

Empowering Federated Learning for Massive Models with NVIDIA FLARE

Holger R. Roth, Ziyue Xu, Yuan-Ting Hsieh, Adithya Renduchintala, Isaac Yang, Zhihong Zhang, Yuhong Wen, Sean Yang, Kevin Lu, Kristopher Kersten, Camir Ricketts, Daguang Xu, Chester Chen, Yan Cheng, Andrew Feng
NVIDIA Corporation
Santa Clara, USA

ABSTRACT

In the ever-evolving landscape of artificial intelligence (AI) and large language models (LLMs), handling and leveraging data effectively has become a critical challenge. Most state-of-the-art machine learning algorithms are data-centric. However, as the lifeblood of model performance, necessary data cannot always be centralized due to various factors such as privacy, regulation, geopolitics, copyright issues, and the sheer effort required to move vast datasets. In this paper, we explore how federated learning enabled by NVIDIA FLARE can address these challenges with easy and scalable integration capabilities, enabling parameter-efficient and full supervised fine-tuning of LLMs for natural language processing and biopharmaceutical applications to enhance their accuracy and robustness.

KEYWORDS

Federated Learning, Massive Models, Large Language Models, Natural language Processing, Biopharma, Drug Discovery, Privacy.

1 INTRODUCTION

Data management and utilization are pivotal challenges in the dynamic realm of artificial intelligence (AI) and large language models (LLMs). Contemporary machine learning algorithms often rely on data-centric approaches and face obstacles in centralized data handling due to multifaceted concerns such as privacy, regulatory constraints, geopolitical factors, copyright issues, and the considerable logistical demands associated with moving extensive datasets.

This paper delves into the practical application of federated learning (FL), particularly exploring the capabilities offered by NVIDIA FLARE¹ (NVFlare) [29] in addressing these challenges. Through seamless and scalable integration, NVFlare facilitates parameter-efficient fine-tuning (PEFT) [8, 13] and full supervised fine-tuning (SFT). With a specific focus on applications in natural language processing (NLP) and biopharma utilizing modern LLM architectures. The aim is to enhance the accuracy and robustness of these models by applying FL in real-world situations.

The Data Challenge. The need to access data from multiple sources is a common scenario in many LLM tasks. Consider scenarios like gathering reports from different hospitals for medical research or collecting financial data from diverse institutions for analysis. Centralizing such data may be impractical and hindered by privacy concerns, regulatory hurdles, etc. FL offers an elegant solution to this issue [26].

Federated Learning. FL has emerged as a practical solution to tackle data challenges. Instead of centrally training models with access to raw data, FL facilitates sharing model updates rather than raw data itself. This means that participating clients can train models locally using their own private datasets to compute a local model update. These local updates are then combined globally to update the model parameters. This approach maintains the privacy of individual datasets while enabling the global model to benefit from the collective knowledge gained during training. The result is the training of more robust and generalizable models [6].

FL provides several options for training AI models. Essentially, it allows for the training of a global model while ensuring the privacy and governance of the data involved. Moreover, FL can be tailored to meet the specific needs of each client, thus enabling the creation of personalized models. Additionally, FL infrastructure extends beyond training and can also be utilized for tasks such as inference and federated evaluation.

2 METHODS

2.1 FL Framework

NVFlare is an open-source framework that allows researchers and data scientists to seamlessly move their machine learning and deep learning workflows into a federated paradigm. Furthermore, it empowers platform developers to construct secure and privacy-preserving solutions for collaborative multiparty distributed workflows. NVFlare is a lightweight, flexible, and scalable FL framework implemented in Python. Notably, it remains agnostic to the underlying training library, allowing developers to use PyTorch, TensorFlow, or even pure NumPy for their data science workflows in a federated setting. In the NVFlare ecosystem, a standard FL workflow, like the well-known federated averaging (FedAvg) algorithm [22], involves the following key steps. Each FL client receives an initial global model from the FL server and conducts local training on their data. Next, the clients transmit model updates to the server for aggregation. The server, in turn, applies these aggregated updates to refine the global model for subsequent training rounds. This procedure is repeated until convergence is achieved. While NVFlare finds frequent use in federated deep learning [6, 10, 17, 28, 32, 35, 39, 41], its versatility extends to supporting general federated computing across diverse clients. It provides the *Controller Programming API*, enabling researchers to craft flexible workflows for orchestrating client collaboration. FedAvg [22] and cyclic weight transfer [3] are examples of such workflows.

At the heart of NVFlare lies the concept of collaboration through "tasks." An FL controller assigns tasks (e.g., deep-learning training with model weights) to one or more FL clients, processes returned

¹<https://github.com/NVIDIA/NVFlare>

results (e.g., model weight updates), and may assign additional tasks based on these results and other factors (e.g., a pre-configured number of training rounds). This task-based interaction repeats until the experiment’s objectives are met.

