Simple, Efficient and Convenient Decentralized Multi-Task Learning for Neural Networks

Amaury Bouchra Pilet, Davide Frey, François Taïani

▶ **To cite this version:**

Amaury Bouchra Pilet, Davide Frey, François Taïani. Simple, Efficient and Convenient Decentralized Multi-Task Learning for Neural Networks. IDA 2021 - 19th Symposium on Intelligent Data Analysis, Apr 2021, Porto, Portugal. ⟨10.1007/978-3-030-74251-5_4⟩. ⟨hal-02373338v6⟩

HAL Id: hal-02373338

https://hal.science/hal-02373338v6

Submitted on 25 Mar 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

Simple, Efficient and Convenient Decentralized Multi-Task Learning for Neural Networks

Amaury Bouchra Pilet, Davide Frey and François Taïani

5 Germinal 229

IRISA, Inria, Univ Rennes, CNRS

Abstract

Machine learning requires large amounts of data, which is increasingly distributed over many systems (user devices, independent storage systems). Unfortunately aggregating this data in one site for learning is not always practical, either because of network costs or privacy concerns. Decentralized machine learning holds the potential to address these concerns, but unfortunately, most approaches proposed so far for distributed learning with neural network are mono-task, and do not transfer easily to multi-tasks problems, for which users seek to solve related but distinct learning tasks and the few existing multi-task approaches have serious limitations. In this paper, we propose a novel learning method for neural networks that is *decentralized*, *multi-task*, and keeps users' data *local*. Our approach works with different learning algorithms, on various types of neural networks. We formally analyze the convergence of our method, and we evaluate its efficiency in different situations on various kind of neural networks, with different learning algorithms, thus demonstrating its benefits in terms of learning quality and convergence.

I Introduction

A critical requirement for machine learning is training data. In some cases, a great amount of data is available from different (potential) users of the system, but simply aggregating this data and using it for training is not always practical. The data might for instance be large and extremely distributed and collecting it may incur significant communication cost. Users may also be unwilling to share their data due to its potential sensitive nature, as is the case with private conversations, browsing histories, or health-related data [Che+17].

To address these issues, several works have proposed to share model-related information (such as gradients or model coefficients) rather than raw data [Rec+11; Kon+16; Bre+17]. These approaches are however typically mono-task, in the sense that all users are assumed to be solving the same ML task. Unfortunately, in a distributed setting, the problems that users want to solve may not be perfectly identical. Let us consider the example of speech recognition being performed on mobile device. At the user level, each has a different voice, and so the different devices do not perform exactly the same task. At the level of a country or region, language variants also impose variants between tasks. For example users from Quebec can clearly be separated from those from France. A decentralized or federated learning platform should therefore accommodate for both types of differences between tasks: at the level of single users and at the level of groups of users.

Multitask learning has been widely studied in a centralized setting [Rud17]. Some distributed solutions exist, but they are typically limited to either linear [OHJ12] or convex [Smi+16; Bel+18; ZBT19] optimization problems. As a result, they are typically not applicable to neural networks, in spite of their high versatility and general success in solving a broad range of machine-learning

problems. Some recent preprints have proposed approaches for multitask learning with neural networks in federated setups, but they have more or less significant limitations (no parallelism, lack of flexibility, no decentralization) [CB19; Jia+19; Ari+19].

In this paper, we introduce a novel effective solution for *decentralized multi-task learning with neural networks*, usable in decentralized or federated setups. Its novelty is twofold. First, to the best of our knowledge, it is the first to combine decentralization, the ability to address multiple tasks, and support for general neural networks. Second, it provides a flexible approach that can operate without knowing the nature of the differences between peers' tasks but can still benefit from such knowledge when available, as our experimental evaluation suggests.

Our approach relies on model averaging, already used in non-multi-tasks federated learning, but its key idea consists in making the averaging partial. First, it enables partial averaging of peer models by making it possible to average specifics subsets of model parameters. Second, it makes it possible to average those subsets only among peers that share similar tasks if this knowledge is available. Experimental results show that partial averaging outperforms both local learning and global averaging.

II The method

II.1 Base system model

We consider a federated/peer-to peer-setup where each local device (peer), p, has an individual dataset, D^p, and maintains a model consisting of a neural network, N^p. We use the notation $N^p.\text{train}(D^p)$ to mean that peer p trains its neural network N^p with some training algorithm on dataset D^p. Similarly, the notation $N^p.\text{test}(D^p)$ means that the network is tested using D^p, returning a success rate $\in \mathbb{R}^+$. Each peer seeks to maximize the success rate of its neural network when tested with its data provider.

Each neural network, N^p, is divided into *parts* (one or more) of neurons having identical input (with different weights), activation function (with different parameters) and with their outputs treated equivalently: $N^p = (N^p_0, ..., N^p_k)$[1]. The essential idea of a part is that neurons in a part can be treated equivalently and permuted without altering the computation. This corresponds, for example, to a fully connected layer, but not to a convolutional layer [LB95], which must be divided into several parts. Parts consist of several neurons and an associated activation function: $N^p_i = ((N^p_{i,0}, ..., N^p_{i,k}), f)$ (Theoretically this definition is actually equivalent to having specific activation functions for each neuron, since a real parameter can be used to index activation functions in a finite set of neurons)[2]. The activation function f is a real function with real parameters $\mathbb{R} \times \mathbb{R}^k \to \mathbb{R}$. Each neuron n(n is used instead of $N^p_{i,k}$ for readability) consists of a weight vector and a vector of activation-function parameters, both vectors are real-valued, $n = (\mathbf{w}, \mathbf{q}) \in \mathbb{R}^j \times \mathbb{R}^k$. $\mathbf{w}$ and $\mathbf{q}$ may be learned by any training method. Neurons take an input vector, $\mathbf{i} \in \mathbb{R}^k$, and return a real value, $f(\mathbf{i} \cdot \mathbf{w}, \mathbf{q})$.

The neurons of a part all have the same input (but some input coordinates may have a fixed 0 weight, as long as this is also the case for all coordinates averaged with them), which can be either the output of another part or the input of the network itself. For modeling, we consider that the input arrives at designated input parts, consisting of neurons which just reproduce the input and are not updated in training. Bias neurons, outputting a constant value, are not considered, since their behavior is equivalent to introducing a bias as activation function parameter. Since they have fixed parameters, input parts are implicitly excluded from the collaborative learning process, since they simply do not learn.

A peer's neural network may include parts that do not exist for all peer's networks.

[1] i, j and k are used as generic index variables. Unless explicitly stated otherwise, i, j and k from different formulas have not the same meaning

[2] $\#\mathbb{R} = \beth_1 > \beth_0 = \aleph_0 > k(\forall_{k\in\mathbb{N}})$, since the set of neurons is finite, it is not an issue to use a real index to define each neuron's activation function, even if $\#\mathbb{R}^{\mathbb{R}} = \beth_2 > \beth_1 = \#\mathbb{R}$ implies that a real value can not index the set of all real functions

II.2 General idea

We reuse the mechanism of model averaging proposed for federated learning [Bre+17], but adapt it to allow multi-task learning in a decentralized setup. More specifically, our intuition is that the model of similar but different tasks can be divided into different portions that represent different levels of "specialization" of each task. A simple application of this principle consists for instance in dividing the model of each task into a global section, shared by all tasks and used for collaborative learning, and a local section, specific to the task at hand. This is similar in a way to transfer learning [PMK91], albeit applied to a multi-task decentralized set-up.

A key property of our approach consists in not limiting this split to a pair of global/local sub-models. Since different peers involved in multi-task learning may have more or less similar functions to learn, we allow peers to have portions of their neural networks that are averaged with only a subset of other peers. The result is a partitioning of each peer's neural-network into several portions, each to be averaged with a specific subset of the peers. We refer to such portions as *slices* and to the model associated with a slice as a *partial model*.

Summing up, our system uses averaging to enable decentralized learning; but we make this averaging partial along two dimensions. First, peers average only limited portions of their neural networks. This causes peer to retain *local* parts of their models thereby enabling multi-task learning. Second, averaging operations do not necessarily involve all peers, rather a peer can average each of its slices with a different subset of peers. This makes it possible for peers that share similar tasks to share some parts of their models while further specializing others.

Unlike modern non-distributed multi-task learning approaches commonly referred to as "transfer learning" [Rud17], we do not constrain the task-specific local portions of a model to consists of its last few layers. Rather, we enable complete freedom in the definition of which parts of the neural network are averaged. Once slices have been defined, the decentralized learning process automatically captures the differences between the tasks associated with different peers. In general, the partitioning of a peer's neural network into slices can be completely independent of the nature or the number of tasks in the decentralized or federated learning environment. However, knowledge of the differences between tasks can enable further optimization, for example by defining that peers with similar tasks should average some of their slices together.

With our system, the differences between peers' respective tasks will automatically be integrated by the decentralized learning process, in the learned parameters, without requiring a specific learning process nor manual setting (using a generic configuration for hyperparameters). It would still be possible, though, to use knowledge of the differences to optimize the system (custom hyperparameters), by grouping peers with more similar tasks or choosing specific portions of the neural network for averaging.

In the following, we provide a formal description of our method.

II.3 Detailed approach

In local learning, each peer learns a specific model; in non-multi-task distributed learning, all peers learn a single model that is shared with every other peer. Our distributed multi-task learning system introduces the notion of *partial models*. Each peer learns a set of partial models; each of which can be learned by one, some or all peers. Figure 1 gives an example of peers with partial models. We can we classify models into three categories. A *global model* consists of a partial model shared by all the peers. A *semi-local model* consists of a partial model shared by several but not all peers, while a *local model* consists of a partial model associated with only one peer. A peer has at least one model of any type, but none of the types is required.

To implement our partial models, we define *slices* as disjoint sets of neurons that cover a peer's entire neural network: $N^p = \biguplus_i S^{p,i}$, where $S^{p,i}$ is a slice ($\biguplus$: disjoint union). Each slice consists of a portion of a peer's neural network that is averaged on a specific subset of peers. Figure 2 gives an example of neural network divided into parts (dashed rectangles) and slices (nodes (neurons) of same

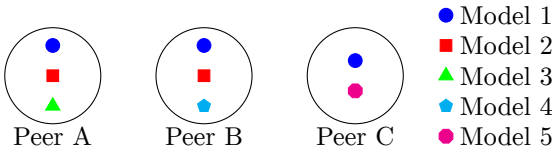


Figure 1: 3 peers with 5 models

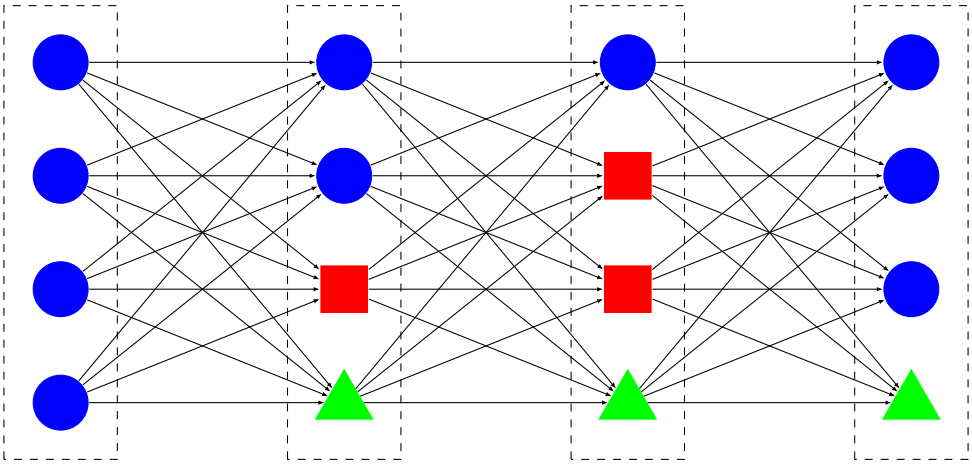


Figure 2: Parts (dashed rectangles) and slices (nodes (neurons) of same color and shape)

color and shape). A peer implements a partial model by associating one of its slices with it. If the implemented partial model is global or semi local, this involves averaging the slice with the peers that also implement the same partial model.

We also define a notion of dependencies between partial models: for example a partial model that recognizes Québécois French may depend on a model that recognizes generic French. If a peer implements a partial model (associating it with one of its slices) it must also implement all the models on which it depends. Dependency defines a partial order (the relationship is antisymmetric and transitive) on the set of models; if A depends on B, we say that A is lower than B. This allows us to determine how to average inter-model weights. An example of models of different types with dependency relationships is given in Figure 3 (boxes ares partial models, arrows are dependency relationships).

