Model Aggregation via Good-Enough Model Spaces

Anonymous Authors[1]

Abstract

In many applications, the training data for a machine learning task is partitioned across multiple nodes, and aggregating this data may be infeasible due to storage, communication, or privacy constraints. In this work, we present Good-Enough Model Spaces (GEMS), a novel framework for learning a global satisficing (i.e. "good-enough") model within a few communication rounds by carefully combining the space of local nodes' satisficing models. In experiments on benchmark and medical datasets, our approach outperforms other baseline aggregation techniques such as ensembling or model averaging, and performs comparably to the ideal non-distributed models.

1. Introduction

There has been significant work in designing distributed optimization methods in response to challenges arising from a wide range of large-scale learning applications. These methods typically aim to train a global model by performing numerous communication rounds between distributed nodes. However, most approaches treat communication reduction as an objective, not a constraint, and seek to minimize the number of communication rounds while maintaining model performance. Less explored is the inverse setting—where our communication budget is fixed and we aim to maximize accuracy while restricting communication to only a few rounds. These *few-shot* model aggregation methods are ideal when any of the following conditions holds:

- **Limited network infrastructure**: Distributed optimization methods typically require a connected network to support the collection of numerous learning updates. Such a network can be difficult to set up and maintain, especially in settings where devices may represent different organizational entities (e.g., a network of different hospitals).
- **Privacy and data ephemerality**: Privacy policies or regulations like GDPR may require nodes to periodically delete the raw local data. Few-shot methods enable learning an aggregate model in ephemeral settings, where a node may lose access to its raw data. Additionally, as fewer messages are sent between nodes, these methods have the potential to offer increased privacy benefits.
- **Extreme asynchronicity**: Even in settings where privacy is not a concern, messages from distributed nodes may be unevenly spaced and sporadically communicated over days, weeks, or even months (e.g., in the case of remote sensor networks or satellites). Few-shot methods drastically limit communication and thus reduce the wall-clock time required to learn an aggregate model.

Throughout this paper, we reference a simple motivating example. Consider two hospitals, $\boldsymbol{A}$ and $\boldsymbol{B}$, which each maintain private (unshareable) patient data pertinent to some disease. As $\boldsymbol{A}$ and $\boldsymbol{B}$ are geographically distant, the patients they serve sometimes exhibit different symptoms. Without sharing the raw training data, $\boldsymbol{A}$ and $\boldsymbol{B}$ would like to jointly learn a single model capable of generalizing to a wide range of patients. The prevalent learning paradigm in this setting—distributed or federated optimization—dictates that $\boldsymbol{A}$ and $\boldsymbol{B}$ share iterative model updates (e.g., gradient information) over a network.

From a meta-learning or multitask perspective, we can view each hospital (node) as a separate learning task, where our goal is to learn a single aggregate model which performs well on each task. However, these schemes often make similar assumptions on aggregating data and learning updates from different tasks.

As a promising alternative, we present *good-enough model spaces* (GEMS), a framework for learning an aggregate model over distributed nodes within a small number of communication rounds. Intuitively, the key idea in GEMS is to take advantage of the fact that many possible hypotheses may yield 'good enough' performance for a learning task on local data, and that considering the intersection between these sets can allow us to compute a global model quickly and easily. Our proposed approach has several advantages. First, it is simple and interpretable in that each node only communicates its locally optimal model and a small amount of metadata corresponding to local performance. Second, each node's message scales linearly in the local model size. Finally, GEMS is modular, allowing the operator to tradeoff the aggregate model's size against its performance via a hyperparameter ϵ.

We make the following contributions in this work. First, we present a general formulation of the GEMS framework. Second, we offer a method for calculating the good-enough space on each node as a $\mathbb{R}^d$ ball. We empirically validate GEMS on both standard benchmarks (MNIST and CIFAR-10) as well as a domain-specific health dataset. We consider learning convex classifiers and neural networks in standard distributed setups as well as scenarios in which some small global held-out data may be used for fine-tuning. We find that on average, GEMS increases the accuracy of local baselines by 10.1 points and comes within 43% of the (unachievable) global ideal. With fine-tuning, GEMS increases the accuracy of local baselines by 41.3 points and comes within 86% of the global ideal.

2. Related Work

Distributed Learning. Current distributed and federated learning approaches typically rely on iterative optimization techniques to learn a global model, continually communicating updates between nodes until convergence is reached. To improve the overall runtime, a key goal in most distributed learning methods is to minimize communication for some fixed model performance; to this end, numerous methods have been proposed for communication-efficient and asynchronous distributed optimization (e.g., Dekel et al., 2012; Recht et al., 2011; Dean et al., 2012; Li et al., 2014; Shamir et al., 2014; Richtárik & Takáč, 2016; Smith et al., 2018; McMahan et al., 2017). In this work, our goal is instead to maximize performance for a fixed communication budget (e.g., only one or possibly a few rounds of communication).

One-shot/Few-shot Methods. While simple one-shot distributed communication schemes, such as model averaging, have been explored in convex settings (Mcdonald et al., 2009; Zinkevich et al., 2010; Zhang et al., 2012; Shamir et al., 2014; Arjevani & Shamir, 2015), guarantees typically rely on data being partitioned in an IID manner and over a small number of nodes relative to the total number of samples. Averaging can also perform arbitrarily poorly in non-convex settings, particularly when the local models converge to differing local optima (Sun et al., 2017; McMahan et al., 2017). Other one-shot schemes leverage ensemble methods, where an ensemble is constructed from models trained on distinct partitions of the data (Chawla et al., 2004; Mcdonald et al., 2009; Sun et al., 2017). While these ensembles can often yield good performance in terms of accuracy, a concern is that the resulting ensemble size can become quite large. In Section 4, we compare against these one-shot baselines empirically, and find in that GEMs can outperform both simple averaging and ensembles methods while requiring significantly fewer parameters.