2.2 Easy Adaptation of ML Workflows via Client API

The NVFlare *Client API* offers a convenient solution for users looking to transition their centralized, local training code to FL with several advantages:

- **Minimal Code Changes:** Users can achieve the transition with only a few lines of code adjustments, eliminating the need for a comprehensive restructuring or implementation of a new class.
- **Simplicity:** The *Client API* minimizes the introduction of new NVFlare-specific concepts to users, streamlining the adaptation process by leveraging familiar programming constructs.
- **Flexibility:** Users can easily adapt existing local training code written in various frameworks such as PyTorch, PyTorch Lightning, and HuggingFace, making the transition seamless and efficient.

The general structure of a popular FL workflow, such as *FedAvg* is as follows:

- (1) FL server initializes an initial model.
- (2) *For each round (global iteration):*
 - (a) FL server sends the global model to clients.
 - (b) Each FL client starts with this global model and trains on their own data.
 - (c) Each FL client sends back their model update.
 - (d) FL server aggregates all the updates and produces a new global model.

On the client side, the training workflow is as follows:

- (1) Receive the model from the FL server.
- (2) Perform local training on the received global model and/or evaluate the received global model for model selection.
- (3) Send the new model back to the FL server.

To convert a centralized training code to FL, we need to adapt the code to execute the following steps:

- (1) Obtain the required information from the received model.
- (2) Run local training.
- (3) Put the results in a new model to be sent back to the FL server.

For a general use case, there are three essential methods for the *Client API*:

- `init()`: Initializes NVFlare Client API environment.
- `receive()`: Receives model from the FL server.
- `send()`: Sends the model to the FL server.

With these simple methods, the developers can use the *Client API* to change their centralized training code to an FL scenario with five lines of code changes as shown in Listing 1.

```
1 import nvflare.client as flare
2
```

```
3 flare.init() # 1. Initializes NVFlare Client API environment.
4 input_model = flare.receive() # 2. Receives model from the FL
  server.
5 params = input_model.params # 3. Obtain the required
  information from the received model.
6
7 # original local training code
8 new_params = local_train(params)
9
10 output_model = flare.FLModel(params=new_params) # 4. Put the
   results in a new `FLModel`
11 flare.send(output_model) # 5. Sends the model to the FL server
   .
```

Listing 1: Client API example.

If using standardized training frameworks such as PyTorch Lightning, the conversion to FL can be even more streamlined. As an example in this paper, we use the GPT model from NVIDIA NeMo framework² [12] to show the application of PEFT and SFT for NLP tasks. NeMo leverages PyTorch Lightning for model training. One notable feature of NVFlare is the *Lightning Client API*, which significantly simplifies the process of converting local training scripts to run in FL scenarios. With just a few lines of code changes, one can seamlessly integrate methods like PEFT and SFT. As shown in Listing 2, the *Lightning trainer* can be adapted to run FL just by calling `flare.patch(trainer)`. Next, an extra while loop (`while flare.is_running():`) is added to allow reusing the same trainer object each round of FL. Optionally, we call `trainer.validate(model)` to evaluate the global model received from the FL server at the current round on the client’s data. This is useful for enabling global model selection on the server based on validation scores received from each client.

```
1 import nvflare.client.lightning as flare
2
3 ...
4 # 1. flare patch
5 flare.patch(trainer)
6
7 # 2. Add while loop to keep receiving the model in each FL
   round.
8 # Note, after `flare.patch` the trainer.fit/validate will get
   the
9 # global model internally at each round.
10 while flare.is_running():
11     # (optional): get the FL system info
12     fl_sys_info = flare.system_info()
13     print("--- fl_sys_info ---")
14     print(fl_sys_info)
15
16     # 3. evaluate the current global model to allow server-
   side model selection.
17     print("--- validate global model ---")
18     trainer.validate(model)
19
20     # 4. Perform local training starting with the received
   global model.
21     print("--- train new model ---")
22     trainer.fit(model)
```

Listing 2: Pytorch Lightning Client API example.

2.3 Server Workflow Implementation

NVFlare’s collaborative computing is achieved through the *Controller/Executor* interactions. The diagram in Fig. 1 shows how the *Controller* and *Executor* interact. The *Controller* is a class that controls or coordinates the *Executors* to get a job done. It is run on

²<https://www.nvidia.com/en-us/ai-data-science/generative-ai/nemo-framework>

the FL server (highlighted on the right). A *Executor* is capable of performing tasks. *Executors* run on FL clients and execute the client API described above. In its control logic (it's `run()` routine), the *Controller* assigns tasks to *Executors* and processes task results from the *Executors*. This allows the easy integration of additional data filters (for example, for adding homomorphic encryption [43] or differential privacy filters [19] to the task data or results received or produced by the server or clients).