II.3.a Partial averaging

Partial models allow us to implement partial averaging, as opposed to global averaging of the whole neural network, as done in non-multi-task collaborative learning. For each partial model, we only average the slices associated with it at the different peers that implement it.

When averaging the slices (one for each peer) associated with a partial model, we average all the activation-function parameter vectors, $\mathbf{q}$, of associated neurons, and a subset of the input weights (a subset of each of $\mathbf{w}$'s elements). Specifically, we average all the intra-model weights, i.e. those connecting two neurons in the same slice. For inter-model input weights, we rely on the notion of dependency between partial models. Specifically, we average only inter-model weights that connect a model to the models it depends on in either direction. If we have model A that depends on model B, we average inter-model weights of neurons of A from neurons of B and those of neurons of B from neurons of A. In other words, the input weights of neurons are not necessarily averaged with the model associated with the neurons they are part of, but rather with the model that is lower in the dependency hierarchy between the model of the neuron they are part of and the model of the neuron from which the input comes; if no such hierarchical relationship exists (the set of all models being only partially ordered by the dependency relationship), the weights are kept local.

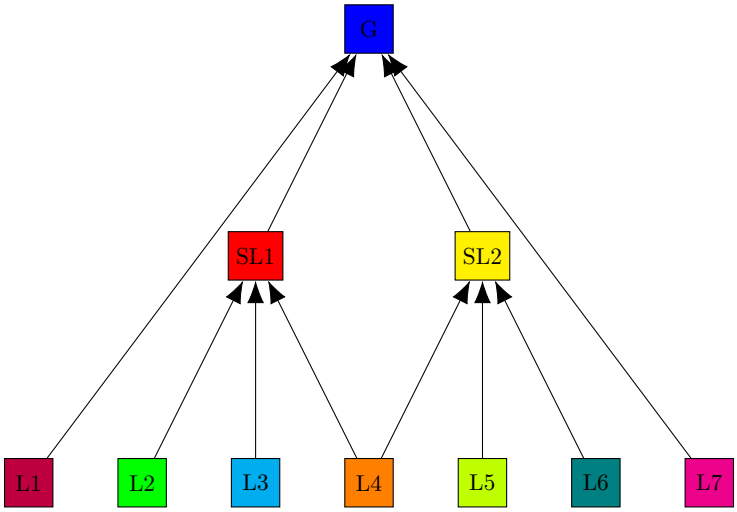
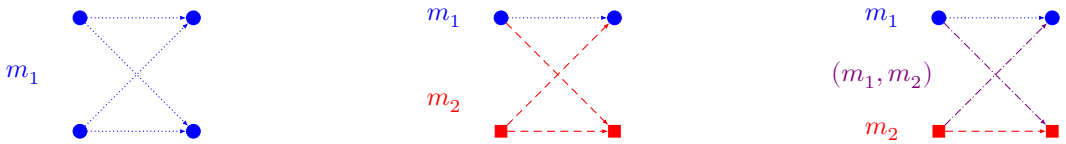


Figure 3: Models with dependency relationships (A →B ⇔ A depends on B)



(a) Only one model (m_1) (b) Red (m_2) depends on Blue (m_1) (c) No dependency

Figure 4: Averaging inter-model weights with different dependencies. A neuron's color and shape indicates the model associated with the neuron; an edge color and style indicates for which model this edge weight will be averaged. We represent two models: blue circles and dotted lines versus red squares and dashed lines. Purple edges refers to weights that are not averaged in either model.

The notion of dependency ensures that inter-model weights have the same meaning for all the peers implementing a model. For example, in Figure 4b, the red model can be Québécois, while the blue model may be generic French. We average inter-model weights among the peers that implement the Québécois model. If there is no dependency between two models (Figure 4c) inter-model weights remain local to each peer. It would in principle be possible to average such weights among all peers that implement both partial models at the same time, but this would add significant complexity to the process for limited gain. An example of 3 peers with 5 partial models and dependency relationships is given in Figure 5.

Math lovers looking for a more mathematical formalization may want to read Appendix A.

II.3.b Algorithm

We detail our multi-task learning process in Algorithm 1. Algorithm 2 presents a modified main loop (and an additional function) to allow cross-model averaging. We use the following notation: $p \Vdash m$ means that peer p implements model m; $n \Subset m$ (resp. $i \Subset m$) means that the neuron n (resp. indexed by i) is part of the (implementation of) model m; $m.deps$ denotes the dependencies of model m. The notation $n \Subset m.deps$ (resp. $i \Subset m.deps$) is a short for $\bigvee_{m' \in m.deps} n \Subset m'$ (resp. $\bigvee_{m' \in m.deps} i \Subset m'$). For simplification, this formalization does not address index differences between peers for equivalent neurons and use I for the set of all neuron indices. This issue can be addressed in an implementation by adding index-offset values for each model and each peer.

Those algorithms use two primitives, *average* and *averageRestricted*. The first averages all the elements of a set of vectors, the second averages predefined coordinates (second parameter) of the vectors from a set (first parameter). In a decentralized setup, these functions can be implemented

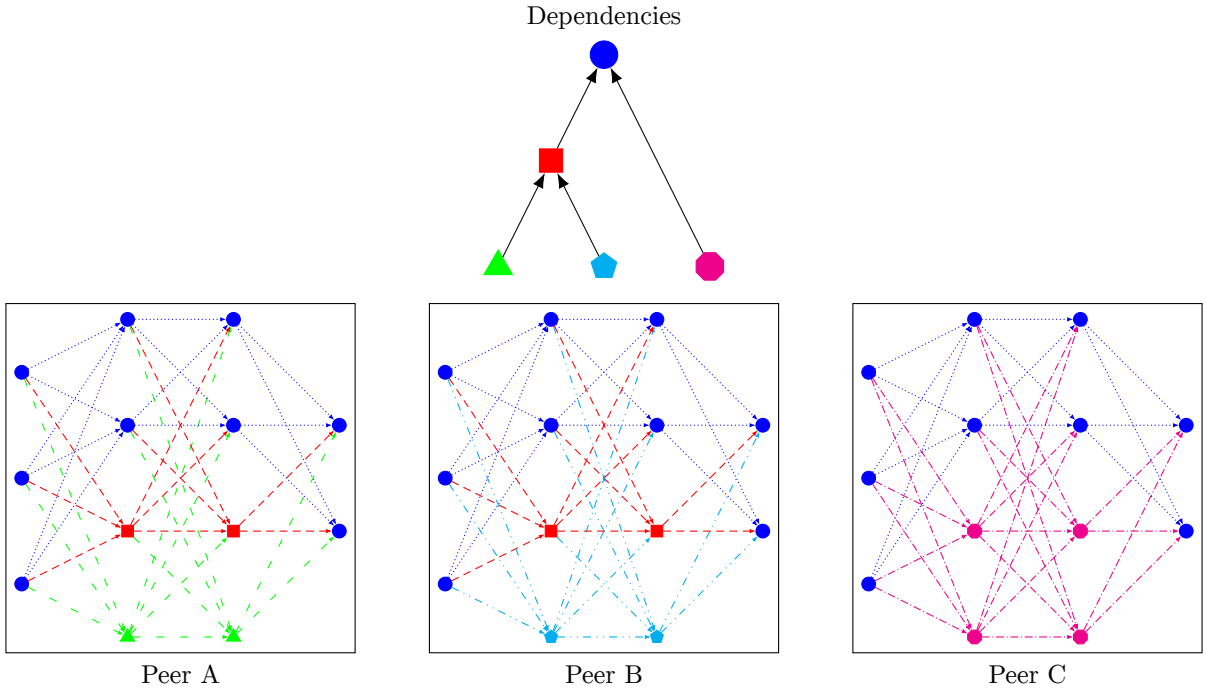


Figure 5: 3 peers' neural networks with 5 models (Blue global; Red semi-local; Green, Cyan and Purple local) and the corresponding dependency graph (A →B ⇔ A depends on B). We that links (weights) connecting two neurons from different models are considered par of the depending model (if any) for averaging purposes.

with a gossip-based averaging protocol [JMB05], or with a private one [BFT19] for improved privacy protection. It is also possible to implement these primitives on a central server, to which peers send their models for averaging, in a federated, rather than decentralized, setup.

Essentially, all peers will independently train their neural network, using some training algorithm with their local training data. Regularly, all peers will synchronize and, for each model, relevant values will be averaged for peers implementing this model. Relevant values being local parameters of neurons implementing the model, as well their weights associated with neurons implementing the same model or one of its dependencies. In the cross-model averaging variant, an additional averaging is done for all pairs of models. Here, we average weights corresponding to links between neurons from the different models of the pair.

II.4 Discussion on centralization

While our method has been designed to be applicable in a decentralized setup, it can also be used in a federated setup. Is this case, on just have to implement the averaging primitives on a central server, to which peers send their models for averaging, rather than using gossip averaging.

It is possible to have several servers as long as a model is associated with only one server. For cross-model averaging, it is required that each pair of models is associated to a unique server.

Since most computing work is done by peers and several servers can be used, this architecture belongs to Edge Computing [Shi+16].

II.5 Privacy

While this method does not include specific privacy protection mechanism, the way it works naturally protects users privacy.

First, only models are transmitted, not the actual data, thus, the only information transmitted is

Algorithm 1: Decentralized multi-task learning

Data: num number of peers, mbn number of training rounds between averaging, M, the set of all models, T, N and D as defined before

loop
  each peer $0 \leqslant p < num$ **does**
    for $0 \leqslant i < mbn$ **do**
      N^p.TRAIN(D^p)
  foreach $m \in M$ **do**
    AVERAGEMODEL($\{p \in [\![0, num[\![| p \Vdash m\}$,$m$) `// Does nothing if model` m `is local`
function AVERAGEMODEL(P,m) **is**
  foreach $i \in T$ **do**
    foreach $k \in T_i | k \Subset m$ **do**
      AVERAGE($\{N^p_{i,k}.\mathbf{q} | p \in P\}$)
      AVERAGERESTRICTED($\{N^p_{i,k}.\mathbf{w} | p \in P\}$,$\{j | j \Subset m \vee j \Subset m.deps\}$)
    foreach $k \in T_i | k \Subset m.deps$ **do**
      AVERAGERESTRICTED($\{N^p_{i,k}.\mathbf{w} | p \in P\}$,$\{j | j \Subset m\}$)

about the whole data-set and not individual data elements.

Also, only portions of each node's model are averaged. When the network converges to a stable state, all averaged slices will converge to the same value for all peers. All peer specific information will end up in the non-averaged portion of the network, which is never transmitted, since this portion is the only thing that differentiate peers. On the first rounds, the information transmitted will be more peer specific, but since the models will not have converged to a stable state yet, this information will be very noisy.

Since semi-locals models can be averaged on specific servers, this also reduces the amount of information that has to be transmitted to the central server (averaging the global model). Semi-local models may, for example, be private and limited to a specific organization's members, among all the organizations involved in the process. On a distributed setup, semi-local models are only transmitted to a limited set of peers.

Also, peers will never get to see any part of another peer's model with a centralized implementation.

When implementing the method using gossip averaging, a private gossip averaging protocol, like the one proposed in [BFT19], can be used.

III Theoretical analysis

We now present a theoretical analysis of our distributed multi-task learning process. Due to the high level of generality of our system (not limited to one specific kind of neural network or learning rule), we provide an analysis of our system's behavior compared to local learning, which can be used to adapt existing convergence proofs to our model. More precisely, we show that in the case of loss-based learning (e.g. SGD), our decentralized system is essentially equivalent to a single model that comprises the whole set of partial models on a single system.

Each peer p has a parameter vector depending on time t (discrete, $\in \mathbb{N}$) $x_p(t) \in \mathbb{R}^{n_p}$ and a loss function $l_p \in \mathbb{R}^{n_p} \to \mathbb{R}$.

We first need to define a global loss function for our problem.