Meta-learning and transfer learning. The goals of meta-learning and transfer learning are seemingly related, as these works aim to share knowledge from one learning process onto others. However, in the case of transfer learning, methods are typically concerned with one-way transfer—i.e., optimizing the performance of a single target model, not jointly aggregate knowledge between multiple models. In meta-learning, such joint optimization is performed, but similar to traditional distributed optimization methods, it is assumed that these models can be updated in an iterative fashion, with potentially numerous rounds of communication being performed throughout the training process.

Version Spaces. In developing GEMS, we draw inspiration from work in *version space learning*, an approach for characterizing the set of logical hypotheses consistent with available data (Mitchell, 1978). Similar to (Balcan et al., 2012), we observe that if each node communicates its version space to the central server, the server can return a consistent hypothesis in the intersection of all node version spaces. However, (Mitchell, 1978; Balcan et al., 2012) assume that the hypotheses of interest are consistent with the observed data—i.e., they perfectly predict the correct outcomes. Our approach significantly generalizes to explore imperfect, noisy hypotheses spaces as more commonly observed in practice.

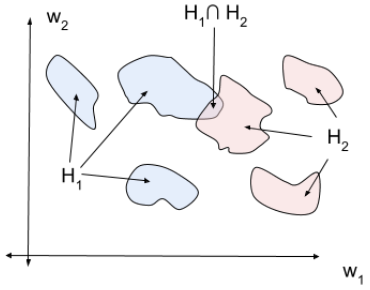
3. Methodology

As in traditional distributed learning, we assume a training set $S = \{(x_i, y_i)\}_{i=1}^m$ drawn from $\mathcal{D}_{\mathcal{X}\times\mathcal{Y}}$ is arbitrarily divided amongst K nodes. We define $S^k := \{(x_1^k, y_1^k), ...\}$ as the subset of training examples belonging to node k, such that $\sum_{k=1}^K |S^k| = m$. We assume that a single node (e.g., a central server) can aggregate updates communicated in the network. Fixing a function class $\mathcal{H}$, our goal is to learn an aggregate model $h_G \in \mathcal{H}$ that approximates the performance of the optimal model $h^* \in \mathcal{H}$ over S while limiting communication to one (or possibly a few) rounds of communication.

In developing a method for model aggregation, our intuition is that the aggregate model should be at least *good-enough* over each node's local data, i.e., it should achieve some minimum performance for the task at hand. Thus, we can compute h_G by having each node compute and communicate a *set* of locally good-enough models to a central server, which learns h_G from the intersection of these sets.

Formally, let $Q : (\mathcal{H}, \{(x_i, y_i)\}^d) \rightarrow \{-1, 1\}$ denote a *model evaluation function*, which determines whether a given model h is good-enough over a sample of data points $\{(x_i, y_i)\}^d \subseteq S$. In this work, define "good-enough" in

Figure 1. Illustration of good-enough model spaces.



terms of the accuracy of h and a threshold ϵ:

$$Q(h, \{(x_i, y_i)\}^d) = \begin{cases} 1 & \frac{1}{d}\sum_{i=1}^{d} \mathbb{I}\{h(x_i) = y_i\} \geq \epsilon \\ -1 & \text{else} \end{cases} \tag{1}$$

Using these model evaluation functions, we formalize the proposed approach for model aggregation, GEMS, in Algorithm 1. In GEMS, each node $k = 1, ..., K$ computes the set of models $H_k = \{h_1, ..., h_n | h_i \in \mathcal{H}, Q_k(h_i, S^k) = 1\}$ and sends it to the central node. After collecting $H_1, ...H_K$, the central node selects h_G from the intersection of the sets, $\cap_i H_i$. When granted access to a small sample of public data, the server can additionally use this auxiliary data further fine-tune the selected $h \in \cap_i H_i$, an approach we discuss further below.

Figure 3 visualizes this approach for a model class with only two weights (w_1 and w_2) and two learners ("red" and "blue"). The 'good-enough' model space, H_k, for each learner is a set of regions over the weight space (the blue regions correspond to one learner and the red regions correspond to second learner). The final aggregate model, h_G, is selected from the area in which the spaces intersect.

For a fixed hypothesis class $\mathcal{H}$, applying Algorithm 1 requires two components: (i) a mechanism for computing H_k over every node, and (ii) a mechanism for identifying the aggregate model, $h_G \in \cap_k H_k$. In this work, we present methods for two types of models: convex models and simple neural networks. For convex models, we find that H_k can be approximated as $\mathbb{R}^d$-ball in the parameter space, requiring only a single round of communication between nodes to learn h_G. For neural networks, we apply Algorithm 1 to each layer in a step-wise fashion, compute H_k as a set of independent $\mathbb{R}^d$-balls corresponding to every neuron in the layer, and identify intersections between different neurons. This requires one round of communication per layer (a few rounds for the entire network).