In Listing 3, we show a simplified implementation of the FedAvg [22] algorithm with NVFlare³. The `run()` routine implements the main algorithmic logic. Subroutines, like `sample_clients()` and `scatter_and_gather_model()` utilize the *communicator* object, native to each *Controller* to get the list of available clients, distribute the current global model to the clients, and collect their results. For simplicity, we do not show the implementation of the aggregation, model update, and saving routines.

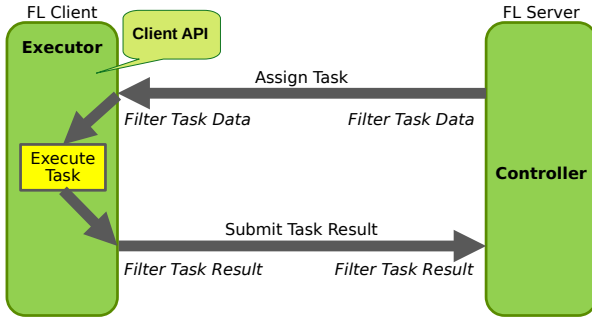


Figure 1: Server workflow *Controller* and *Executor* with *Client API*.

```

1  class FedAvg(Controller):
2      def __init__(self,
3          min_clients: int,
4          num_rounds: int
5          ):
6          self.model = ... # initialize the global model
7          ...
8
9
10     def run(self) -> None:
11         self.info("Start FedAvg.")
12
13         for _current_round in range(self._num_rounds):
14             self.info(f"Round {self._current_round} started.")
15             # 1. sample the available clients
16             clients = self.sample_clients(self._min_clients)
17             # 2. send the current global model to clients and
18             # receive the model updates
19             results = self.scatter_and_gather_model(targets=
20                 clients)
21             # 3. aggregate the results
22             aggregate_results = self.aggregate(results)
23             # 4. update the current global model
24             self.update_model(aggregate_results)
25             # 5. save the current global model
26             self.save_model()
27
28         self.info("Finished FedAvg.")
29
30     def sample_clients(min_clients):

```

³Scheduled for upcoming 2.5.0 release of NVFlare.

```

29     # add optional random sampling strategy
30     return self.communicator.get_clients()[0:min_clients]
31
32     def scatter_and_gather_model(targets, data):
33         return self.communicator.broadcast_and_wait(
34             task_name="train",
35             min_responses=self.min_clients,
36             data=self.model,
37             targets=targets,
38             callback=None)
39     ...

```

Listing 3: Federated averaging workflow controller example.

Due to the separation of the controller logic and the communication object (`self.communicator`), it is possible to run an NVFlare *Controller* both on the server and the clients, allowing a straightforward implementation of alternative communication strategies such as split learning [11] or swarm learning [40].

2.4 Scalable Model Training via Streaming

As FL or AI tasks in general become more and more complex, their model sizes increase [33]. The size of mainstream LLMs can be enormous, ranging from a few billion parameters to tens of billions of parameters, which leads to a significant increase in model sizes that need to be communicated during FL training. However, using native communication protocols directly can introduce inefficiencies and instability issues. Furthermore, protocols such as gRPC have hard size limits (2 GB) for single messages. Typical model sizes of modern LLMs exceed those limits and can even reach hundreds of GB. NVFlare supports large models as long as the system memory of servers and clients can handle it. However, it requires special considerations because the network bandwidth and, thus, the time to transmit such a large amount of data during an NVFlare job runtime varies significantly.

In the NVFlare 2.4.0 release, communication capabilities have been significantly enhanced through our new *data streaming API*. Our streaming API has four different variations: byte streaming, blob streaming, file streaming, and object streaming, which can work together with different communication protocols (drivers gRPC, HTTP, TCP, etc.). The "Streamable Framed Message" (SFM) layer manages the drivers and connections and sends messages. One can change the driver without affecting the upper-layer applications. In other words, one can switch between gRPC, TCP, HTTP, etc., and the applications built on top will work without any changes. One can even build customer drivers that suit their needs.

As illustrated in Fig. 2, the large model is now divided into 1 megabyte (MB) chunks and streamed to the target (server or client), bringing a complete transformation to the overall system with the introduction of a new streaming layer designed to handle large data transfers. Once the message arrives at the target end-point, the object is re-assembled to restore the original message payload. Refer to Section 4.1 for quantitative results on large data streaming.