Since all peers' parameters are associated with a partial model (which can be local, for the parameters that are not averaged; note that coordinates not averaged must be considered part of the local

Algorithm 2: Decentralized multi-task learning, cross-model averaging extension

Data: num number of peers, mbn number of training rounds between averaging, M, the set of all models, T, N and D as defined before

loop
  each peer $0 \leqslant p < num$ **does**
    for $0 \leqslant i < mbn$ **do**
      N^p.TRAIN(D^p)
  foreach $m \in M$ **do**
    AVERAGEMODEL($\{p \in [\![0, num[\![| p \Vdash m\}$,$m$) `// Does nothing if model` m `is local`
  foreach $(m1, m2) \in M \times M | m1 \neq m2$ **do** `// Cross-model part`
    AVERAGECROSSMODELS($\{p \in [\![0, num[\![| p \Vdash m1 \wedge p \Vdash m2\}$,$m1$,$m2$) `/* Does nothing if no or only one peer implements both models */`
function AVERAGECROSSMODELS(P,$m1$,$m2$) **is**
  foreach $i \in T$ **do**
    foreach $k \in T_i | k \Subset m1$ **do**
      AVERAGERESTRICTED($\{N^p_{i,k}.\mathbf{w} | p \in P\}$,$\{j | j \Subset m_2\}$)
    foreach $k \in T_i | k \Subset m2$ **do**
      AVERAGERESTRICTED($\{N^p_{i,k}.\mathbf{w} | p \in P\}$,$\{j | j \Subset m_1\}$)

model of a peer, also if the cross-model averaging extension is used, each pair of models have to be considered has one additional model for this analysis), we define a global parameter vector. To this end, we just take all partial models used, $m_0(t) \in \mathbb{R}^{r_0}, m_1(t) \in \mathbb{R}^{r_1}, ..., m_q(t) \in \mathbb{R}^{r_q}$ and concatenate them: $M(t) = m_0(t)m_1(t)...m_q(t) \in \mathbb{R}^u$ ($u = \sum_i r_i$). If a peer p implements models 1, 3 and 5, then, after averaging, we have $x_p(t) = m_1(t)m_3(t)m_5(t)$.

To define our global loss function, $L \in \mathbb{R}^u \to \mathbb{R}$, we first define a set of functions $h_p \in \mathbb{R}^u \to \mathbb{R}$. $h_p(v)$ is simply equal to $l_p(v')$ where v' is the vector v restricted to the coordinates that corresponds to models implemented by peer p. For example, if $v = w_0w_1w_2w_3w_4$ and the peer p implements models 1, 2 and 4, then $h_p(v) = l_p(w_1w_2w_4)$. Since the values of parameters associated with models not implemented by peer p have no direct influence on the output of p's neural network, it is logical that they have no influence on p's loss.

Now we can simply define our global loss function as the sum of all h's: $L(v) = \sum_p h_p(v)$.

Let $v_{\llcorner}k_{\lrcorner}$ correspond to the coordinates of vector v associated with model k. We assume that each peer uses a learning rule satisfying the following property $\mathbb{E}[x_p(t+1) - x_p(t)] = -\lambda_p(t) \circ \nabla l_p(x_p(t))$ ($z \circ z'$ is the element-wise product of vectors z and z'), which holds true for common variants of SGD ($\lambda_p(t)$ is the vector of the learning rates of peer p at time t for all parameters). Since we use model averaging, we need to distinguish peers' models before and after averaging; we use $x_p(t)$ for the value before averaging and $x'_p(t)$ for the value after averaging ($x'_p(t)_{\llcorner}k_{\lrcorner} = m_k(t)$) and the version of the previous formula that we apply is $\mathbb{E}[x_p(t+1) - x'_p(t)] = -\lambda_p(t) \circ \nabla l_p(x'_p(t))$.

Now, for each model m_i we have:

$$m_k(t) = \frac{\sum_{p|p \Vdash m_k} x_p(t)_{\llcorner}k_{\lrcorner}}{\#\{p \mid p \Vdash m_k\}}$$

$$m_k(t+1) - m_k(t) = \frac{\sum_{p|p \Vdash m_k} x_p(t+1)_{\llcorner}k_{\lrcorner}}{\#\{p \mid p \Vdash m_k\}} - m_k(t)$$

$$m_k(t+1) - m_k(t) = \frac{\sum_{p|p \Vdash m_k} x_p(t+1)_{\llcorner}k_{\lrcorner}}{\#\{p \mid p \Vdash m_k\}} - \frac{\sum_{p|p \Vdash m_k} x'_p(t)_{\llcorner}k_{\lrcorner}}{\#\{p \mid p \Vdash m_k\}}$$

$$m_k(t+1) - m_k(t) = \frac{\sum_{p|p\Vdash m_k} x_p(t+1)_{\llcorner}k_{\lrcorner} - \sum_{p|p\Vdash m_k} x'_p(t)_{\llcorner}k_{\lrcorner}}{\#\{p \mid p \Vdash m_k\}}$$

$$m_k(t+1) - m_k(t) = \frac{\sum_{p|p\Vdash m_k} x_p(t+1)_{\llcorner}k_{\lrcorner} - x'_p(t)_{\llcorner}k_{\lrcorner}}{\#\{p \mid p \Vdash m_k\}}$$

$$\mathbb{E}[m_k(t+1) - m_k(t)] = \frac{\sum_{p|p\Vdash m_k} \mathbb{E}[x_p(t+1)_{\llcorner}k_{\lrcorner} - x'_p(t)_{\llcorner}k_{\lrcorner}]}{\#\{p \mid p \Vdash m_k\}}$$

$$\mathbb{E}[m_k(t+1) - m_k(t)] = -\frac{\sum_{p|p\Vdash m_k} \lambda_p(t)_{\llcorner}k_{\lrcorner} \circ \nabla l_p(x'_p(t))_{\llcorner}k_{\lrcorner}}{\#\{p \mid p \Vdash m_k\}}$$

We now add the assertion that all peers implementing a model use the same learning rate for the parameters part of this model $\forall_{k,p,p'|p\Vdash m_k \wedge p'\Vdash m_k} \lambda_p(t)_{\llcorner}k_{\lrcorner} = \lambda_{p'}(t)_{\llcorner}k_{\lrcorner} = \Lambda_k(t)$. If the learning rate is supposed to be variable and not depending only on t, it is possible to compute a single learning rate for all peers implementing a model or to compute an optimal learning rate locally for each peer, average all value and use this averaged value. Now we can write $\lambda_k(t) = \frac{\Lambda_k(t)}{\#\{p|p\Vdash m_k\}}$ and $\Lambda(t) = \lambda_0(t)\lambda_1(t)...\lambda_q(t)$.

$$\mathbb{E}[m_k(t+1) - m_k(t)] = -\frac{\sum_{p|p\Vdash m_k} \Lambda_k(t) \circ \nabla l_p(x'_p(t))_{\llcorner}k_{\lrcorner}}{\#\{p \mid p \Vdash m_k\}}$$

$$\mathbb{E}[m_k(t+1) - m_k(t)] = -\frac{\Lambda_k(t)}{\#\{p \mid p \Vdash m_k\}} \circ \sum_{p|p\Vdash m_k} \nabla l_p(x'_p(t))_{\llcorner}k_{\lrcorner}$$

$$\mathbb{E}[m_k(t+1) - m_k(t)] = -\frac{\Lambda_k(t)}{\#\{p \mid p \Vdash m_k\}} \circ L(M(t))_{\llcorner}k_{\lrcorner}$$

$$\mathbb{E}[m_k(t+1) - m_k(t)] = -\lambda_k(t) \circ \nabla L(M(t))_{\llcorner}k_{\lrcorner}$$

Which, when we consider all models at the same time, gives us:

$$\mathbb{E}[M(t+1) - M(t)] = -\Lambda(t) \circ \nabla L(M(t))$$

We are back to the same property for the global learning rule as for the local ones except that the loss function is the sum of the local losses and the learning rate of each parameter is divided by the number of peers having this parameter as part of their model.

This requires some discussion. Is having the learning rates divided by the number of peers implementing the corresponding model bad? Remember that the global loss function is a sum. If you consider the simple case where all peers have the same loss function, this mean that, compared to local learning, the (global) loss function will have its values, and thus its gradient, multiplied by the number of peers. This means that, in this case, if the learning rate was not divided by the number of peers, federated learning would be equivalent to learning with a higher learning rate, while it is reasonable to think that it should be equivalent to local learning with no modification of the learning rate, which is what we achieve with our reduced global learning rate. More shortly, the reduced learning rate compensates the increased gradient.

We can summarize this analysis the following way: for a loss-based learning algorithm (typically SGD), our decentralized system is equivalent to training the whole set of partial models at once (on a single system) with a loss function that is the sum of peers local loss functions and a learning rate that is, for each partial model, divided by the number of peers implementing the model (which compensates the increased gradient induced by the sum).

IV Application to specific neural networks

Our method is general and, to be useful, needs to be implemented with some kind of neural network.

For our experiments, we focus on the Multi-Layer Perceptron, because it is a well-known, simple and efficient kind of neural network that is easy to train (with gradient descent). We also conduct additional experiments with a second kind of neural network, a custom neural network relying on associative learning, to prove that our method is not limited to MLP and SGD.

IV.1 Multi-Layer Perceptron

We worked with a basic MLP [GBC16], without convolutional layers [LB95]. The simple design of this kind of this well-known neural network being well suited for testing. For the same reason, we trained it using classical Stochastic Gradient Descent with Back Propagation [RHW86], without any additions (variable learning-rate, momentum). While such additions may give better performance, we wanted to stick to the most classical case possible to keep our results clean from potential side effects of more complex designs.

For this kind of neural networks the parts are, naturally, the layers. For each layer, we can associate a fixed number of neurons with each model we want to use.

Note that, in the case of Convolution Neural Networks (CNN) [LB95], unlike MLP, each convolution layer is not a single part, since neurons constituting it have different inputs. Those layers could easily be kept in a single slice but partial averaging is also possible, if the layers uses siblings cells: a few constitutional cells sharing the same inputs. In that case, a set of sibling cells is a part.

Figure 6 gives and example of an MLP with 4 models. Here, the red model depends on green and yellow, which themselves depends on blue. Grey lines are the ones that can not be associated with a unique model (they can be associated with the (unordered) pair $(green, yellow)$).

MLP will be our main test case for experiments. It will allow us to evaluate how our method works depending on various parameters.

IV.2 Associative neural network

We designed a custom kind of neural network using an associative learning rule. It is a proof of concept to show how our method performs on a network that differs significantly from MLP and uses a learning algorithm different from gradient descent.

We took elements from Hopfield Networks [Hop82] for the neurons themselves and Restricted Boltzmann Machines [Smo86] for the structure of the network and the data is represented using Sparse Distributed Representation [AH15]. Our network is divided into a visible and a hidden layer, forming a complete (directed) bipartite graph. The activation functions are fixed thresholds (output is either 1 or 0) and the network is used for completion tasks: a partial vector (with 0 instead of 1 for some of its coordinates) is given as input and the network must return the complete vector as output.

The process is the following: the visible layer's neurons outputs are set to the values of the input vector, the hidden layers' neuron's output is computed from the values of the visible layer, finally, the visible neurons' values are updated, based on the hidden layer's values. The computation is simple: if the weighted sum of the inputs is $\geqslant$ some threshold (hyperparameter), the output is 1, 0 otherwise. See Algorithm 3 for pseudo-code.

For learning, the network is given a complete vector as input, then the hidden layer's values are computed, finally, both layers' neurons' weights are updated. The learning rule is the following: if two neurons are active together, the weight of the link between them is increased, if one is active and not the other, this weight is decreased. Increment values depend on a fixed learning rate (half of the distance from the extreme value times the learning rate) and weights are in the interval $[-1; 1]$. See Algorithm 4 for pseudo-code.

