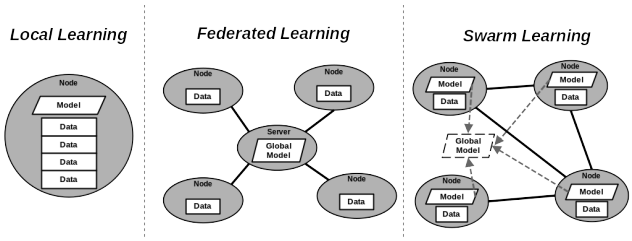


Fig. 1: Diagram of different learning algorithms.

One SL algorithm, referred to by this paper as SwarmBC, uses a blockchain to store the global model [10]. In this version of SL, training is performed by repeating the following steps:

1) A node obtains a copy of the global model from the blockchain.
2) The node performs some training with their copy of the model and their own private data.
3) The updated model is merged with the latest blockchain model and sent back to the blockchain for other nodes to use.

SL exhibits many of the same benefits of FL over conventional learning, but it also improves upon FL in some aspects. As is often the case when comparing decentralised algorithms to their centralised counterparts [11], the absence of a central server theoretically makes SL more resilient and robust to failures than FL. A SL system is also less sensitive to disruptions in connections between nodes. In FL, if a connection between two nodes ceases to exist, the node can no longer participate in learning. In contrast, in SL, even if a connection between two nodes is lost, the two connected nodes are still likely to be connected to other operational nodes, enabling the system to continue functioning normally.

The utilisation of FL with leader election effectively addresses the challenge of ensuring system robustness, as it enables any node to serve as a replacement for the server in the event of network disconnection. Nevertheless, it is important to note that a swarm of nodes is not often completely interconnected, and the connections between the nodes are often subject to changes. This dynamic and sparse network topology can present significant complexities when it comes to leader election, leading to an increase in overhead [12]. The lack of a leader election process in SL makes it a more viable option in such situations.

Finally, a swarm of nodes can offer greater scalability compared to its centralized counterpart, particularly if agents are restricted to communicating with only their nearby neighbours [13]. This is primarily because in such cases, there is no need for all communications to travel through a single node or server. In a swarm where nodes are limited to a certain number of close neighbours, the size of the swarm becomes less significant, as nodes always have the same number of neighbours. Consequently, SL is typically more scalable than FL.

II. Algorithm Design

A. Design Overview

In SL, a node is an agent responsible for facilitating the improvement of the global model. The global model is an abstract concept representing the consensus of all nodes in the network. In the proposed version of SL, referred to as SwarmAvg, each node maintains its own model, known as the local model, which is an approximation of the global model. However, in SwarmAvg, the consensus algorithm used is repeated averaging, not blockchain like in SwarmBC. This means that at the start of each training step, every node may start with a slightly different model. As training progresses and performance begins to plateau, each node's local model should not only converge towards a minima with respect to loss, but also towards each other.

Each node in the network possesses a confidential dataset. In order to train the global model, nodes fit their own local model on their local dataset. To ensure each nodes models do not diverge from each other, the local model is then combined with the neighbours local models. This process is repeated until training is complete. The actions in each training step for SwarmAvg are as follows:

1) Fit local model to local dataset.
2) Send local model to all neighbours.
3) Combine neighbouring models into local model.

Additionally, each node retains a local cache of the most recent models of its neighbouring nodes. This cache is updated each time a neighbouring node transmits its model to the node in question, instead of being updated on-demand during the combination step. This means that nodes never have to wait for other nodes to reply to them, which could mitigate issues with poor internet connections.

B. The Algorithm

The training step, found at Algorithm 1, is the section of the algorithm which runs continuously during the lifetime of a node. It takes care of training and synchronising the local model. The provided algorithm represents the logic of a single training step, meaning that it should be run in a loop that terminates once a stop condition, such as target accuracy, has been reached.

The model received event, which can be found at Algorithm 2, is run on the local node every time a remote node sends the local node a model update. This event takes care of updating the local model cache, to ensure the local node has the most up-to-date information. The model update sent from the remote node should contain the model and training counter of the remote node.