We can compute these $\mathbb{R}^d$ balls by fixing the center at the optimal local model on a device. The radius for the ball is computed via binary search: at each iteration, the node samples a candidate hypothesis h and evaluates $Q(h, S^k)$. The goal is to identify that largest radius such that all models located in the $\mathbb{R}^d$ ball are good-enough. Algorithm 2 presents a simple method for constructing H_k. More details can be found in Appendix A (convex setting) and Appendix B (neural network setting).

Algorithm 1 GEMS Meta-Algorithm

Input: $S = \{(x_i, y_i)\}_{i=1}^m$
for $k = 1, \cdots, m$ in parallel **do**
  Node k computes good-enough model space, H_k, according to (1)
end for
Return intersection $h_G \in \cap_k H_k$

Algorithm 2 ConstructBall

Input: k, $f_w(\cdot)$, $Q(\cdot)$,w_k^* $S^k = \{(x_1^k, y_1^k), ..\}$, R_{max}, Δ
Sets c_k to w_k^*.
Initialize $R_{\text{lower}} = 0$, $R_{\text{upper}} = R_{\text{max}}$
while $R_{\text{upper}} - R_{\text{lower}} > \Delta$ **do**
  Set $R = \text{Avg}(R_{\text{upper}}, R_{\text{lower}})$
  Sample $w_1, ..., w_p$ from surface of $\mathbb{B}_R(c_k)$
  if $Q(f_{w'}, S^k) = 1, \forall w' = w_1, ..., w_p'$ **then**
    Set $R_{\text{lower}} = R$
  else
    Set $R_{\text{upper}} = R$
  end if
end while
Return H_k

Fine-tuning. In many contexts, a small sample of public data S_{public} may be available to the central server. This could correspond to a public research dataset, or devices which have waived their privacy right. The coordinating server can fine-tune H_G on S_{public} by updating the weights for a small number of epochs. We find that fine-tuning is particularly useful for improving the quality of the GEMS aggregate model, H_G, compared to other baselines.

4. Evaluation

We now present the evaluation results for GEMS on three datasets: MNIST (LeCun et al., 1998), CIFAR-10 (Krizhevsky & Hinton, 2009), and HAM10000 (Tschandl et al., 2018), a medical imaging dataset. HAM10000 (HAM) consists of images of skin lesions, and our model is tasked with distinguishing between 7 types of lesions. Full details can be found in Appendix C.1. We focus on the performance of GEMS for neural networks, and discuss results for convex models in Appendix A.

Table 1. NN Results

Dataset	K	Global	Local	Averaged	GEMS	GEMS Tuned
MNIST	5	0.965 (0.000)	0.199 (0.010)	0.259 (0.039)	**0.439 (0.044)**	0.886 (0.007)
CIFAR-10	5	0.651 (0.004)	0.183 (0.009)	0.128 (0.023)	**0.223 (0.011)**	0.502 (0.011)
HAM	3	0.601 (0.002)	**0.271 (0.061)**	0.195 (0.042)	0.269 (0.089)	0.525 (0.014)

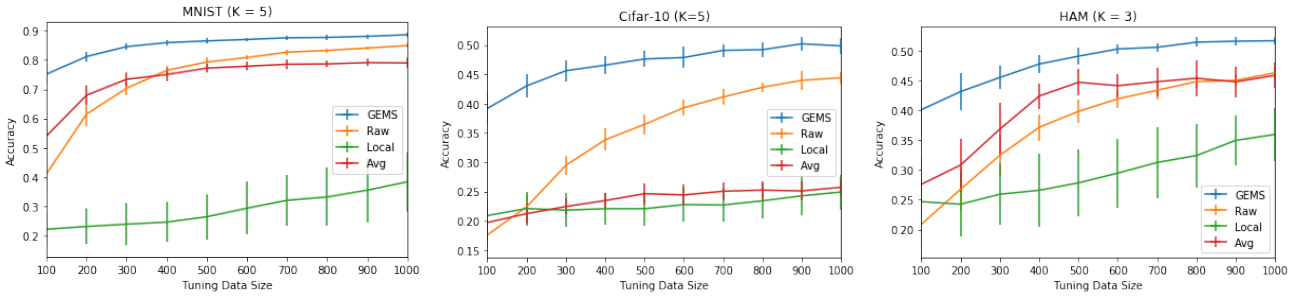


Figure 2. Comparative effects of fine-tuning for GEMS vs Baselines (Neural Network)

We partitioned data by label, such that all train/validation images corresponding to a particular label would be assigned to the same node. We consider three baselines: 1) *global*, a model trained on data aggregated across all nodes, 2) *local*, the average performance of models trained locally on each node, and 3) *naive average*, a parameter-wise average of all local models. All results are reported on the aggregated test set consisting of all test data across all nodes. Fine-tuning consists of updating the last layer's weights of the GEMS model for 5 epochs over a random sample of 1000 images from the aggregated validation data. We report the average accuracy (and standard deviation) of all results over 5 trials.

Neural network performance. We evaluated the neural network variant of GEMS on simple two layer feedforward neural networks (Table 1). The precise network configuration and training details are outlined in Appendix C.4. In the majority of cases, the untuned GEMS model outperforms the local/average baselines. Moreover, fine-tuning has a significant impact, and tuned GEMS model 1) significantly outperforms every baseline, and 2) does not degrade as K increases. In Appendix F, we demonstrate that GEMS is more parameter efficient than ensemble baselines, delivering better accuracy with fewer parameters.