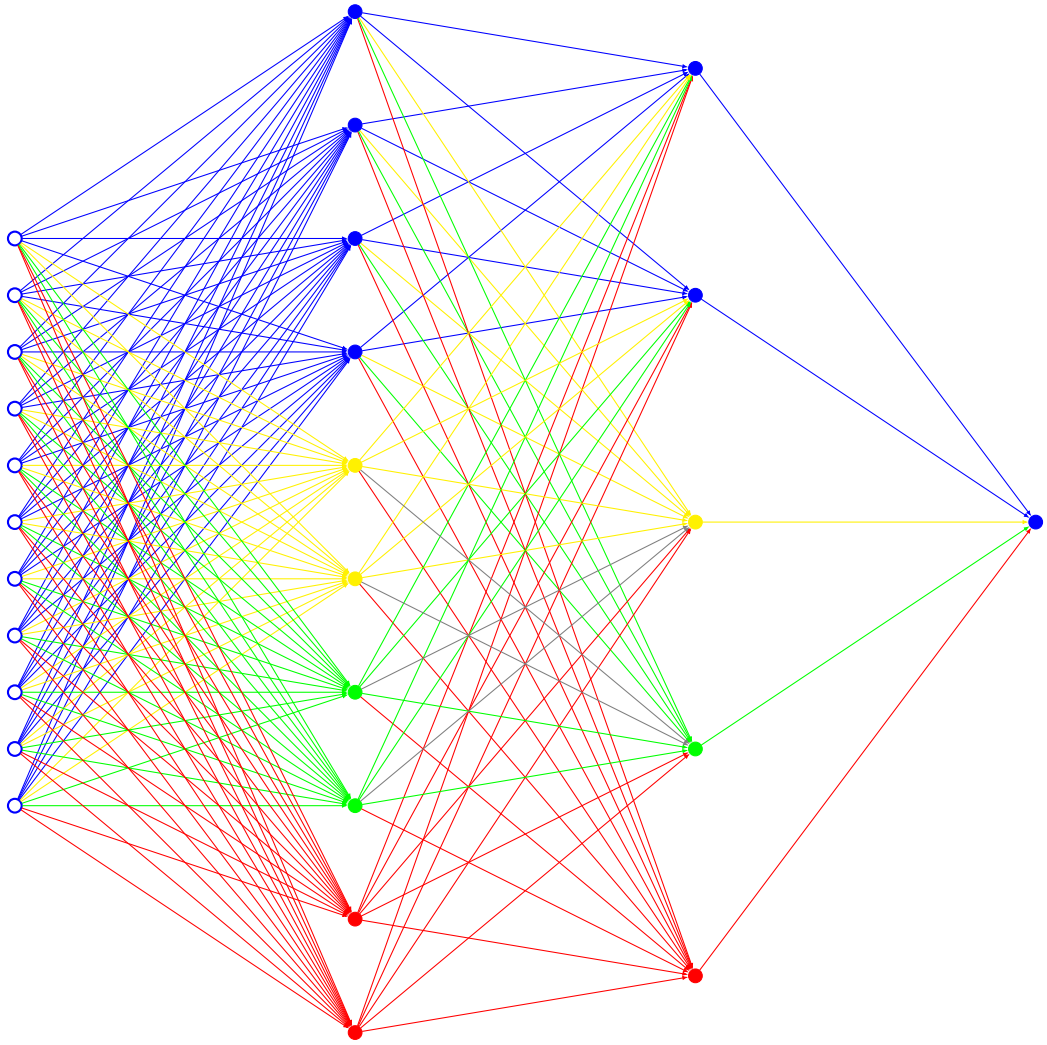


Figure 6: A multi-layer perceptron with 4 partial models

V Experiments

Our experiments aim to show the efficiency of our method on simple examples and evaluate how this efficiency is affected by various parameters. While real world applications of our method are likely to be more complex, these simple examples are an efficient way to provide detailed results about our method's behavior.[3]

Our results from different tasks are summarized in Figure 7. We see that our approach leads systematically compared to local training and global training (non-multi-task federated averaging). More details in the followings subsections.

V.1 Multi-Layer Perceptron - General considerations

This section summarizes the general considerations applicable to the two test series we performed on the Multi-Layer Perceptron.

In both cases, we used a classical Stochastic-Gradient-Descent algorithm with Back Propagation [RHW86] with a fixed learning rate of 0.1 and a sigmoid activation function $f(x) = \frac{1}{1+e^{-x}}$.

We use the following notation to describe our MLP layouts: $n{=}m{=}p{=}q$ means that we have a four-layer MLP (input, output, and two hidden layers) with n neurons in the first (input) layer, m in

[3]Our code is available at https://gitlab.inria.fr/abouchra/distributed_neural_networks

Algorithm 3: Computation

Data: *thld* threshold for neurons' activation

for $0 \leqslant i < hidden.size$ **do**
  if $visible.output \cdot hidden.weight[i] \geqslant thld$ **then**
    $hidden.output[i] \leftarrow 1$
  else
    $hidden.output[i] \leftarrow 0$
for $0 \leqslant i < visible.size$ **do**
  if $hidden.output \cdot visible.weight[i] \geqslant thld$ **then**
    $visible.output[i] \leftarrow 1$
  else
    $visible.output[i] \leftarrow 0$

the second, and so on. We use a similar notation to describe partial models: n-m-p-q means that the described model has n neurons for the first layer, m for the second, and so on.

We use a 784=300=100=k configuration, were k is the number of classes for the relevant task, for our MLP (two hidden layers), unless explicitly stated otherwise, a fixed learning rate of 0.1 and a sigmoid activation function $f(x) = \frac{1}{1+e^{-x}}$.

The "Accuracy" metrics, since the tasks we used with MLP are symbol recognition tasks, is simply the proportion of recognized symbols. A symbol is considered recognized if the most active neuron of the last layer corresponds to this symbol.

The "averaging level" is the number of neurons in the global model (shared by all peers), others neurons being in local models, in the second hidden layers, which contains a total of 100 neurons. The levels we tested are 0, 15, 30, 60, 80 and 100. Some neurons from the first hidden layer are also averaged, the exact numbers are, respectively, 0, 50, 100, 200, 250, 300. An averaging level of 0 is equivalent to local learning, 100 is equivalent to training a single global model (with averaging), similar to federated averaging.

V.2 Multi-Layer Perceptron - FEMNIST

The FEMNIST dataset [Cal+19] targets the recognition of 62 classes of handwritten characters (uppercase and lowercase letters, plus digits). FEMNIST is derived from the NIST Special Database 19 [Nat16], as is the well-known MNIST [LeC+98] dataset. Compared to MNIST, FEMNIST adds non-digit characters and groups characters by writer. This grouping, originally intended for non-multi-task federated learning, can also be used for multi-task federated/decentralized learning since different writers do not have the same handwriting, creating the similar but different tasks we need to evaluate our system.

To adapt FEMNIST to the multi-task setup, we preprocess the dataset as follows. First, after partitioning the data by writer, we sort the writers by descending numbers of samples. This allows us to select the writers with the largest samples in each experiment. We also had to perform a random shuffle on the samples for each writer, since the natural order of the samples has obvious bias (big clusters of digits notably).

Our experiments simulate a peer-to-peer/federated setup. We assign exactly one writer to each peer and randomly shuffle its samples. Then we limit all peers' (writers') sets of samples to the same size, truncated the largest sets. This prevents a "super peer" bias, where one peer with a very large dataset could, alone, allow the learning process to gain greater general accuracy than all other peers. This bias would artificially increase the average accuracy, masking the interest of collaboration between multiple smaller peers.

Algorithm 4: Learning

Data: *thld* threshold for neurons' activation, l the learning rate

```
for 0 ⩽ i < hidden.size do
    for 0 ⩽ j < visible.size do
        if visible.output[j] == 1 then
            if hidden.output[i] == 1 then
                hidden.weight[i][j] += (1 − hidden.weight[i][j]) / 2 · l
            else
                hidden.weight[i][j] −= (1 + hidden.weight[i][j]) / 2 · l
for 0 ⩽ i < visible.size do
    for 0 ⩽ j < hidden.size do
        if hidden.output[j] == 1 then
            if visible.output[i] == 1 then
                visible.weight[i][j] += (1 − visible.weight[i][j]) / 2 · l
            else
                visible.weight[i][j] −= (1 + visible.weight[i][j]) / 2 · l
```

While this anti-bias rule is a great constraint that poses limitations on both, the number of peers we can include in our tests and the number of samples we can use, effectively reducing the maximum possible accuracy, we believe that it is very important to try to eliminate, has much as possible, such bias in the dataset to really prove the efficiency of distributed multi-task learning itself, without any side effects. While some could argue that dataset size disparity is present in real-world datasets, we can answer that, if the only interest of federated learning is transferring information from "rich" peers to "poor" peers, then, nothing would force those rich peers to share with poor, preferring to keep their large dataset for themselves or sell them or sell the models learned from them, making federated/distributed learning effectively useless. Thus, we will perform our analysis on peers with same size dataset, to prove the efficiency our system, even when it can not benefit from this wealth redistribution effect.

Each peers data is divided in a training and a testing section.

Each test is done on three variants of the FEMNIST task. FEMNIST-A, the classical FEMNIST task, with all characters (62 classes). FEMNIST-M, were some similar-looking characters are merged in a single class as proposed in the NIST Special Database 19 documentation [Nat16] (47 classes). FEMNIST-D, limited to digits (10 classes).

We performed a total of 3 independent tests with MLP on FEMNIST, each designed to evaluate one specific aspect of our system: partial averaging efficiency, different averaging schemes, effect of the number of peers.

Results are median of 10 runs. The error bars corresponds to the second and third quintiles.

V.2.a Averaging level

This first experiment aims to show that performing a partial averaging is better, in the considered multi-task situation, than a complete averaging or no averaging and to determine which level of averaging is the best.

For this experiment we used 16 peers, each with a training set of 360 characters for A and M variants, 100 for the D variant. The test set was 60 characters long for A and M variants and 30 characters long for the D variant. The mini-batch size was 180 for A and M variants and 50 for the D variant.

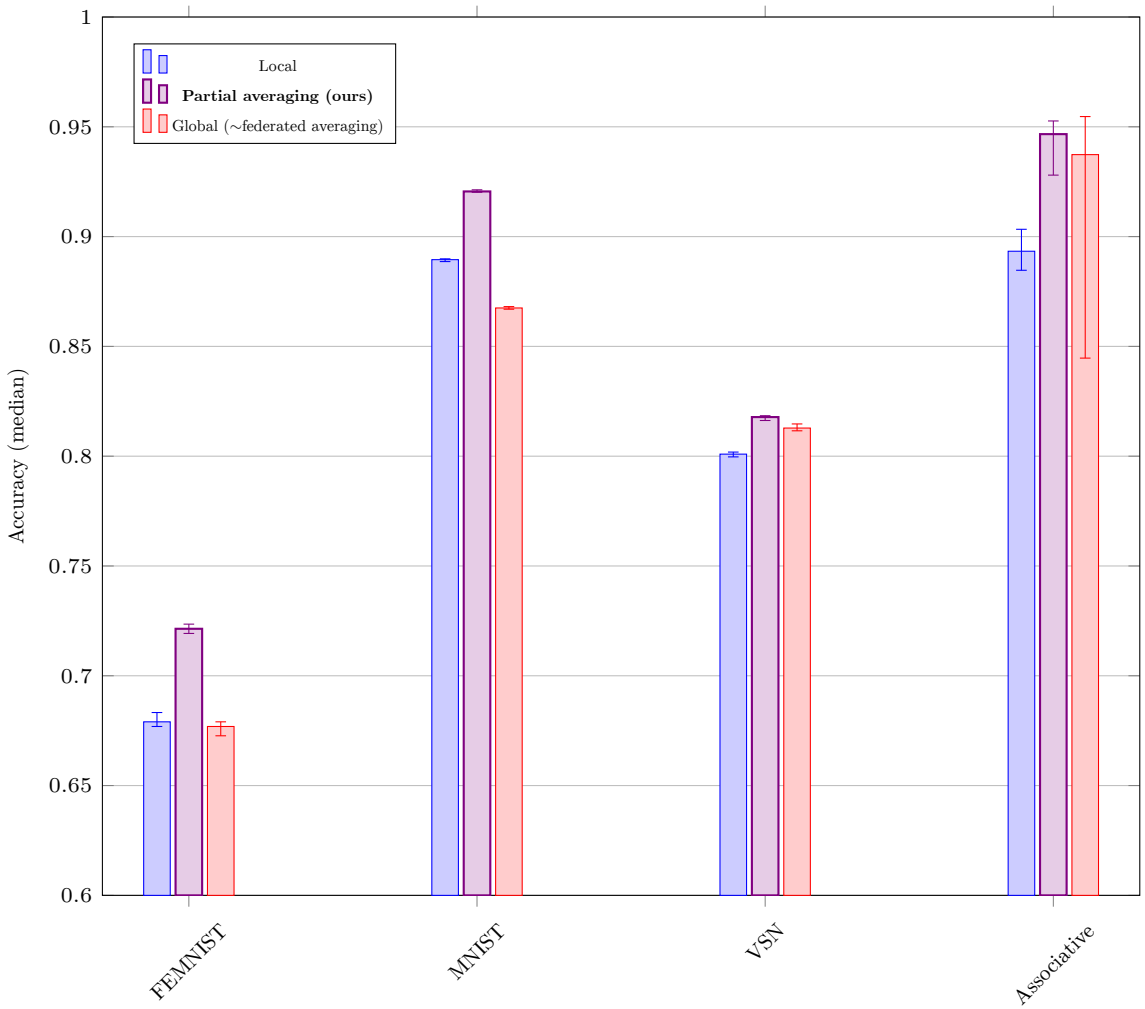


Figure 7: Results summary
Partial averaging is 80% for all but Associative, 70%

The "averaging level" is the number of neurons in the global model (shared by all peers), others neurons being in local models, in the second hidden layers, which contains a total of 100 neurons. The levels we tested are 0, 15, 30, 60, 80 and 100. Some neurons from the first hidden layer are also averaged, the exact numbers are, respectively, 0, 50, 100, 200, 250, 300. Each line corresponds to a different mini-batch number (from 30 to 1000). The averaging process was done after every mini-batch.