In both of the below algorithms, a capital M stands for the word *Model*, and TC stands for the words *Training Counter*. The function $\mu_e(x)$ means the element-wise $mean(x)$ where x is a list of arrays of identical lengths, and the symbol $\mu(x)$ means the $mean(x)$, where x is a list of scalars.

Algorithm 1 A Single Training Step

1: TRAIN(localM, localData)
2: $localTC \leftarrow localTC + 1$
3: **for all** $n \in neighbors$ **do**
4: SENDTO($(localM, localTC)$, n)
5: **end for**
6: **for** $x \in range(maxSyncWaits)$ **do**
7: $neighborMs \leftarrow \emptyset$
8: $neighborTCs \leftarrow \emptyset$
9: **for all** $n \in neighbors$ **do**
10: $M, TC \leftarrow$ CACHELOOKUP(n)
11: **if** $TC + \beta \geq localTC$ **then**
12: APPEND($neighborMs$, M)
13: APPEND($neighborTCs$, TC)
14: **end if**
15: **end for**
16: **if** $length_{neighborMs} \geq \gamma$ **then**
17: $localM \leftarrow (1 - \alpha) * localM + \alpha * \mu_e(neighborMs)$
18: $localTC \leftarrow (1 - \alpha) * localTC + \alpha * \mu(neighborTCs)$
19: break
20: **else**
21: SLEEP(syncWaitTime)
22: continue
23: **end if**
24: **end for**

Algorithm 2 Model Received Event

Input: $neighbour$, $neighbourM$, $neighbourTC$

1: **if** INCACHE($neighbour$) **then**
2: $\_, neighbourTCOld \leftarrow$ CACHELOOKUP($neighbour$)
3: **if** $neighbourTC > neighbourTCOld$ **then**
4: SETCACHE($neighbour$, $(neighbourM, neighbourTC)$)
5: **end if**
6: **else**
7: SETCACHE($neighbour$, $(neighbourM, neighbourTC)$)
8: **end if**

C. Blockchain-less Algorithm

Whereas SwarmBC employs a blockchain as the mechanism for distributing the global model, SwarmAvg utilises a variation of averaging with its neighbours and does not use a blockchain. This decision was made due to the long transaction confirmation time of large blockchains, which could adversely affect training as nodes continuously upload their latest models to the network. If this process were to take an excessive amount of time, each node would be training on an outdated model.

Furthermore, blockchains are typically performance intensive, but SwarmAvg is designed for deployment in large swarms with each agent having low processing power. If a blockchain were utilised, some of the processing power that could be devoted to training would instead be utilised for validating transactions. In scenarios where processing power is limited, it would be practical to avoid the use of blockchains.

D. The Training Counter

A vital aspect of SwarmAvg, specifically the combination step, involves evaluating the performance of each nodes local model. The conventional approach would involve testing each model using an independent test set. However, due to the inability to exchange data among nodes, there is no way for each node to share the same test set, resulting in non-comparable scores for each model. In order to circumvent this problem, this paper presents a heuristic metric referred to as the "training counter," which serves as an approximation of the level of training for a network by estimating the number of training steps performed on a given model.

The training counter is incremented by 1 when the local model is trained on the local dataset, indicating that an additional step of training has been performed. Following the combination step, the training counter is also updated to reflect the how the models were combined, which can be seen in Line 18 of Algorithm 1.

E. Combining Neighbouring Models

During the combination step, a node merges its local model with those of its neighbours, producing an updated estimate of the global model. The method of combination used by this paper is called Averaging with Synchronisation Rate (ASR). This involves computing the average model of all neighbours, then calculating the weighted mean between that model and the local model. This is demonstrated on Line 17 of Algorithm 1.

The synchronisation rate, denoted as α, indicates the degree to which each node adjusts its local model to align with the global model. If α is set too low, each node's model in the network will diverge, resulting in each node becoming trapped at a local minima, an effect referred to as divergent training. On the other hand, if α is too high, the progress achieved by the local node will be discarded at each averaging step, which can result in slower learning.