3 APPLICATIONS

3.1 Adaption of Foundational LLMs

Foundational LLMs are pre-trained on a vast amount of general text data [2]. However, they may not be specialized for specific domains or downstream tasks. Further fine-tuning allows these models to adapt and specialize for particular domains and tasks, making them

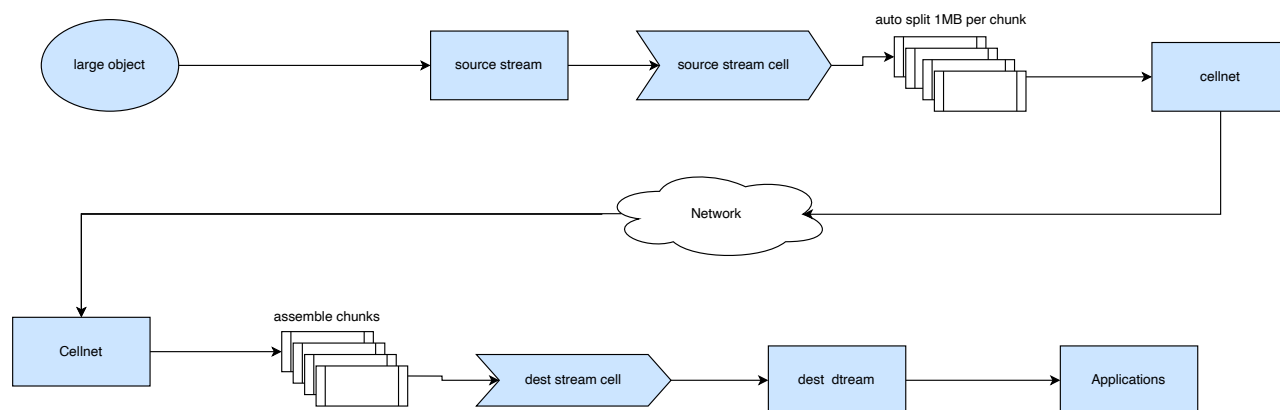


Figure 2: Data streaming API.

more effective and accurate in delivering domain- and task-specific results, which is essential to harness their potential and adapt them to various applications’ diverse and evolving needs.

PEFT and SFT are two vital approaches that aim to tailor foundational LLMs to specific domains and tasks efficiently and effectively. Both targets achieve domain and task-specific adaptation based on foundational LLMs. SFT fine-tunes all LLM parameters, while PEFT tries to add adaptation parameters/layers while keeping the LLM parameters fixed, making it a cost-effective and resource-efficient option. These techniques play a pivotal role in harnessing the power of LLMs for a multitude of applications, offering tailored and resource-aware solutions for a wide range of applications.

3.2 FL for LLM Adaptations

As with other AI techniques, the performance of LLMs benefits from larger and more diverse datasets. More data usually translates to better accuracy, improved robustness, and generalizability.

As shown in Fig. 3, using PEFT, the parameters of the foundational LLMs are frozen and remain fixed during training and evaluation, while additional parameters are injected for customization. Hence, only these injected parameters are tuned at local clients and aggregated globally. Using SFT, on the other hand, all the parameters of the LLMs are fine-tuned and communicated for aggregation.

As SFT fine-tunes the entire network, the whole model must be transferred and aggregated. This transmission challenge must be properly addressed to enable SFT with recent LLMs in FL using NVFlare’s data streaming API (see Section 2.4).

3.3 Federated Protein Embeddings and Task Model Fitting

Next, we explore obtaining protein-learned representations in the form of embeddings using an ESM-style pre-trained model [23]. The model is trained with NVIDIA’s BioNeMo framework⁴ for LLM training and inference.

Using BioNeMo, users can obtain numerical vector representations of protein sequences called embeddings. Protein embeddings

can then be used for visualization or making downstream predictions.

Here, we are interested in training a neural network to predict subcellular location from an embedding.

Subcellular Location Prediction. The data we will be using comes from the work by Stärk et al. [34]. In this paper, the authors developed a machine learning algorithm to predict the subcellular location of proteins from sequence through protein language models that are similar to those hosted by BioNeMo. Protein subcellular location refers to where the protein localizes in the cell; for example, a protein may be expressed in the ‘Nucleus’ or the ‘Cytoplasm.’ Knowing where proteins localize can provide insights into the underlying mechanisms of cellular processes and help identify potential targets for drug development. Figure 4 includes a few examples of subcellular locations in an animal cell.

We will utilize FASTA⁵ sequences for our target input sequences in a benchmark dataset called “Fitness Landscape Inference for Proteins” (FLIP) [5]. FLIP encompasses experimental data across adeno-associated virus stability for gene therapy, protein domain B1 stability and immunoglobulin binding, and thermostability from multiple protein families.