Fine-tuning. The results in Table 1 suggest that fine-tuning on a holdout set of samples S_{public} has a significant effect on the GEMS model. We evaluate the effect of fine-tuning as the number of public data samples (the size of the tuning set) changes. For neural networks (Figure 2), finetuned GEMS consistently outperforms 1) the finetuned baselines, and 2) a 'raw' model trained directly on S_{public}. This suggest that the GEMS model is learning weights that are more amenable to fine-tuning, and are perhaps capturing better representations for the overall task. Though this advantage diminishes as the tuning sample size increases, the advantage of GEMS is especially pronounced for smaller samples, and achieves remarkable improvements with just 100 images.

Intersection Analysis. In certain cases, GEMS may not find an intersection between different nodes. This occurs when the task is too complex for the model, or ϵ is set too high. In practice, we notice that finding an intersection requires us to be conservative (e.g low values) when setting ϵ for each node. We explain this by our choice to represent H_k as an $\mathbb{R}^d$ ball. Though $\mathbb{R}^d$ balls are easy to compute and intersect, they're fairly coarse approximations of the actual good-enough model space. To illustrate node behavior at different settings of ϵ, we defer the reader to experiments performed in Appendix G.

5. Conclusion

In summary, we introduce GEMS, a framework for learning an aggregated model across different nodes within a few rounds of communication. We validate one approach for constructing good-enough model spaces (as $\mathbb{R}^d$ balls) on three datasets for both convex classifiers and simple feed-forward networks. Despite the simplicity of the proposed approach, we find that it outperforms a wide range of baselines for effective model aggregation.

References

Arjevani, Y. and Shamir, O. Communication complexity of distributed convex learning and optimization. In *Advances in neural information processing systems*, pp.

1756–1764, 2015.

Balcan, M. F., Blum, A., Fine, S., and Mansour, Y. Distributed learning, communication complexity and privacy. In *Conference on Learning Theory*, pp. 26–1, 2012.

Chawla, N. V., Hall, L. O., Bowyer, K. W., and Kegelmeyer, W. P. Learning ensembles from bites: A scalable and accurate approach. *Journal of Machine Learning Research*, 5(Apr):421–451, 2004.

Chollet, F. et al. Keras. `https://keras.io`, 2015.

Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Le, Q. V., Mao, M., Ranzato, M., Senior, A., Tucker, P., Yang, K., and Ng, A. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, pp. 1223–1231, 2012.

Dekel, O., Gilad-Bachrach, R., Shamir, O., and Xiao, L. Optimal Distributed Online Prediction Using Mini-Batches. *Journal of Machine Learning Research*, 13:165–202, 2012.

Krizhevsky, A. and Hinton, G. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

LeCun, Y., Bottou, L., Bengio, Y., Haffner, P., et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Li, M., Andersen, D. G., Smola, A. J., and Yu, K. Communication efficient distributed machine learning with the parameter server. In *Advances in Neural Information Processing Systems*, pp. 19–27, 2014.

Mcdonald, R., Mohri, M., Silberman, N., Walker, D., and Mann, G. S. Efficient large-scale distributed training of conditional maximum entropy models. In *Advances in Neural Information Processing Systems*, pp. 1231–1239, 2009.

McMahan, B., Moore, E., Ramage, D., Hampson, S., and y Arcas, B. A. Communication-efficient learning of deep networks from decentralized data. In *AISTATS*, 2017.

Mitchell, T. M. Version spaces: an approach to concept learning. Technical report, STANFORD UNIV CALIF DEPT OF COMPUTER SCIENCE, 1978.

Recht, B., Re, C., Wright, S., and Niu, F. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pp. 693–701, 2011.

Richtárik, P. and Takáč, M. Distributed coordinate descent method for learning with big data. *Journal of Machine Learning Research*, 17:1–25, 2016.

Shamir, O., Srebro, N., and Zhang, T. Communication-efficient distributed optimization using an approximate newton-type method. In *International conference on machine learning*, pp. 1000–1008, 2014.

Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Smith, V., Forte, S., Ma, C., Takac, M., Jordan, M. I., and Jaggi, M. Cocoa: A general framework for communication-efficient distributed optimization. *Journal of Machine Learning Research*, 18:1–47, 2018.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

Sun, S., Chen, W., Bian, J., Liu, X., and Liu, T.-Y. Ensemble-compression: A new method for parallel training of deep neural networks. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pp. 187–202. Springer, 2017.

Tschandl, P., Rosendahl, C., and Kittler, H. The HAM10000 dataset: A large collection of multi-source dermatoscopic images of common pigmented skin lesions. *CoRR*, abs/1803.10417, 2018. URL `http://arxiv.org/abs/1803.10417`.

Zhang, Y., Wainwright, M., and Duchi, J. Communication-efficient algorithms for statistical optimization. In *Neural Information Processing Systems*, 2012.

Zinkevich, M., Weimer, M., Li, L., and Smola, A. J. Parallelized stochastic gradient descent. In *Advances in neural information processing systems*, pp. 2595–2603, 2010.

A. GEMS: Convex

We provide a more detailed explanation of the GEMS algorithm for convex settings.