This method is beneficial over simply averaging all known models, as it provides some resistance to variation in the number of neighbours of a node. When using simple averaging, the amount of change made by the local node that persists into the next training step is inversely proportionate to the number of neighbours which were combined. However, when using ASR, the amount of change made that persists will be independent of how many neighbours that are present.

This may increase the stability of training if the environment involves a dynamic set of neighbours.

F. Training Counter Filtering

A node may only include its neighbour models if they meet the following statement:

$$neighborTrainingCounter + \beta \geq localTrainingCounter$$

The training offset β is the amount the training counter of a neighbour can be behind the local training counter before it is ignored.

The reason for training counter filtering is increased fault tolerance. Consider the situation where node *A* has received model updates from many nodes, one of which being node *B*. However, node *B* goes offline and no longer is sending model updates. Without training counter filtering, node *A* will continue to combine the outdated node *B* model with it's own for as long as it is training. However, if training counter filtering is enabled, after a number of training steps the outdated model *B* updates will be ignored.

An issue with training counter filtering pertains to the presence of runaway nodes. These nodes possess a substantially higher training counter compared to all other nodes in the network, meaning that when filtering is applied, they are left with no neighbours to utilise in the combination step. Consequently, a runaway node may start to overfit on its own training data, thereby leading to decreased local performance, as well as potential performance reductions in the rest of the network. To address this problem, each node must wait until it has obtained at least γ viable neighbours prior to performing the combination step.

III. Strategy for Gathering of Results

The experiments were conducted through the simulation of a group of nodes on a single computer. The accuracy of each node was evaluated on an unseen test set and subsequently recorded in a file after each training step. All nodes shared the same test set. The dataset used was MNIST Fashion (MNIST-F). The machine learning model was a simple Convolutional Neural Network. Below are the specific details of the running of each experiment:

A. Chosen Metric: Accuracy

The metric chosen for the following experiments was accuracy, due to its comprehensible nature. The accuracy of each node is computed after each training step using the test subset of MNIST-F, and none of the nodes are ever provided access to the test set for training.

B. Number of Simulated Nodes

The experiments were conducted using 10 nodes, not including the server in cases where FedAvg was employed. The decision of how many nodes to simulate was based on the highest node count attainable without causing inconsistencies and crashes due to resource depletion of the training machine.

C. Repeated Testing for Noise Reduction

The training process for each experiment was conducted 10 times, and the resulting accuracies of every node were recorded. The accuracy value for each time step was calculated as the median accuracy across all nodes and runs at that time step.

D. Configuring the SwarmAvg Algorithm

In each of the following experiments, the algorithm was configured using a specific set of parameters $\alpha\beta\gamma$. These parameters were obtained heuristically by making an initial guess, testing, and then fine-tuning them until a satisfactory outcome was reached.

E. Data Provided to Each Node

The experiments evaluate the algorithm's performance using three levels of data volume per node. These levels are considerably smaller than the full MNIST-F dataset as the algorithm is intended for scenarios where each nodes access to data is restricted. To create a subsection of data for each node, a random sampling with replacement method was used to select the desired number of datapoints. The random sampling was performed only once for each node, after which that nodes data subset remains constant. The three levels of data volume can be seen in TABLE **??**. Additionally in one of the tests, each node only had access to 3 out of the 10 possible classes. This was achieved by assigning each node 3 random classes which it could sample data from.

F. Sparsely Connected Network of Nodes

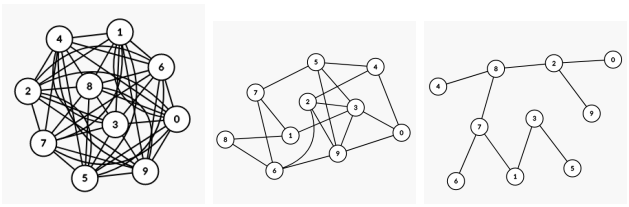
In reality, it is uncommon for each node to be linked with every other node. To simulate a more realistic scenario, a value *Density* (ρ). When ρ is set to 0, the network is minimally linked, meaning that each node has at least one indirect path to every other node, but the minimal number of connections required to accomplish this exist. When ρ is set to 1, all nodes are connected to one another in a dense fashion. Since this measure may not provide a straightforward indicator of network topology, two additional metrics are provided: Mean Minimum Hops (MMH) and Mean Connections per Node (MCPN). MMH denotes the mean minimum number of transitions required to get from one node to another in the network. MCPN denotes the average number of connections a given node possess, for a network of 10 nodes.