Model Architecture. We utilize the *ESM-1nv* model developed using the BioNeMo framework. The model uses an architecture called “Bidirectional Encoder Representations from Transformers” (BERT) and is based on the ESM-1 model [7, 27]. Pre-norm layer normalization and GELU [14] activation are used throughout. The model has six layers, 12 attention heads, a hidden space dimension of 768, and contains 44M parameters. The input sequence length is limited to 512 amino acids.

In Section 4.4, we show results for running federated inference to extract protein embeddings from the clients’ local data, followed by the application of FedAvg to training an MLP for the downstream subcellular location prediction task.

⁴<https://www.nvidia.com/en-us/clara/bionemo>

⁵https://en.wikipedia.org/wiki/FASTA_format

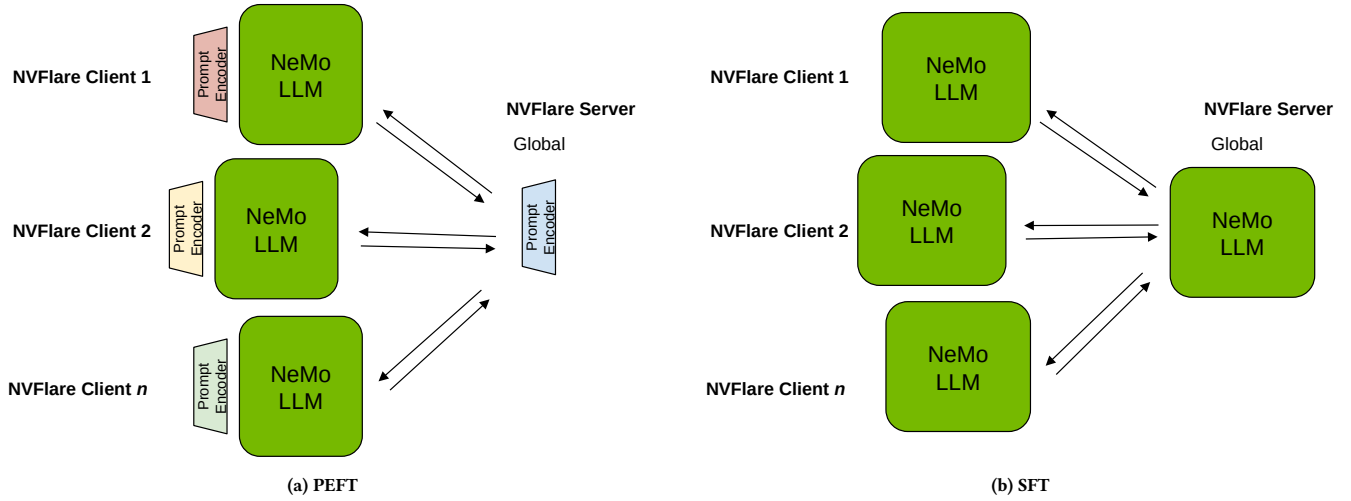


Figure 3: Federated parameter-efficient fine-tuning (PEFT) and full supervised fine-tuning (SFT) with global model and n clients.

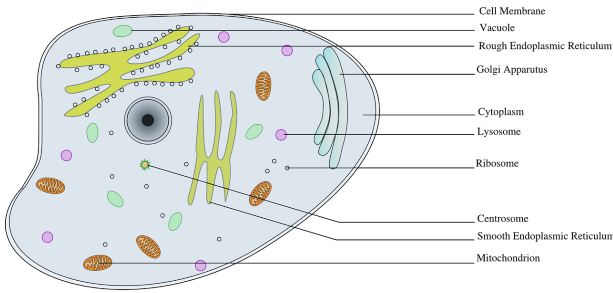


Figure 4: Cross section of an animal cell [25].

4 RESULTS

4.1 Large message streaming

We have extensively tested the streaming feature across regions and cloud providers, including AWS and Azure, to test the transfer of large model sizes. This example used a randomly initialized model consisting of a dictionary of 64 keys. Each key held a 2GB floating-point number array (resulting in a total model size of 128GB). This message size is much larger than commonly used modern LLMs, such as the *LLama-2* 7B parameter model⁶ [37], which requires checkpoint sizes of 13.5GB to store all its parameters. Even large models such as *CodeLlama* 70B parameter model⁷ [30], which takes ~ 140 GB to store their checkpoints, are similar to our experiment.

In the following, we are using two clients: *Site-1* with a fast connection and *Site-2* with a slower one⁸. Figure 5 illustrates the server’s and clients’ memory usage during the FL training run over three rounds.

⁶<https://huggingface.co/meta-llama/Llama-2-7b>

⁷<https://huggingface.co/codellama/CodeLlama-70b-hf>

⁸The server was deployed on Azure with 900GB RAM. Each client had ~ 380 GB RAM.