Results are median of 10 runs. The error bars corresponds to the second and third quintiles. The results are presented in Figures 8, 9 and 10 (in order, A, M and D variants).

From those results, we can conclude that, whatever the mini-batch number is, the best performance is always achieved using partial averaging. In general, the best performance is achieved with an averaging level of 80. This validates our partial averaging concept.

V.2.b Averaging scheme

In this experiment, we compare our approach of partial averaging of each hidden layer (in its best performing variant, 784-250-80-62, with other partial averaging schemes based on complete averaging of specific layers. Notably, this includes a configuration similar to what is commonly used in local multi-task learning approaches design as "transfer learning": sharing everything but the last layer (784-300-100-0).

For this experiment we used 16 peers, each with a training set of 360 characters for A and M

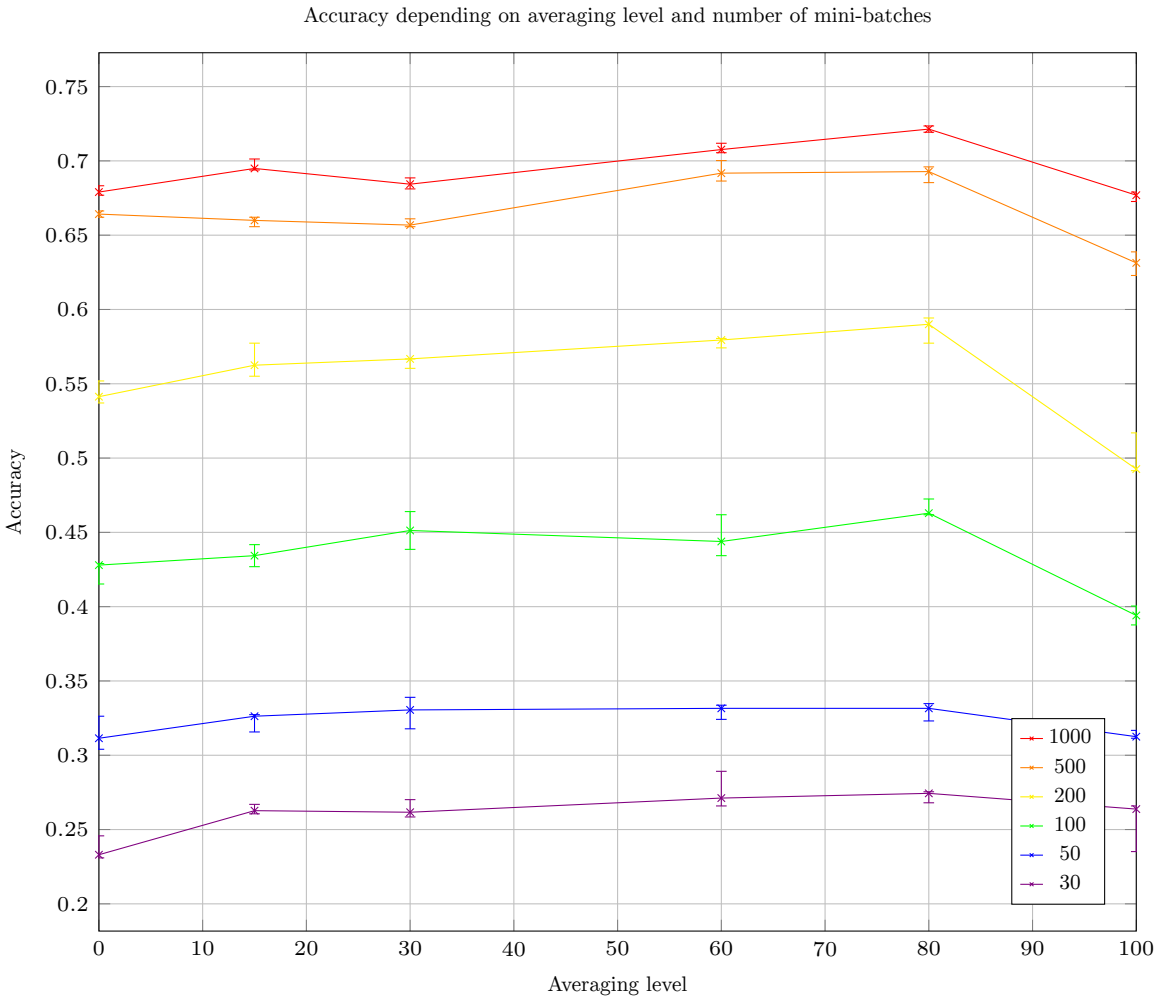


Figure 8: Averaging level on FEMNIST-A - Accuracy for various averaging levels and numbers of mini-batches

variants, 100 for the D variant. The test set was 60 characters long for A and M variants and 30 characters long for the D variant. The mini-batch size was 180 for A and M variants and 50 for the D variant.

Each line corresponds to a different averaging scheme. 200 mini-batches were used for training. The averaging process was done after every mini-batch.

Results are median of 10 runs. The error bars corresponds to the second and third quintiles. The results are presented in Figures 11, 12 and 13 (in order, A, M and D variants).

This experiment shows that, in the FEMNIST case, whatever the mini-batch number is, partially averaging hidden layers is superior to all tested schemes based on averaging specific layers, including 784-300-100-0. This proves the interest of having a flexible system allow partial averaging of layers rather than specific layers averaging.

V.2.c Number of peers

In this experiment, we want to evaluate how our system's performance is affected by the number of peers.

Due to the nature of the FEMNIST dataset, with writers that can in no way be considered "uniform" either themselves or in their respective differences, this test will present the following particularities. First, for all "numbers of peers" tested, we actually tested 64 peers and the "number of peers" value is

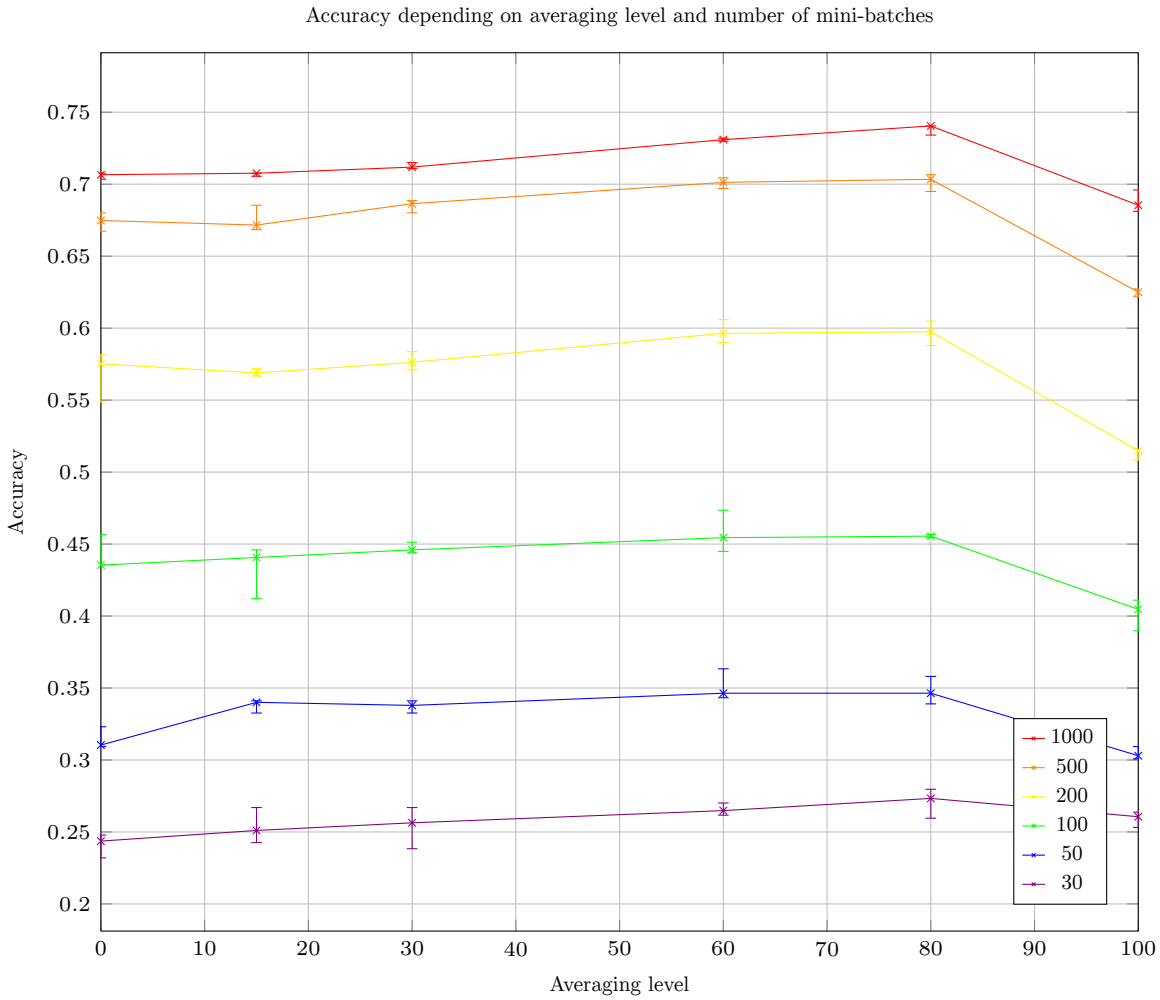


Figure 9: Averaging level on FEMNIST-M - Accuracy for various averaging levels and numbers of mini-batches

the number of peers sharing the same "global model". For example, the value for 16 peers is obtained by taking 4 groups of 16 peers (64 total) and performing the averaging process separately for each group. This eliminates any bias that could be exhibited if, for example, the first peers in our dataset were actually "easier". Secondly, we need to admit that another bias can still exist in our results. Even with our grouping method, some groups can be harder than others, notably, it is likely that multi-task learning is easier if you only have 4 different tasks rather than 16. Due to the nature of the FEMNIST dataset, each peer has its own, specific, task, thus, more peers mean more tasks. As a consequence, this tests is not only testing the effect of the number of peers but also the number of different task. While the number of peers should increase accuracy, the number of tasks should decrease it. We have to keep this in mind when interpreting our results. Also, in addition to the number of peers, the different sets of peers may have unknown and unpredictable biases, so, we have to be careful with our conclusions.

For this experiment, each peer had a training set of 350 characters for A and M variants, 100 for the D variant. The test set was 60 characters long for A and M variants and 30 characters long for the D variant. The mini-batch size was 175 for A and M variants and 50 for the D variant.

Each line corresponds to a different averaging level. 200 mini-batches were used for training. The averaging process was done after every mini-batch.