Several different values of ρ were tested for each count of data. The values that were tested, along with their corresponding MMH and MCPN, are presented in TABLE I. Also included is *Number of Federated Learning Nodes* (NFLN), which denotes the number nodes that FL was given to approximate the number of connections a SL node would have for that value of ρ.

ρ	MMH	MCPN	NFLN
1	1.0	9.0	10
0.25	1.7	3.6	4
0	3.0	1.8	2

TABLE I: The statistics for different density levels

In order to assist the reader with visualizing the various values of ρ, some sample networks that were generated for each ρ can be found in Figure 2.



(a) $density = 1.0$ (b) $density = 0.25$ (c) $density = 0.0$

Fig. 2: Example networks of nodes generated for each ρ value, visualised using the tool at *https://csacademy.com/app/graph_editor*.

IV. Results

Tests were performed according to the methodology set out in Section III.

A. Test 1 - Comparison of Methods whilst Varying ρ

The first experiment performed involved testing both FedAvg and SwarmAvg at differing values of ρ and comparing the two methods. This test was performed over different data volumes to simulate more of a range of conditions for the algorithms. Below are the results obtained from these tests.

Accuracy by Training Step for 1000 Samples, with Varying Density

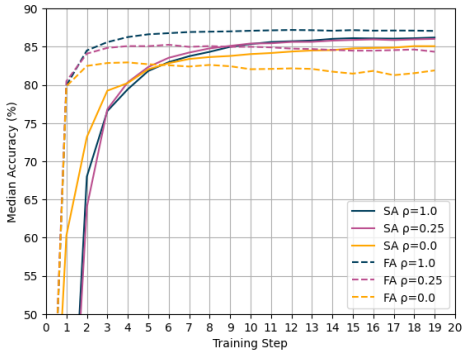


Fig. 3: Median accuracy by training step across 10 repeats. Each node has 1000 random data samples from MNIST-F.

Accuracy by Training Step for 100 Samples, with Varying Density

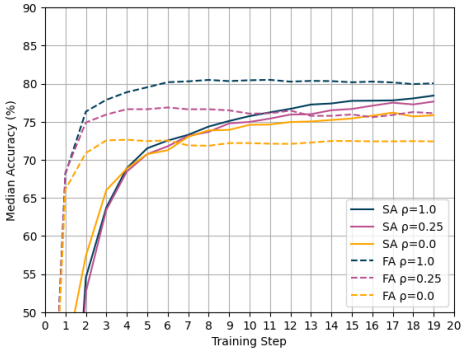


Fig. 4: Median accuracy by training step across 10 repeats. Each node has 100 random data samples from MNIST-F.

Accuracy by Training Step for 25 Samples, with Varying Density

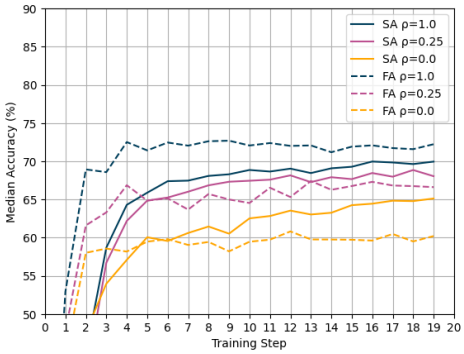


Fig. 5: Median accuracy by training step across 10 repeats. Each node has 25 random data samples from MNIST-F.

In all cases, it can be observed that the lower values of ρ cause the performance of both algorithms to decrease, however the decrease in performance is more severe when each node in the network had access to a lower volume of data. Furthermore, FedAvg is affected to a significantly higher degree by the decrease in ρ than SwarmAvg. This is apparent due to the fact that FedAvg performs better than SwarmAvg for the high value of ρ, but falls behind when run with a lower ρ value.