The local training task was to add a small number to those arrays. The aggregator on the server side was not changed. This job required at least two clients and ran three rounds to finish. During the experiment, the server used over 512GB, i.e., $128\text{GB} \times 2$ (clients) $\times 2$ (model and runtime space). Although most of the time, the server was using less than 512GB, there were a few peaks that reached 700GB or more.

The *Site-1* client, with its fast bandwidth connection with the server, received and sent the models in about 100 minutes and entered a nearly idle state with little CPU and memory usage after the communication ended. Both clients used about 256GB, i.e., $128\text{GB} \times 2$ (for model and runtime space), but at the end of receiving large models and at the beginning of sending large models, these two clients required more than 378GB, i.e., $128\text{GB} \times 3$.

4.2 Federated PEFT Performance

For PEFT, we utilize NeMo’s PEFT methods. With a single line of configuration change, you can experiment with various PEFT techniques, such as p-tuning [20], adapters [15], or Low-Rank Adaptation (LoRA) [16], all of which introduce a small number of trainable parameters to the LLM. These parameters condition the model to generate the desired output for the downstream task. These approaches minimize resource utilization by training a smaller subset of parameters compared to full-finetuning. We train a GPT Megatron model with 345 million parameters on a financial sentiment prediction task [21] using LoRA. In total, this data contains 1,800 pairs of headlines and corresponding sentiment labels. We use a Dirichlet sampling strategy [38] for creating a heterogeneous data partition among the clients. Examples of how the training data is distributed among the three clients using different values of α are shown in Fig. 6.

Figure 7 shows examples of how the training data is distributed among the three clients when using different α values. The lines

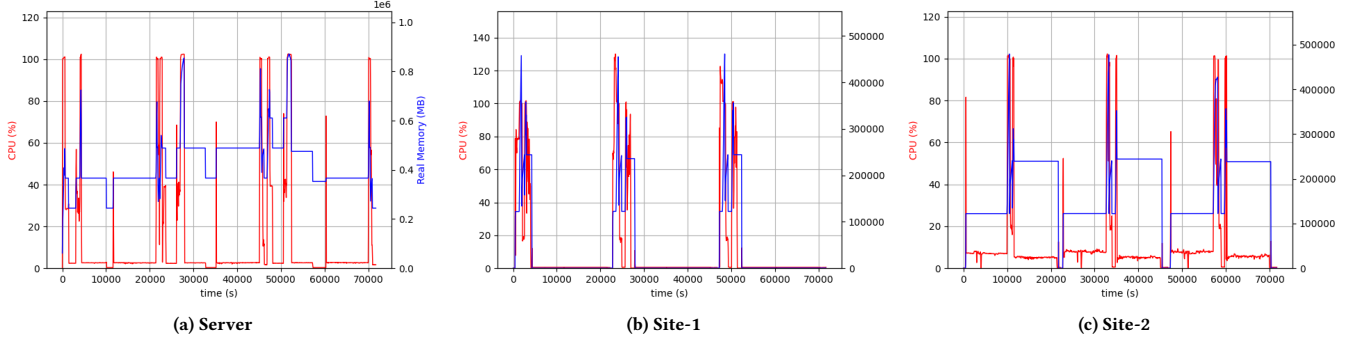


Figure 5: Memory usage during streaming of a 128GB large model.

show the mean accuracy of local models during training, and shaded areas indicate the 95% confidence interval.

Example input headlines from the financial sentiment prediction tasks and the predictions from the trained global model are shown in **bold**:

- The products have a low salt and fat content.
sentiment: neutral
- The agreement is valid for four years.
sentiment: neutral
- Diluted EPS rose to EUR3 .68 from EUR0 .50.
sentiment: positive
- The company is well positioned in Brazil and Uruguay.
sentiment: positive
- Profit before taxes decreased by 9% to EUR 187.8 mn in the first nine months of 2008, compared to EUR 207.1 mn a year earlier.
sentiment: negative

4.3 Federated SFT Performance

For SFT, we conducted experiments using the *nemo-megatron-gpt-1.3B*⁹ model, SFT for five rounds, training on three open datasets Alpaca¹⁰ [36], databricks-dolly-15k¹¹ [4], OpenAssistant Conversations¹² [18], one for each client.

Figure 8 illustrates the validation curves under all experiment settings: local-only training on each of the three datasets, on a combined dataset, and federated learning with all three clients training together using the FedAvg algorithm. Smooth curves represent local training, while “step curves”, identified by red dots, are for FL - the “steps” are due to global model aggregation and update at the beginning of each FL round.