Consider the class of linear separators $f_w(\cdot)$ parameterized by a weight vector $w \in \mathbb{R}^d$. For each node k, we compute H_k as $\mathbb{R}^d$-ball in the parameter space, represented as a tuple $(c_k \in \mathbb{R}^d, r_k \in \mathbb{R})$ corresponding to the center and radius. Formally, $H_k = \{w \in \mathbb{R}^d | ||c_k - w||_2 \leq r_k\}$. Fixing ϵ as our minimum acceptable performance, we want to compute H_k such that $\forall w \in H_k$, $Q(w, S^k) = 1$. Intuitively, every model contained within the d-ball should have an accuracy greater than or equal to ϵ.

Construction: Algorithm 2 presents the H_k construction algorithm for node k, where

$$w_k^* = \arg\min_w \frac{1}{|S^k|} \sum_{i=1}^{|S^k|} \ell(f_w(x_i), y_i)$$

over node data S^k, ϵ is fixed hyperparameter, and $Q(\cdot)$ is a minimum accuracy threshold defined according to Eq. 1. R_{max} and Δ define the scope and stopping criteria for the binary search.

Intersection: Given K nodes with individual $H_i = (c_i, r_i)$, then $\cap_i H_i = \{w \in \mathbb{R}^d \mid ||c_i - w||_2 \leq r_i, \forall i = 1, .., K\}$. We pick a point in this intersection by solving:

$$h_G = \arg\min_w \sum_{i=1}^{K} \max(0, ||c_i - w||_2 - r_i) \qquad (2)$$

which takes a minimum value of 0 when $w \in \cap_i H_i$. This w can be improved by fine-tuning on a limited sample of 'public data'.

B. GEMS: Neural Networks

We provide a more detailed explanation of the GEMS algorithm applied to neural networks.

First, we observe that the final layer of an MLP is a linear model. Hence, we can apply the method above with no modification. However, the input to this final layer is a set of stacked, non-linear transformations which extract feature from the data. For these layers, the approach presented above faces two challenges:

1. **Node specific features**: When the distribution of data is non-i.i.d across nodes, different nodes may learn different feature extractors in lower layers.
2. **Model Isomorphisms**: MLPs are extremely sensitive to the weight initialization. Two models trained on the same set of samples (with different initializations) may have equivalent behavior despite learning weights. In particular, reordering a model's hidden neurons (within the same layer) does not alter the model's predictions, but corresponds to a different weight vector w.

In order to construct H^k for hidden layers, we modify the approach presented in Appendix A, applying it to individual hidden units. Formally, let the ordered set $[f^j_{w_1}(\cdot), ..., [f^j_{w_L}(\cdot)]$ correspond to the set of L hidden neurons in layer j. Here, $f^j_{w_l}(\cdot) = g(w_l^T z^{j-1})$ denotes the function computed by the l-th neuron over the output from the previous layer z^{j-1}, with $g(\cdot)$ corresponding to some non-linearity (i.e. ReLU). Fixing an indexed ordering over d data points, let $z_l^j = [(z_l^j)_1, ..., (z_l^j)_d]$ denote the vector of activations produced by $f^j_{w_l}(\cdot)$. We can define an alternative Q over a neuron, in terms of z^{j-1} and z_l^j (the neuron's input and output):

$$Q_{\text{neuron}}(w', \{((z^{j-1})_i, (z_l^j)_i)\}^d) = \mathbb{I}\{\sqrt{\sum_{i=1}^{d} \left(f^j_{w'}(z^{j-1})_i) - (z_l^j)_i\right)^2} \leq \epsilon_j\} \qquad (3)$$

Broadly, Q_{neuron} returns 1 if the output of $f_{w'}$ over z^{j-1} is within ϵ of z_l^j, and -1 otherwise. We can now apply Algorithm 2 to each neuron. Formally:

1. Each node k learns a locally optimal model m^k, with optimal neuron weights w_l^{j*}, over all j, l.
2. Fix hidden layer $j = 1$. Apply Algorithm 2 to each hidden neuron $[f^j_{w_1}(\cdot), ..., [f^j_{w_L}(\cdot)]$, with $Q(\cdot)$ according to Eq 3 and predefined hyperparameter ϵ_j. Denote the $\mathbb{R}^d$ ball constructed for neuron l as $H^k_{j,l}$.
3. Each node communicates its set of $H^k_{j,\cdot} = [H^k_{j,1}, ..., H^k_{j,L}]$ to the central server which constructs the aggregate hidden layer $f_{G_{j,\cdot}}$ such $\forall i, k, \exists i' : f_{G_{j,i'}} \in H^k_{j,i}$. This is achieved by greedily applying Eq 2 to tuples in the cartesian product $H^1_{j,\cdot} \times ... \times H^K_{j,\cdot}$. Neurons for which no intersection exists are included in $f_{G_{j,\cdot}}$, thus trivially ensuring the condition above.
4. The server sends $h_{G_{j,\cdot}}$ to each node, which insert $h_{G_{j,\cdot}}$ at layer j in their local models and retrain the layers above j.
5. Increment j, and return to (2) if any hidden layers remain.

Step (3) is expensive for large L and K as $|H^1_{j,\cdot} \times ... \times H^K_{j,\cdot}|$ increases exponentially. A simplifying assumption is that if $H^k_{j,i}$ and $H^k_{j,l}$ are 'far', then the likelihood of intersection is low. Operationalizing this, we can perform k-means clustering over all neurons. In step (3), we now only look for intersections between tuples of neurons in the same cluster. Neurons for which no intersection exists are included in $f_{G_{j,\cdot}}$. For notational clarity, we denote the number of clusters with which k-means is run as m_ϵ, in order to distinguish it from device index k.