It should also be noted that SwarmAvg takes more training steps that FedAvg to converge on its highest accuracy in all tests. This implies that SwarmAvg takes longer to perform training than FedAvg, which may be a consideration when applying the algorithm to a use-case.

B. Test 2 - Comparison of Methods whilst Varying ρ with 3 Available Classes

Decentralized machine learning algorithms commonly involve a scenario where each node possesses a dataset whose distribution does not align with the overall data distribution across all nodes. In order to simulate such a scenario, a situation was created where each node was given access a unique set of three out of ten classes of MNIST-F. The

performance of SwarmAvg and FedAvg were tested for each ρ value, with the results shown below.

Accuracy by Training Step for 1000 Samples, with Varying Density, and 3 Classes

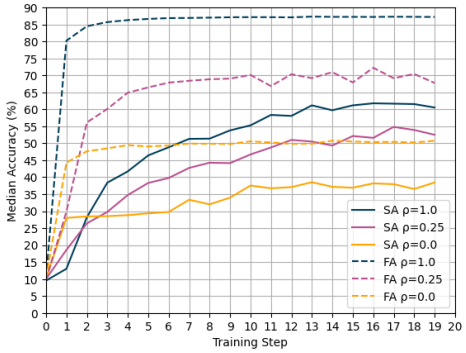


Fig. 6: Median accuracy by training step across 10 repeats. Each node has 1000 random data samples from MNIST-F.

Accuracy by Training Step for 100 Samples, with Varying Density, and 3 Classes

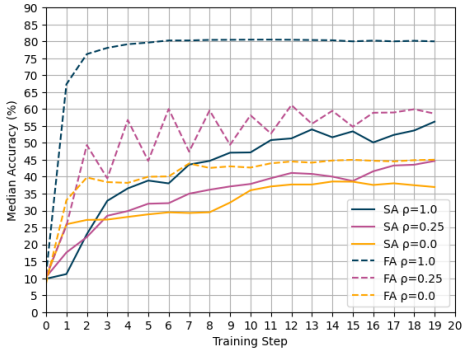


Fig. 7: Median accuracy by training step across 10 repeats. Each node has 100 random data samples from MNIST-F.

Accuracy by Training Step for 25 Samples, with Varying Density, and 3 Classes

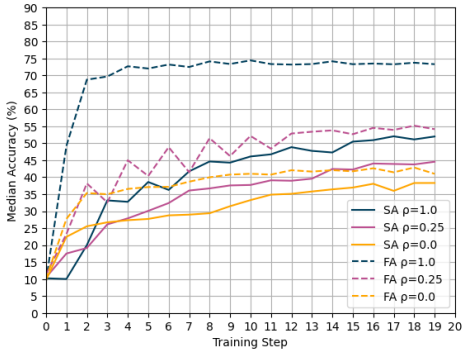


Fig. 8: Median accuracy by training step across 10 repeats. Each node has 25 random data samples from MNIST-F.

In all three figures, it is clear that SwarmAvg performs worse than FedAvg when both algorithms are run with the same ρ and data volume. It can be observed that the discrepancy in performance is exaggerated with both higher volumes of data, and higher values of ρ.

In both Fig. 7 and Fig. 8, the FedAvg result for a ρ value of 0.25 seems to cause some sort of oscillation in training accuracy. After examination of the raw results, this does not appear to be an issue related to having too few repeats, but instead seems to be a consistent feature in many of the training runs.

In all cases, the initial training steps performed by FedAvg result in a higher accuracy than SwarmAvg, meaning that FedAvg is faster at training than SwarmAvg in these tests.

V. Comparison of SwarmAvg to FedAvg

SwarmAvg has the significant benefit of being more fault tolerant than FedAvg, primarily due to its lack of a central server. Not only this, but in FedAvg if a node drops its connection to the server, that node is effectively ignored from training, which has a negative effect on performance. In SwarmAvg, that node can still be included in training.