Results are median of 10 runs. The error bars corresponds to the second and third quintiles. The

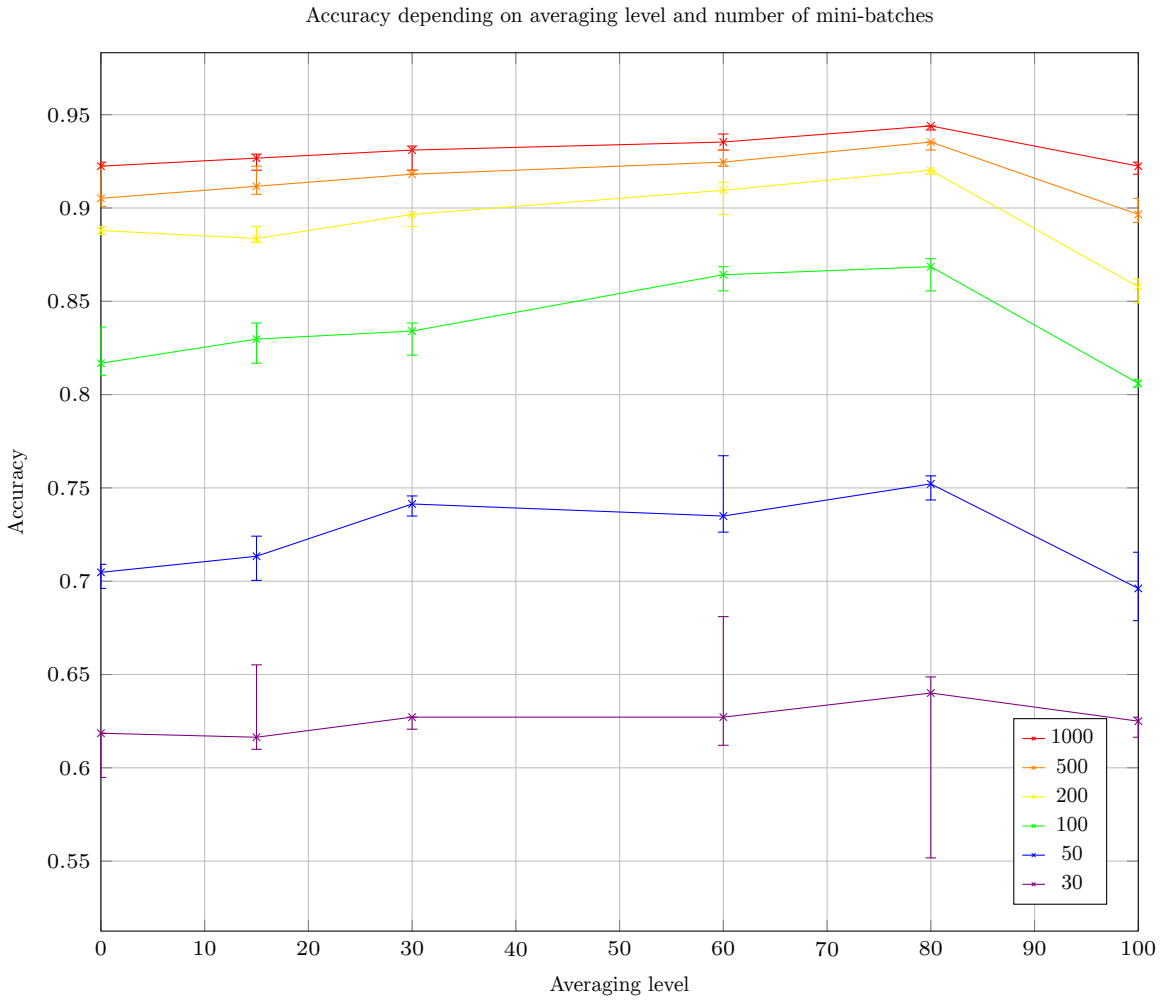


Figure 10: Averaging level on FEMNIST-D - Accuracy for various averaging levels and numbers of mini-batches

results are presented in Figures 14, 15 and 16 (in order, A, M and D variants).

Obviously, the line for a 0 averaging level is flat, since the number of peers has no influence in this case. It clearly appears that full averaging (100) rapidly loses efficiency when the number of peers increases, specifically at the beginning. This is due to increased number of different tasks; an averaging level of 100 being unable to perform multi-task learning. For an averaging level of 80, we observe, however, that our multi-task learning system seems to allow effective gains.

V.3 Multi-Layer Perceptron - modified MNIST

We continue our experiments with the MLP on the well-known MNIST [LeC+98] dataset. The task is simple: recognizing handwritten digits. To generate different but related learning tasks for each peer, we permute digits. Compared to FEMNIST, this more artificial testing allows us to perform a more precise evaluation, by providing better control on the respective tasks of peers.

To obtain different training sets for different peers, we divided the MNIST training set in successive sequences. This is important because practical implementations will be likely to have peers with completely independent data; keeping data local being one of the key features of our method. The test set is the 1000 first digits of the MNIST test set.

All the training sets are different, not overlapping, parts of the whole MNIST training set (60000 digits). These choices are a compromise to have a sufficient number of peers while not reducing the

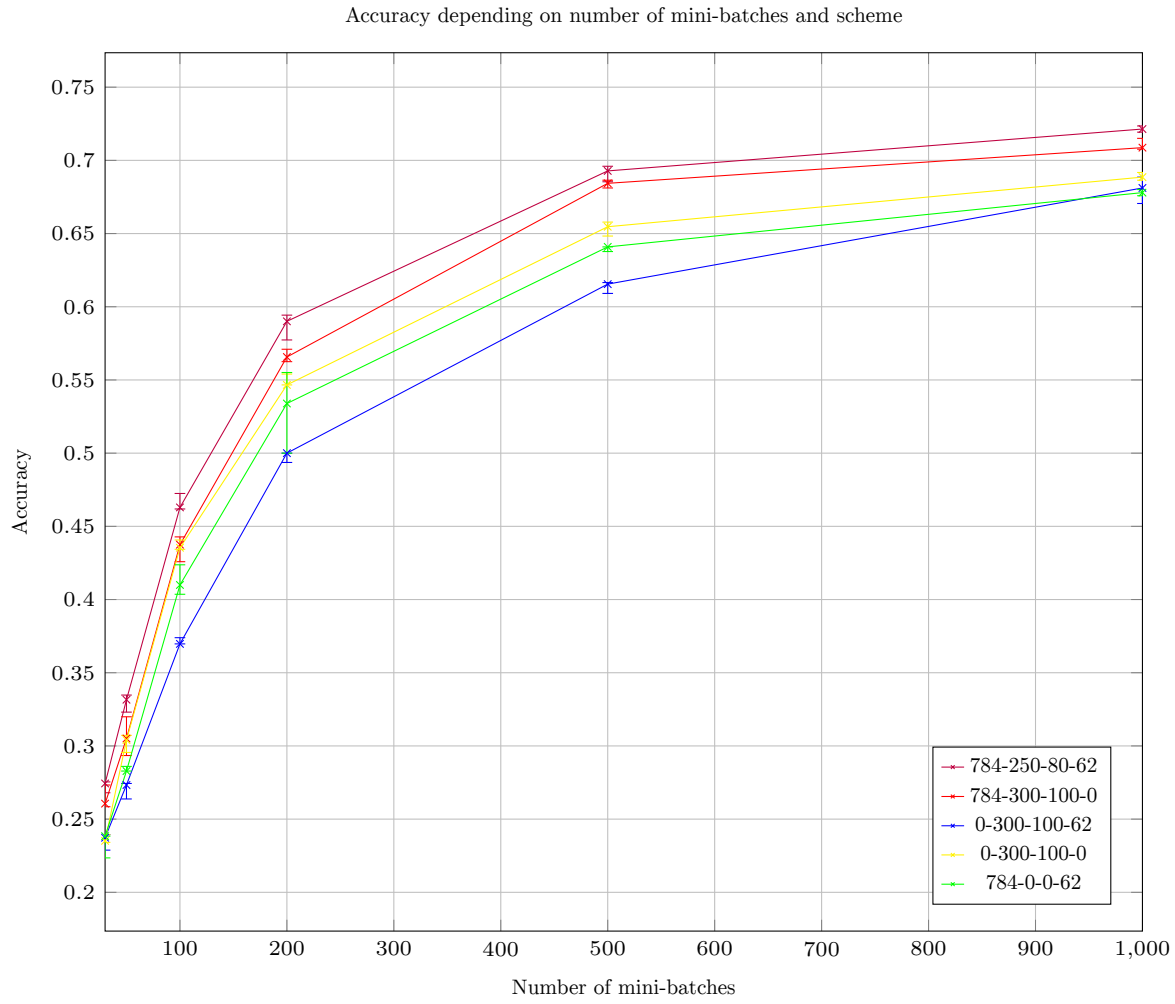


Figure 11: Averaging scheme on FEMNIST-A - Accuracy for various numbers of mini-batches and schemes

training set size too much and maintaining independent training sets, within the limitation imposed by the total size of the MNIST training set.

Differences between peers are induced by permuting digits.

We performed a total of 9 independent tests with MLP on MNIST, each designed to evaluate one specific aspect of our system: partial averaging efficiency, convergence, effect of the number of peers, semi-local models, semi-local+local models, effect of the level of difference between tasks, effect on the number of peers with modified tasks, effect of the training set size, use of a more problem-specific layout.

Results are median of 10 runs, except for semi-local averaging, which has 20 runs (since results were closer, we wanted more samples). The error bars corresponds to the second and third quintiles.

V.3.a Averaging level

This first experiment aims to show that performing a partial averaging is better, in the considered multi-task situation, than a complete averaging or no averaging and to determine which level of averaging is the best.

For this experiment we used 16 peers, each with a training set of 3500 digits. All the training sets are different, not overlapping, parts of the whole MNIST training set (60000 digits). These choices are a compromise to have a sufficient number of peers while not reducing the training set size too much

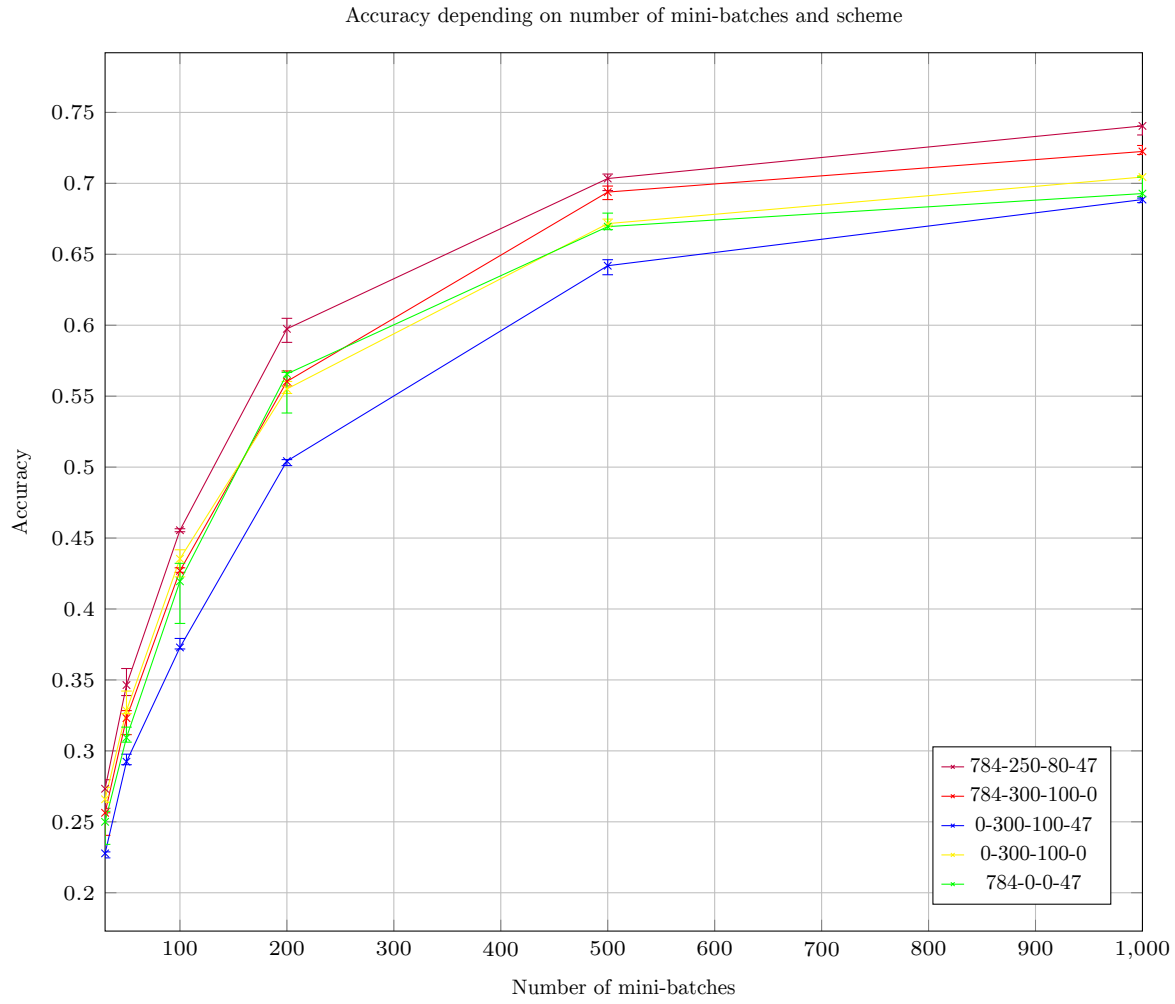


Figure 12: Averaging scheme on FEMNIST-M - Accuracy for various numbers of mini-batches and schemes

and maintaining independent training sets, with MNIST training set size limitation.

The "averaging level" is the number of neurons in the global model (shared by all peers), others neurons being in local models, in the second hidden layers, which contains a total of 100 neurons. The levels we tested are 0, 15, 30, 60, 80 and 100. Some neurons from the first hidden layer are also averaged, the exact numbers are, respectively, 0, 50, 100, 200, 250, 300. Each line corresponds to a different mini-batch size (from 50 to 3000). The averaging process was done after every mini-batch. 30 mini-batches were used for training. Differences between peers are induced by permuting digits 8 and 9 for 7 of the 16 peers, giving us a close to but not even split.