Evaluating LLMs can be a non-trivial task. Following popular benchmark tasks, we perform three language modeling tasks under zero-shot settings, including HellaSwag (H) [42], PIQA (P) [1], and WinoGrande (W) [31]. Table 1 shows the results of each SFT model, with “BaseModel” representing the model before SFT. We utilize both “unnormalized” ($*_{acc}$) and “normalized” ($*_{\widehat{acc}}$) metrics and compute their mean for overall evaluation [9]. As shown, FL can

Table 1: Model performance on three benchmark tasks: HellaSwag (H), PIQA (P), and WinoGrande (W).

	H_{acc}	$H_{\widehat{acc}}$	P_{acc}	$P_{\widehat{acc}}$	W_{acc}	Mean
BaseModel	0.357	0.439	0.683	0.689	0.537	0.541
Alpaca	0.372	0.451	0.675	0.687	0.550	0.547
Dolly	0.376	0.474	0.671	0.667	0.529	0.543
Oasst1	0.370	0.452	0.657	0.655	0.506	0.528
Combined	0.370	0.453	0.685	0.690	0.548	0.549
FedAvg	0.377	0.469	0.688	0.687	0.560	0.556

help achieve the best overall performance compared to the models fine-tuned on the individual datasets by effectively combining updates from diverse sources without having to centralize the data as in the “Combined” setting.

4.4 Subcellular Structure Prediction

First, we run federated inference of the ESM-1nv model to extract embeddings, which requires a GPU with at least 12GB memory.

Next, we want to classify proteins for their subcellular location. Hence, we train a simple *scikit-learn* [24] Multi-layer Perceptron (MLP) classifier on top of the extracted ESM-1nv features using FedAvg. The MLP model uses a network of hidden layers to fit the input embedding vectors to the model classes (the cellular locations above). The results in Fig. 9 show the local (clients train on their local data alone) and global (using FedAvg) model performances varying the number of 32 hidden units of the MLP from one layer with 32 hidden units to four layers containing 512, 256, 128, and 64 units, respectively.

As the MLP parameters increase, the local models tend to overfit to the training data, while the FL models can benefit from the larger effective training set sizes and perform well on the test sets.

5 CONCLUSION

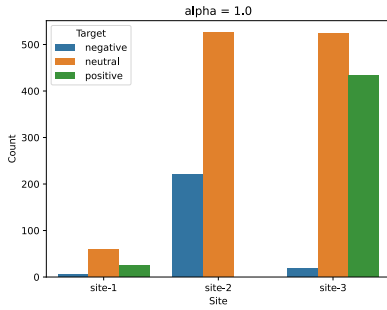
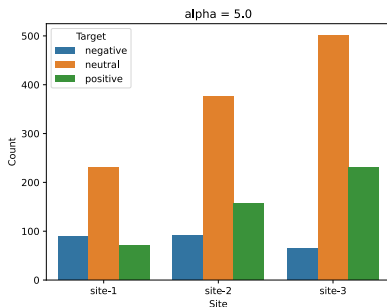
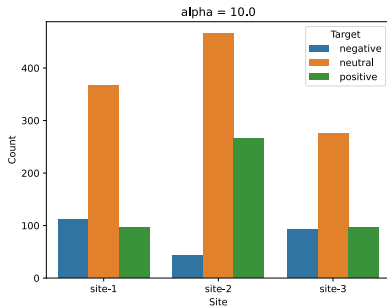
FL presents exciting opportunities for customizing foundational LLMs and tackling data challenges while prioritizing privacy. Fine-tuning techniques, which aim to adapt foundational LLMs for specific domains and tasks, can be seamlessly applied within an FL

⁹<https://huggingface.co/nvidia/nemo-megatron-gpt-1.3B>

¹⁰<https://huggingface.co/datasets/tatsu-lab/alpaca>

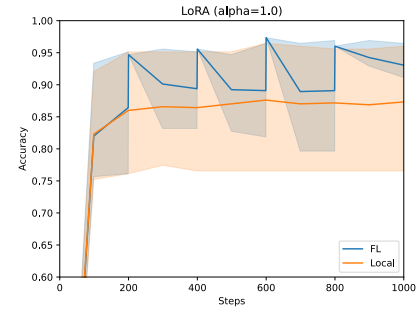
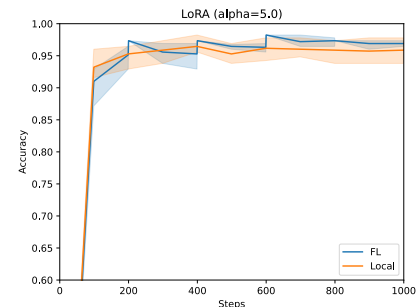
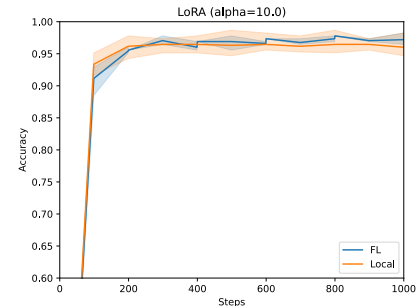
¹¹<https://huggingface.co/datasets/databricks/databricks-dolly-15k>