However, SwarmAvg has been shown to be slower to converge than FedAvg. Not only this, but SwarmAvg can create a significantly higher amount of network traffic than FedAvg or FL in general. In SwarmAvg at $\rho = 1$, the number of connections scale by $O(n^2)$, where n is the number of nodes. FL will always both number of connections and network traffic scale with $O(n)$, as every node only needs a single connection to the server. However, it is worth noting that SwarmAvg will consume less network traffic in networks of nodes with lower ρ values.

An important advantage of SwarmAvg over FedAvg is that former performed better in a sparsely connected network of nodes, when each node had access to an even distribution of data from the dataset. This may be a more realistic scenario for certain low powered networks of devices such as IoT networks or robotic swarms. However, when restricting each node to have access to just 3 out of 10 of the possible classes, SwarmAvg experiences a far greater drop in performance than FedAvg. Reducing this performance decrease should be a future area of research.

Acknowledgment

I would like to thank my supervisor, Dr Mohammad Soorati, for supporting and inspiring me throughout the project, and really helping me to learn a lot more about the topic of this report. I would also like to thank my secondary supervisor, Dr Jo Grundy, for always being fast to respond to any questions I have had, and always being willing to help out.

Finally, I would like to thank my peers James Harcourt, Daniel Say, Jack Darlison, and Siobhan Tinsley for having engaging discussions with me about the direction of my project, and always providing helpful critical feedback.

REFERENCES

[1] M. Kop, "Machine learning & eu data sharing practices," Stanford-Vienna Transatlantic Technology Law Forum, Transatlantic Antitrust …, 2020.

[2] G. A. Kaissis, M. R. Makowski, D. Rückert, and R. F. Braren, "Secure, privacy-preserving and federated machine learning in medical imaging," *Nature Machine Intelligence*, vol. 2, no. 6, pp. 305–311, 2020.

[3] C. Zhang, Y. Xie, H. Bai, B. Yu, W. Li, and Y. Gao, "A survey on federated learning," *Knowledge-Based Systems*, vol. 216, p. 106775, 2021.

[4] Q. Li, Z. Wen, Z. Wu, S. Hu, N. Wang, Y. Li, X. Liu, and B. He, "A survey on federated learning systems: vision, hype and reality for data privacy and protection," *IEEE Transactions on Knowledge and Data Engineering*, 2021.

[5] H. B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y. Arcas, "Communication-efficient learning of deep networks from decentralized data," 2016.

[6] V. Mothukuri, R. M. Parizi, S. Pouriyeh, Y. Huang, A. Dehghantanha, and G. Srivastava, "A survey on security and privacy of federated learning," *Future Generation Computer Systems*, vol. 115, pp. 619–640, 2021.

[7] T. Zhang, L. Gao, C. He, M. Zhang, B. Krishnamachari, and A. S. Avestimehr, "Federated learning for the internet of things: Applications, challenges, and opportunities," *IEEE Internet of Things Magazine*, vol. 5, no. 1, pp. 24–29, 2022.

[8] M. Xie, G. Long, T. Shen, T. Zhou, X. Wang, and J. Jiang, "Multi-center federated learning," *CoRR*, vol. abs/2005.01026, 2020.

[9] J. Guo, Q. Zhao, G. Li, Y. Chen, C. Lao, and L. Feng, "Decentralized federated learning with privacy-preserving for recommendation systems," *Enterprise Information Systems*, vol. 0, no. 0, p. 2193163, 2023.

[10] W.-H. et al., "Swarm learning for decentralized and confidential clinical machine learning," *Nature*, vol. 594, pp. 265–270, Jun 2021.

[11] J. C. Varughese, R. Thenius, T. Schmickl, and F. Wotawa, "Quantification and analysis of the resilience of two swarm intelligent algorithms.," in *GCAI*, pp. 148–161, 2017.

[12] J. Augustine, T. Kulkarni, and S. Sivasubramaniam, "Leader election in sparse dynamic networks with churn," in *2015 IEEE International Parallel and Distributed Processing Symposium*, pp. 347–356, IEEE, 2015.

[13] M. Dorigo, M. Birattari, and M. Brambilla, "Swarm robotics," *Scholarpedia*, vol. 9, no. 1, p. 1463, 2014. revision #138643.