Results are median of 10 runs. The error bars corresponds to the second and third quintiles. The results are presented in Figure 17.

From those results, we can conclude that, whatever the mini-batch size is, the best performance is always achieved using partial averaging. For higher mini-batch size, the best performance is achieved with an averaging level of 80.

The significant drop of performance observed with an averaging level of 100 is due to fact that a unique model can not address the permutation of 8 and 9 for certain peers. This validates our partial averaging concept.

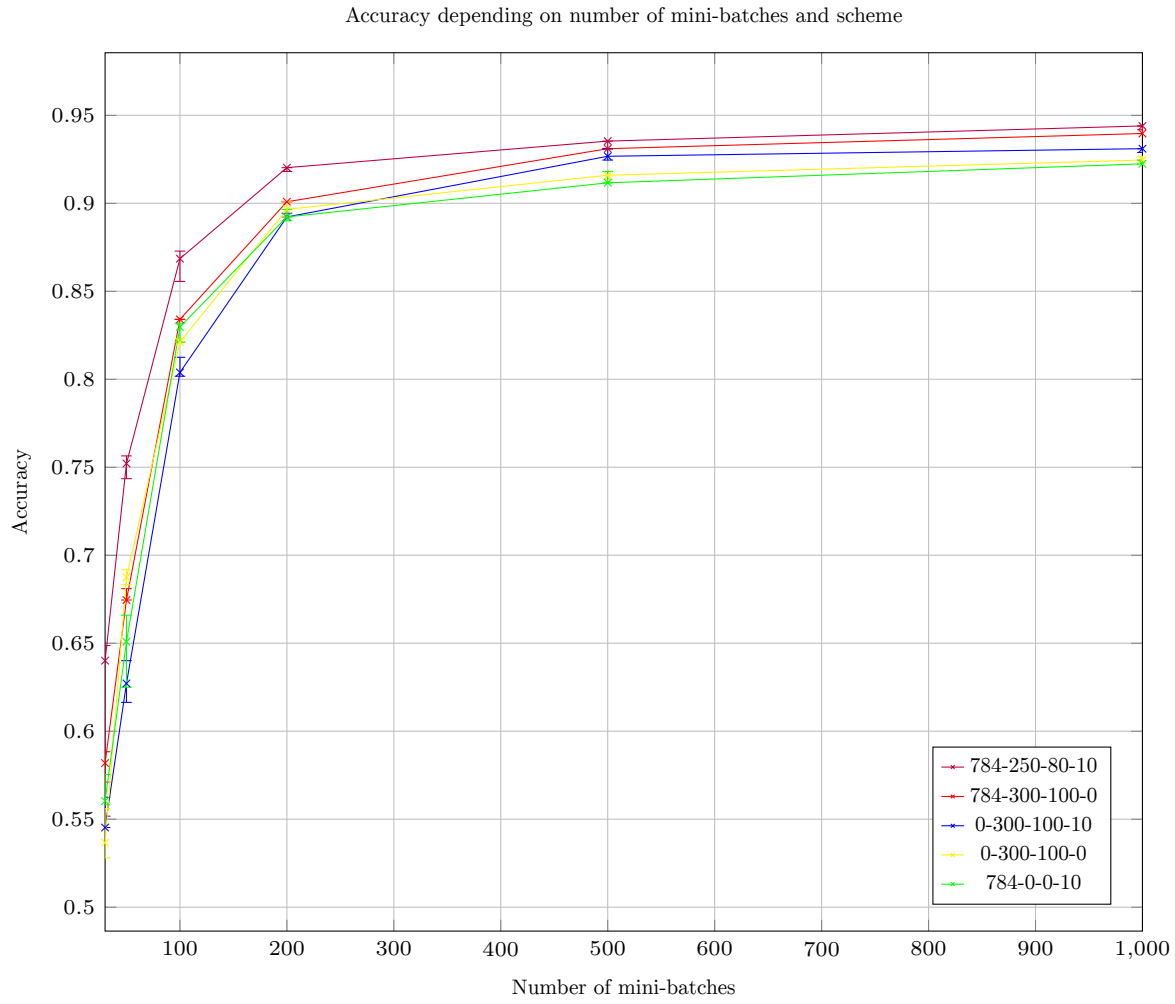


Figure 13: Averaging scheme on FEMNIST-D - Accuracy for various numbers of mini-batches and schemes

V.3.b Convergence

The point of this test is to see how the accuracy evolves during the training process.

For this experiment we tested different averaging levels, with 16 peers, a fixed mini-batch size of 100. Training sets are independent and of size 3500. The smaller mini-batch size allows finer grain in accuracy sampling.

The "averaging level" is the number of neurons in the global model (shared by all peers), others neurons being in local models, in the second hidden layers, which contains a total of 100 neurons. The levels we tested are 0, 80 and 100. Some neurons from the first hidden layer are also averaged, the exact numbers are, respectively, 0, 250, 300. Each line corresponds to a different averaging level. The averaging process was done every 10 mini-batch. 30 mini-batches were used for training. Differences between peers are induced by permuting digits 8 and 9 for 7 of the 16 peers.

Results are median of 10 runs. The results are presented in Figure 18.

At the beginning of the training, no averaging is leading and 80 averaging is last but when approaching maximum accuracy, 80 becomes first and 100 last. The three curves have similar shapes and remain very close.

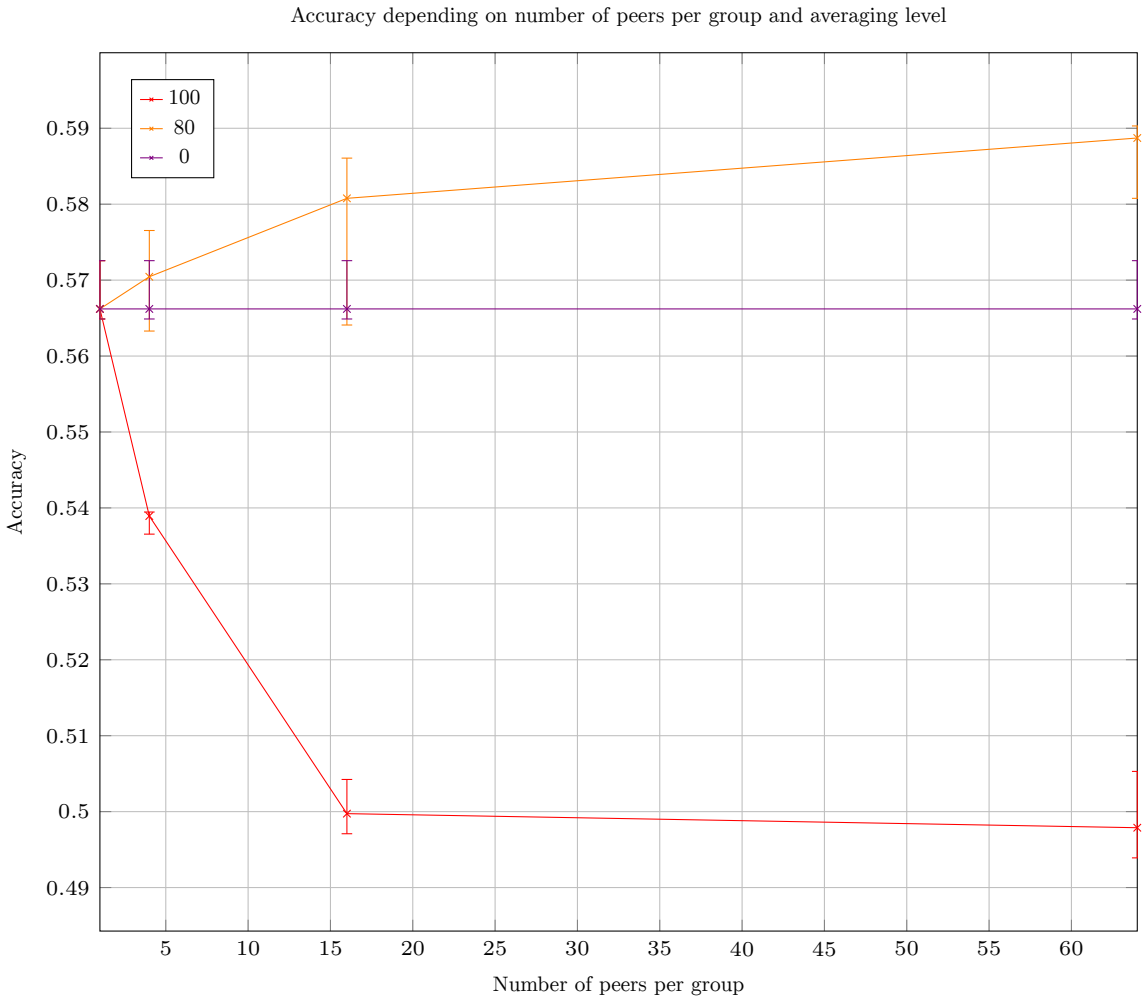


Figure 14: Number of peers on FEMNIST-A - Accuracy for various numbers of peers per group and averaging levels

V.3.c Number of peers

In this experiment, we want to evaluate how different averaging levels' performances are affected by the number of peers.

For this experiment we tested different numbers of peers, from 4 to 16, and different averaging levels with a fixed mini-batch size of 500. Training sets are independent and of size 3500.

The "averaging level" is the number of neurons in the global model (shared by all peers), others neurons being in local models, in the second hidden layers, which contains a total of 100 neurons. The levels we tested are 0, 15, 30, 60, 80 and 100. Some neurons from the first hidden layer are also averaged, the exact numbers are, respectively, 0, 50, 100, 200, 250, 300. Each line corresponds to a different averaging level. The averaging process was done after every mini-batch. 30 mini-batches were used for training. Differences between peers are induced by permuting digits 8 and 9 for one fourth of peers. This ration diving evenly all tested numbers of peers.

Results are median of 10 runs. The error bars corresponds to the second and third quintiles. The results are presented in Figure 19.

We see here that, obviously, the number of peers has no effect when averaging is not used. It also has a small effect with a limited averaging level but, with high level partial averaging (60-80), we get a significant performance gain when increasing the number of peers. The performance of full averaging, despite gaining from peer number increase, remains low. This indicates that partial averaging allows

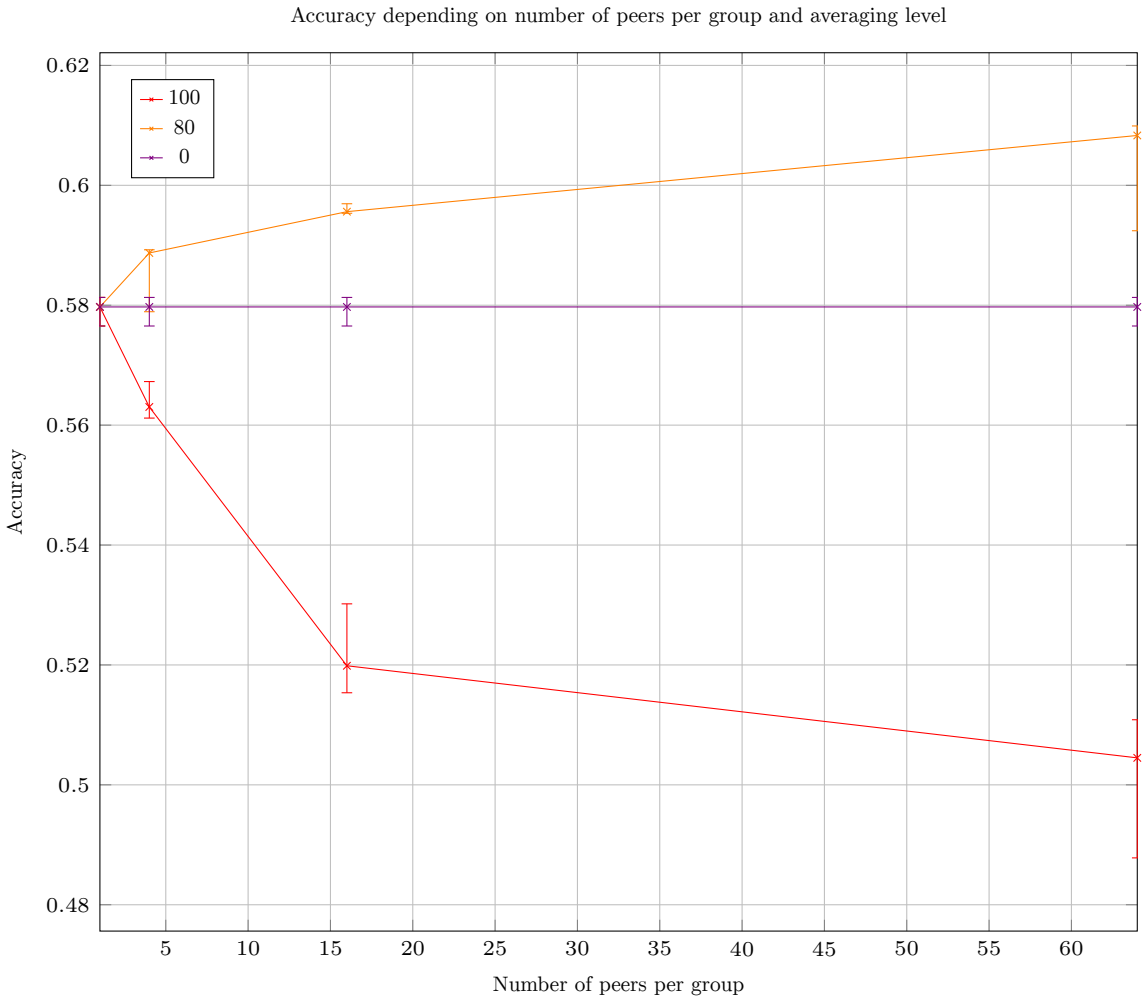


Figure 15: Number of peers on FEMNIST-M - Accuracy for various numbers of peers per group and averaging levels

for performance gain when connecting more peers, giving access to more training examples.