C. Experimental Setup

C.1. Preprocessing

We describe preprocessing/featurization steps for our empirical results.

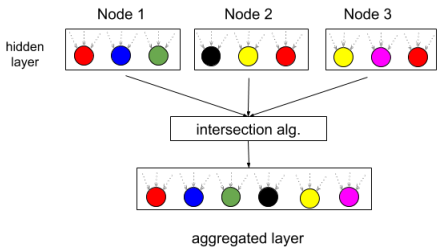


Figure 3. Illustration of how GEMS is used to construct a hidden layer for neural networks. In this example, neurons with overlapping good-enough spaces have the same color.

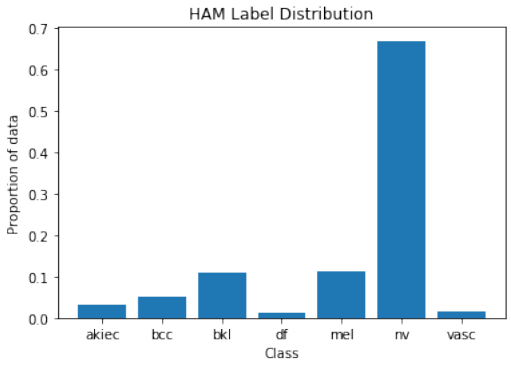


Figure 4. Distribution of classes for HAM

MNIST. We used the standard MNIST dataset.

CIFAR-10. We featurize CIFAR-10 (train, test, and validation sets) using a pretrained ImageNet VGG-16 model (Simonyan & Zisserman, 2014) from Keras. All models are learned on these featurized images.

HAM10000. The HAM dataset consists of 10015 images of skin lesions. Lesions are classified as one of seven potential types: actinic keratoses and intraepithelial carcinoma (akiec), basal cell carcinoma (bcc), benign keratosis (bkl), dermatofibroma (df), melanoma (mel), melanocytic nevi (nv), and vascular lesions (vasc). As Figure 4 shows, the original original dataset is highly skewed, with almost 66% of images belonging to one class. In order to balance the dataset, we augment each class by performing a series of random transformations (rotations, width shifts, height shifts, vertical flips, and horizontal flips) via Keras (Chollet et al., 2015). We sample 2000 images from each class. We initially experimented with extracting ImageNet features (similar to our proceedure for CIFAR-10). However, training a model on these extractions resulted in poor performance. We constructed our own feature extractor, by training a simple convolutional network on 66% of the data, and trimming the final 2 dense layers. This network contained 3 convolutional layers (32, 64, 128 filters with 3×3 kernels) interspersed with 2×2 MaxPool layers, and followed by a single hidden layer with 512 neurons.

C.2. Data Partitioning

Given K nodes, we partitioned each dataset in order to ensure that all images corresponding to the same class belonged to the same node. Table 2 provides an explicit breakdown of the label partitions for each of the three datasets, across the different values of K we experimented with.

We divided each dataset into train, validation, and test splits. All training occurs exclusively on the train split and all results are reported for performance on the test split. We use the validation split to construct each node's good-enough model space. We use a train/val/test split of 50000/5000/5000 for MNIST and CIFAR-10. For HAM, we use a 80/10/10 percentage split (since no conventional train/test partitioning exists).

Dataset	K	Label Division
MNIST	2	$[\{0,1,2,3,4\},\{0,1,2,3,4\}]$
MNIST	3	$[\{0,1,2\},\{3,4,5\},\{6,7,8,9\}]$
MNIST	5	$[\{0,1\},\{2,3\},\{4,5\},\{6,7\},\{8,9\}]$
CIFAR10	2	$[\{0,1,2,3,4\},\{0,1,2,3,4\}]$
CIFAR10	3	$[\{0,1,2\},\{3,4,5\},\{6,7,8,9\}]$
CIFAR10	5	$[\{0,1\},\{2,3\},\{4,5\},\{6,7\},\{8,9\}]$
HAM	2	$[\{0,1,2,3\},\{4,5,6\}]$
HAM	3	$[\{0,1\},\{2,3\},\{4,5,6\}]$

Table 2. Label Partitions across different K

C.3. Convex Model Training

Our convex model consists of a simple logistic regression classifier. We train with Adam, a learning rate of 0.001, and a batch size of 32. We terminate training when training accuracy converges.

C.4. Neural Network Model Training

Our non-convex model consists of a simple two layer feed-forward neural network. For MNIST and HAM, we fix the hidden layer size to 50 neurons. For CIFAR-10, we fix the hidden layer size to 100 neurons. We apply dropout (Srivastava et al., 2014) with a rate of 0.5 to the hidden layer. We train with Adam, a learning rate of 0.001, and a batch size of 32. We terminate training when training accuracy converges.

D. Convex Results

We evaluate the convex variant of GEMS on logistic classifiers. The results for all three datasets for a varying number nodes K is presented in Table 3. Fine-tuning consists of updating the weights of the GEMS model for 5 epochs over a random sample of 1000 images from the aggregated validation data. Training details are provided in Appendix C.3

In a convex setting, we find that GEMS frequently defaults to a weighted average of the parameters. Hence, the GEMS results closely mirror naive averaging. As the number of agents increases, both untuned GEMS and the baselines significantly decrease in performance. However, tuned GEMS remains relatively consistent, and outperforms all other baselines. We use $\epsilon = 0.70$ for MNIST, $\epsilon = 0.40$ for HAM, and $\epsilon = 0.20$ for CIFAR-10.