¹²<https://huggingface.co/datasets/OpenAssistant/oasst1>


 (a) $\alpha = 1.0$

 (b) $\alpha = 5.0$

 (c) $\alpha = 10.0$
Figure 6: Simulation of different data distributions among clients.

paradigm, leveraging the broader availability of diverse distributed datasets. NVFlare offers communication support to facilitate collaborative LLM training. These techniques, when combined with advancements in model development, pave the way for more adaptable and efficient LLMs.

This paper primarily highlights NVFlare’s new features, demonstrating its capability to simplify and scale the adaptation of LLMs with popular fine-tuning approaches like PEFT and SFT within FL, thereby supporting federated training of massive models. Two key features stand out: the *Client API* and the ability to stream large datasets.


 (a) $\alpha = 1.0$

 (b) $\alpha = 5.0$

 (c) $\alpha = 10.0$
Figure 7: PEFT accuracy curves on clients using their “Local” data alone versus the accuracy when training a joint model using “FL” that can learn from the data available at all sites without having to centralize the data.

Utilizing the *Client API*, it becomes straightforward to translate existing code developed for centralized training into a federated scenario. It avoids the need for a major restructuring of the training code to fit certain client class structures as required by other FL frameworks.

The *Streaming API* enables the communication of arbitrary large message sizes, paving the way for enabling FL for massive models, such as modern LLMs.

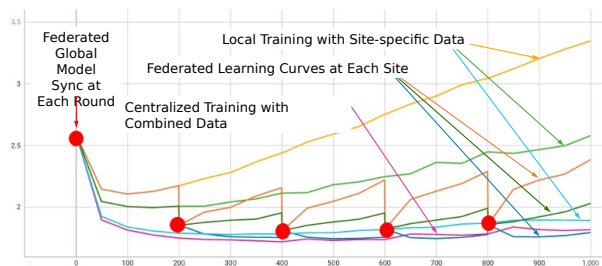
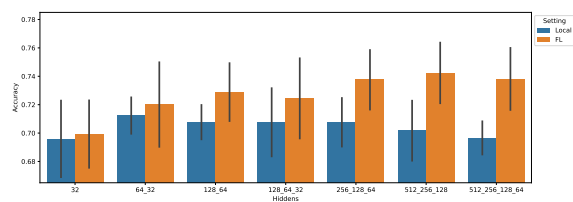


Figure 8: SFT validation loss curve.



- [35] J. Sun, Z. Xu, D. Yang, V. Nath, W. Li, C. Zhao, D. Xu, Y. Chen, and H. R. Roth. Communication-efficient vertical federated learning with limited overlapping samples. *arXiv preprint arXiv:2303.16270*, 2023.
- [36] R. Taori, I. Gulrajani, T. Zhang, Y. Dubois, X. Li, C. Guestrin, P. Liang, and T. B. Hashimoto. Stanford alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca, 2023.
- [37] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.
- [38] H. Wang, M. Yurochkin, Y. Sun, D. Papailiopoulos, and Y. Khazaeni. Federated learning with matched averaging. *arXiv preprint arXiv:2002.06440*, 2020.
- [39] P. Wang, C. Shen, W. Wang, M. Oda, C.-S. Fuh, K. Mori, and H. R. Roth. Conditstfl: Conditional distillation for federated learning from partially annotated data. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 311–321. Springer, 2023.
- [40] S. Warnat-Herresthal, H. Schultze, K. L. Shastry, S. Manamohan, S. Mukherjee, V. Garg, R. Sarveswara, K. Händler, P. Pickkers, N. A. Aziz, et al. Swarm learning for decentralized and confidential clinical machine learning. *Nature*, 594(7862):265–270, 2021.
- [41] A. Xu, W. Li, P. Guo, D. Yang, H. R. Roth, A. Hatamizadeh, C. Zhao, D. Xu, H. Huang, and Z. Xu. Closing the generalization gap of cross-silo federated medical image segmentation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 20866–20875, 2022.
- [42] R. Zellers, A. Holtzman, Y. Bisk, A. Farhadi, and Y. Choi. Hellaswag: Can a machine really finish your sentence? *arXiv preprint arXiv:1905.07830*, 2019.
- [43] C. Zhang, S. Li, J. Xia, W. Wang, F. Yan, and Y. Liu. Batchcrypt: Efficient homomorphic encryption for {Cross-Silo} federated learning. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 493–506, 2020.