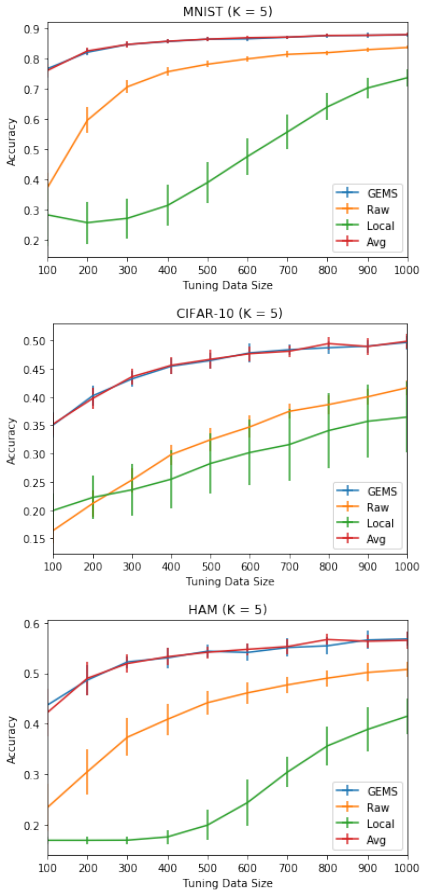


Figure 5. Comparative effects of fine-tuning for GEMS vs Baselines (Convex)

E. Neural Network Results

Table 4 presents the neural network results for MNIST. We use $\epsilon = 0.7$ for the final layer, and let ϵ_j denote the deviation allowed for the hidden neurons (as defined in Eq 3).

Table 5 presents the neural network results for CIFAR-10. We use $\epsilon = 0.2$ for the final layer.

Table 6 presents the neural network results for CIFAR-10. We use $\epsilon = 0.25$ for the final layer.

F. Ensemble Results

For neural networks, GEMS provides a modular framework to tradeoff between the model size and performance, via hyperparameters m_ϵ (the number of clusters created when identifying intersections) and ϵ_j (the maximum output deviation allowed for hidden neurons). Intuitively both parameters control the number of hidden neurons in the aggregate model h_G. Table 7 compares adjustments for ϵ_j and m_ϵ on CIFAR-10 for 5 nodes against an ensemble of local device models. We observe that the GEMS performance correlates with the number of hidden neurons, and that GEMS outperforms the ensemble method at all settings (despite having fewer parameters). For ease of clarity, we describe the model size in terms of the number of hidden neurons. For ensembles, we sum the hidden neurons across all ensemble members. All results are averaged over 5 trials, with standard deviations reported.

G. Intersection Analysis

We notice that in order for GEMs to find an intersection, we have to set ϵ conservatively. We illustrate this phenomenon in Figure 6. We consider the convex MNIST case ($K = 2$), and do a grid search over different values of ϵ for each node. We plot whether an intersection was identified, and the resulting accuracy at that setting.

Table 3. Convex Results

Dataset	K	Global	Local	Averaged	GEMS	GEMS Tuned
MNIST	2	0.926 (0.001)	0.481 (0.027)	**0.780 (0.015)**	**0.780 (0.015)**	0.889 (0.003)
MNIST	3	0.926 (0.001)	0.325 (0.042)	**0.705 (0.013)**	0.647 (0.033)	0.878 (0.004)
MNIST	5	0.925 (0.001)	0.198 (0.010)	**0.458 (0.038)**	**0.458 (0.038)**	0.880 (0.006)
CIFAR-10	2	0.585 (0.013)	**0.385 (0.025)**	0.253 (0.027)	0.234 (0.017)	0.494 (0.009)
CIFAR-10	3	0.590 (0.006)	**0.273 (0.062)**	0.193 (0.020)	0.193 (0.020)	0.491 (0.008)
CIFAR-10	5	0.591 (0.015)	**0.178 (0.010)**	0.150 (0.008)	0.150 (0.008)	0.500 (0.014)
HAM	2	0.559 (0.002)	0.344 (0.018)	**0.400 (0.020)**	0.353 (0.011)	0.491 (0.006)
HAM	3	0.559 (0.002)	0.263 (0.054)	**0.343 (0.012)**	**0.343 (0.012)**	0.483 (0.009)

Table 4. MNIST Results (Neural Network)

K	ϵ_{hidden}	m_ϵ	Global	Local	Averaged	GEMS	GEMS Tuned
2	0.01	1	0.964 (0.001)	0.492 (0.024)	0.641 (0.058)	**0.766 (0.083)**	0.888 (0.004)
3	1.0	100	0.965 (0.002)	0.329 (0.043)	0.422 (0.038)	**0.754 (0.024)**	0.926 (0.006)
5	1.0	100	0.965 (0.000)	0.199 (0.010)	0.259 (0.039)	**0.439 (0.044)**	0.886 (0.007)

Table 5. CIFAR-10 Results (Neural Network

K	ϵ_j	m_ϵ	Global	Local	Averaged	GEMS	GEMS Tuned
2	0.1	1.0	0.650 (0.004)	**0.405 (0.019)**	0.192 (0.026)	0.335 (0.041)	0.568 (0.007)
3	0.3	150	0.653 (0.004)	0.284 (0.061)	0.163 (0.029)	**0.333 (0.059)**	0.538 (0.009)
5	0.3	200	0.651 (0.004)	0.183 (0.009)	0.128 (0.023)	**0.223 (0.011)**	0.502 (0.011)

Table 6. HAM Results (Neural Network)

K	ϵ_j	m_ϵ	Global	Local	Averaged	GEMS	GEMS Tuned
2	0.01	1.0	0.594 (0.005)	0.354 (0.022)	0.273 (0.032)	**0.399 (0.039)**	0.539 (0.008)
3	0.07	100	0.601 (0.002)	**0.271 (0.061)**	0.195 (0.042)	0.269 (0.089)	0.525 (0.014)

Table 7. Model Size Results (MNIST, $K = 5$)

Method	Accuracy	# hidden neurons
Tuned GEMS ($m_\epsilon = 75$, $\epsilon_j = 0.5$)	0.872 (0.007)	74.00 (0.00)
Tuned GEMS ($m_\epsilon = 100$, $\epsilon_j = 1.0$)	0.886 (0.007)	99.00 (0.00)
Tuned GEMS ($m_\epsilon = 50$, $\epsilon_j = 1.0$)	0.862 (0.009)	49.0 (0.00)
Tuned GEMS ($m_\epsilon = 75$, $\epsilon_j = 1.0$)	0.867 (0.008)	79.00 (0.00)
Ensemble	0.210 (0.006)	250.00 (0.0)

Table 8. Model Size Results (CIFAR-10, $K = 5$)

Method	**Accuracy**	**# hidden neurons**
Tuned GEMS ($m_\epsilon = 150$, $\epsilon_j = 0.7$)	0.454 (0.018)	163.40 (1.20)
Tuned GEMS ($m_\epsilon = 150$, $\epsilon_j = 0.5$)	0.492 (0.012)	246.20 (8.93)
Tuned GEMS ($m_\epsilon = 200$, $\epsilon_j = 0.3$)	0.502 (0.011)	379.60 (6.68)
Tuned GEMS ($m_\epsilon = 100$, $\epsilon_j = 0.3$)	0.501 (0.011)	386.00 (18.76)
Ensemble	0.194 (0.005)	500.00 (0.0)

Table 9. Model Size Results (HAM, $K = 3$)

Method	**Accuracy**	**Num hidden**
Tuned GEMS ($m_\epsilon = 75$, $\epsilon_j = 0.07$)	0.517 (0.006)	76.00 (0.89)
Tuned GEMS ($m_\epsilon = 100$, $\epsilon_j = 0.07$)	0.525 (0.014)	100.20 (1.17)
Tuned GEMS ($m_\epsilon = 75$, $\epsilon_j = 0.03$)	0.521 (0.016)	114.60 (0.80)
Ensemble	0.245 (0.010)	150

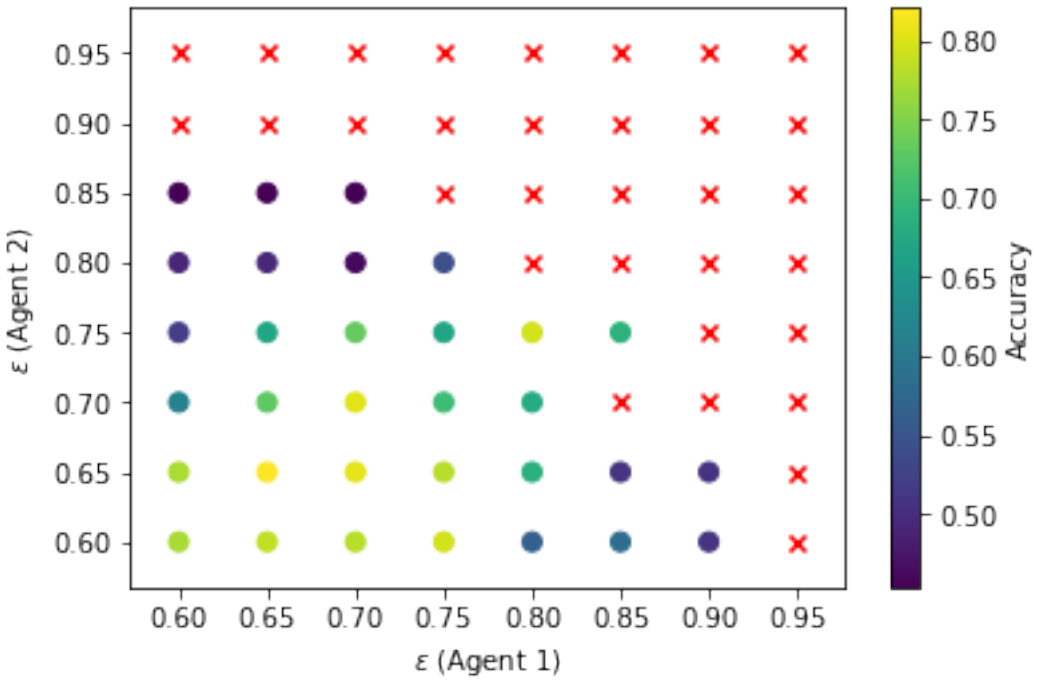


Figure 6. The x-axis corresponds to different settings of ϵ for 1 node, and the y-axis corresponds to different settings of ϵ for the 2nd node. Red crosses denote values where GEMS failed to find an intersection. The color of the circular markers denotes the accuracy of the intersected model.