V.3.d Semi-local averaging

Here, we want to see how semi-local models can improve performance, compared to just using global and local models.

For this experiment we tested different averaging schemes with different mini-batch sizes (from 25 to 150). Smaller mini-batches allowing more frequent averaging for this more difficult context. Training sets are independent and of size 200, so that the smaller mini-batches would still cover a significant portion of the training sets. We used 12 peers; a highly divisible number allowing more complex permutation patterns.

Differences between peers are induced by applying all elements of $\mathfrak{S}_3$ to the last 3 digits. The schemes tested are the following: no averaging, complete averaging of all peers' neural networks, complete averaging limited to peers with identical permutation, averaging with an 80 level of all peers' neural networks, averaging with an 80 level (784-250-80-10) of all peers' neural networks + averaging with a 20 level (0-50-20-0) of peers with identical permutation. For the last scheme, we tested two variants: in the first case, no model was declared dependent of another one, which meant that no inter-model weight was averaged, in the second case, the semi-local models were declared dependent on the global model, allowing inter-model weights between global and semi-local to be averaged with

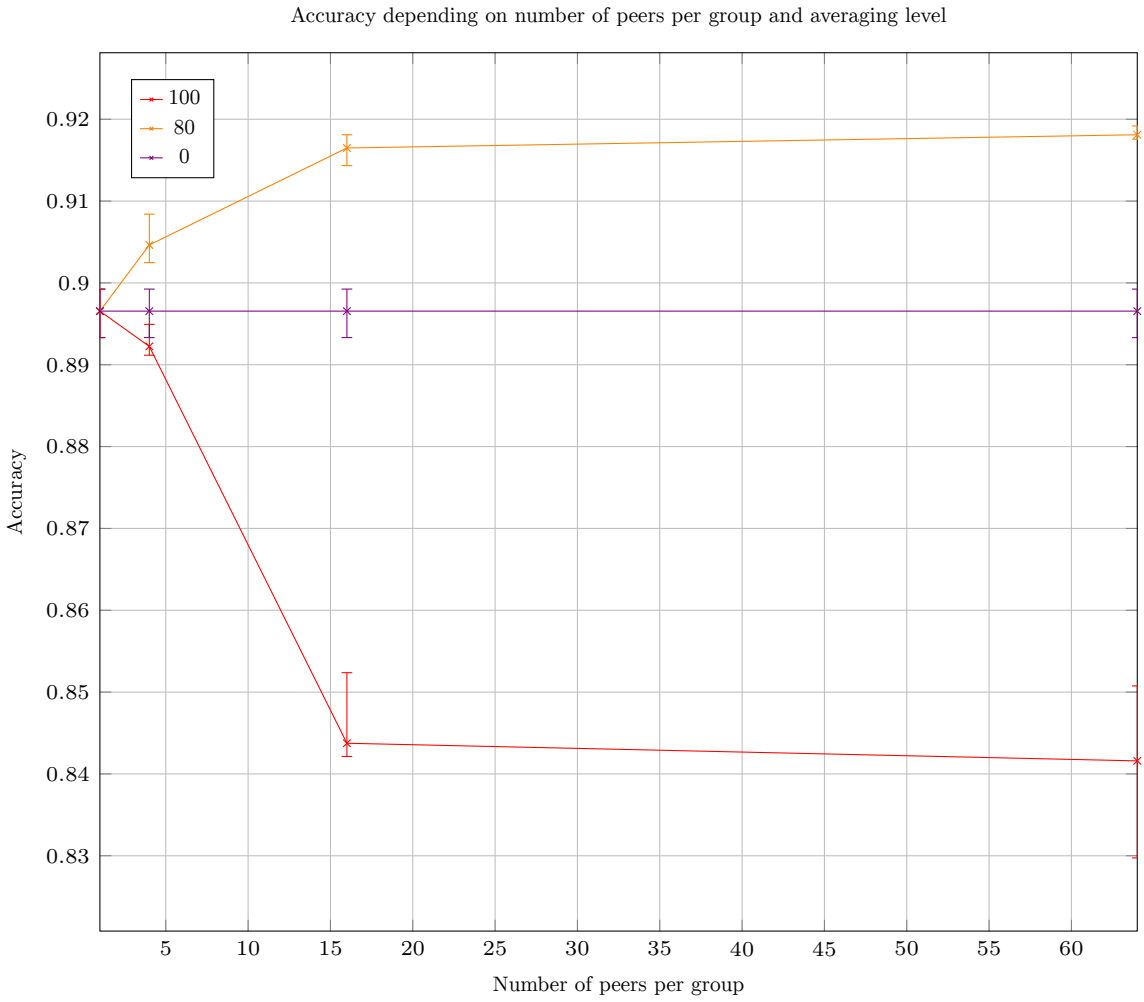


Figure 16: Number of peers on FEMNIST-D - Accuracy for various numbers of peers per group and averaging levels

the semi-local models' internal weights. Each line corresponds to a different averaging scheme. The averaging process was done after every mini-batch. 100 mini-batches were used for training. Each permutation is used for 2 peers (12 peers total).

Results are median of 20 runs (more precise than 10, since results were close). The error bars corresponds to the second and third quintiles. The results are presented in Figure 20.

On this test, we observe that no averaging is the method with the worst performance. Full averaging is better but still less efficient than averaging per class or 80 averaging. Having a 80 global model + a 20 semi-local model ended-up being the best solution here and declaring the dependency allowed greater gains and declaring the dependency allowed greater gains, very significant for higher mini-bash sizes.

V.3.e Semi-local averaging with local models

In this test, we evaluate how a combination of global, semi-local and locals models performs.

For this experiment we tested different averaging schemes with different mini-batch sizes (from 100 to 800), allowing a medium averaging frequency, adapted to the difficulty of the task (complex permutation set, but with few peers). Training sets are independent and of size 1000. We used 4 peers, fewer peers implying that some would have unique permutations, the natural use-case for local models.

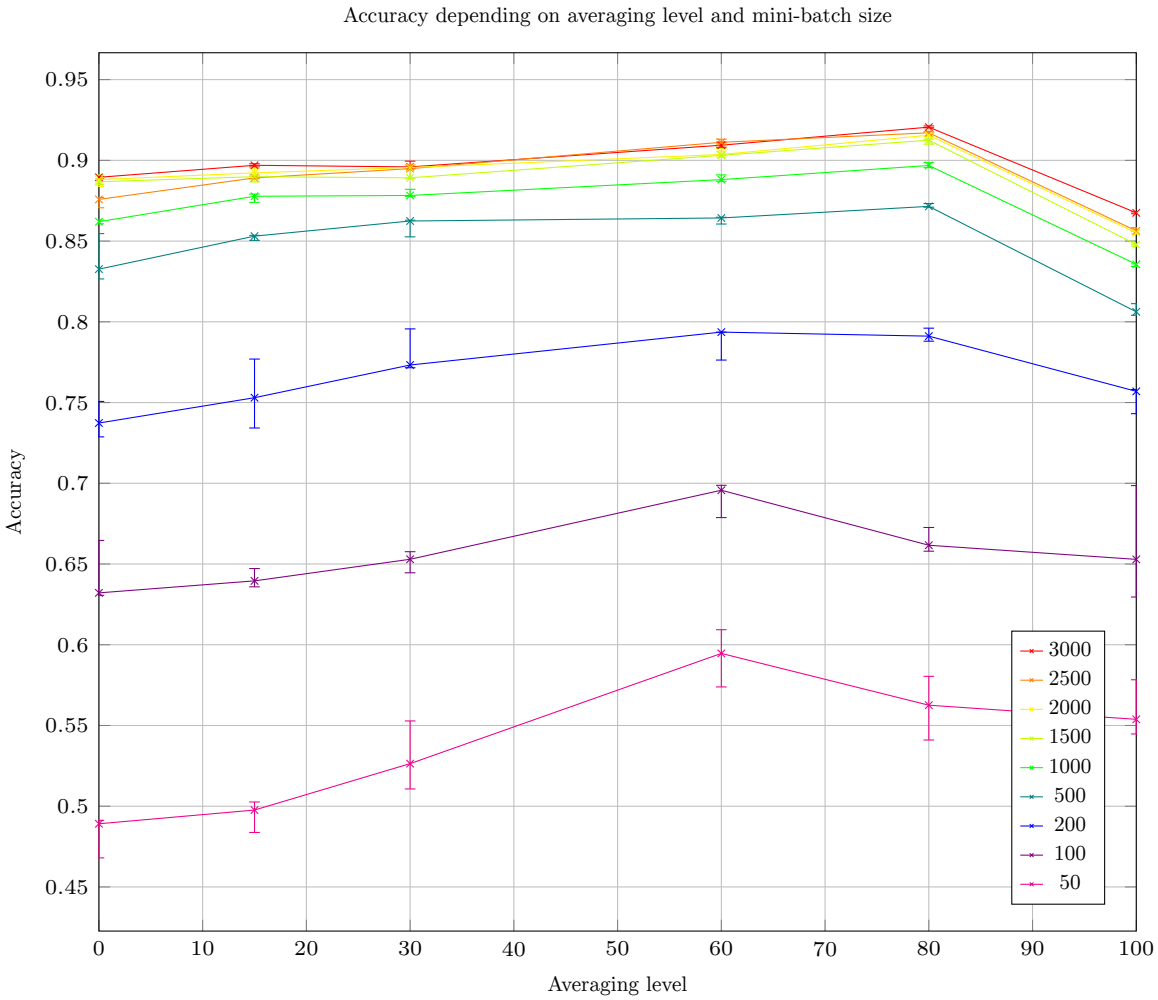


Figure 17: Averaging level on modified MNIST - Accuracy for various averaging levels and mini-batch sizes

Differences between peers are induced by permuting digits 8 and 9 for peer 2 and 3 + digits 6 and 7 for peer 3. The schemes tested are the following: no averaging, complete averaging of all peers' neural networks, complete averaging limited to peers with identical permutation, averaging with an 80 level of all peers' neural networks, averaging with a 70 level (784-220-70-10) of all peers' neural networks + averaging with a 30 level (0-80-30-0) for peers 0 and 1 and an averaging with a 15 level (0-40-15-0) for peers 2 and 3. For the last scheme, we tested two variants: in the first case, no model was declared dependent of another one, which meant that no inter-model weight was averaged, in the second case, the semi-local models were declared dependent on the global model, allowing inter-model weights between global and semi-local to be averaged with the semi-local models' internal weights. Each line corresponds to a different averaging scheme. The averaging process was done after every mini-batch. 50 mini-batches were used for training.

Results are median of 20 runs (more precise than 10, since results were close). The error bars corresponds to the second and third quintiles. The results are presented in Figure 21.

On this test, due to the important differences between peers, full averaging was clearly inferior to all other schemes, including no averaging. 80 averaging was better than no averaging, but still inferior to 100 averaging per class with big enough mini-batches. 70 + 30 for identical peers and 15 for peers with only one common inversion was the best and declaring the dependency allowed greater gains.

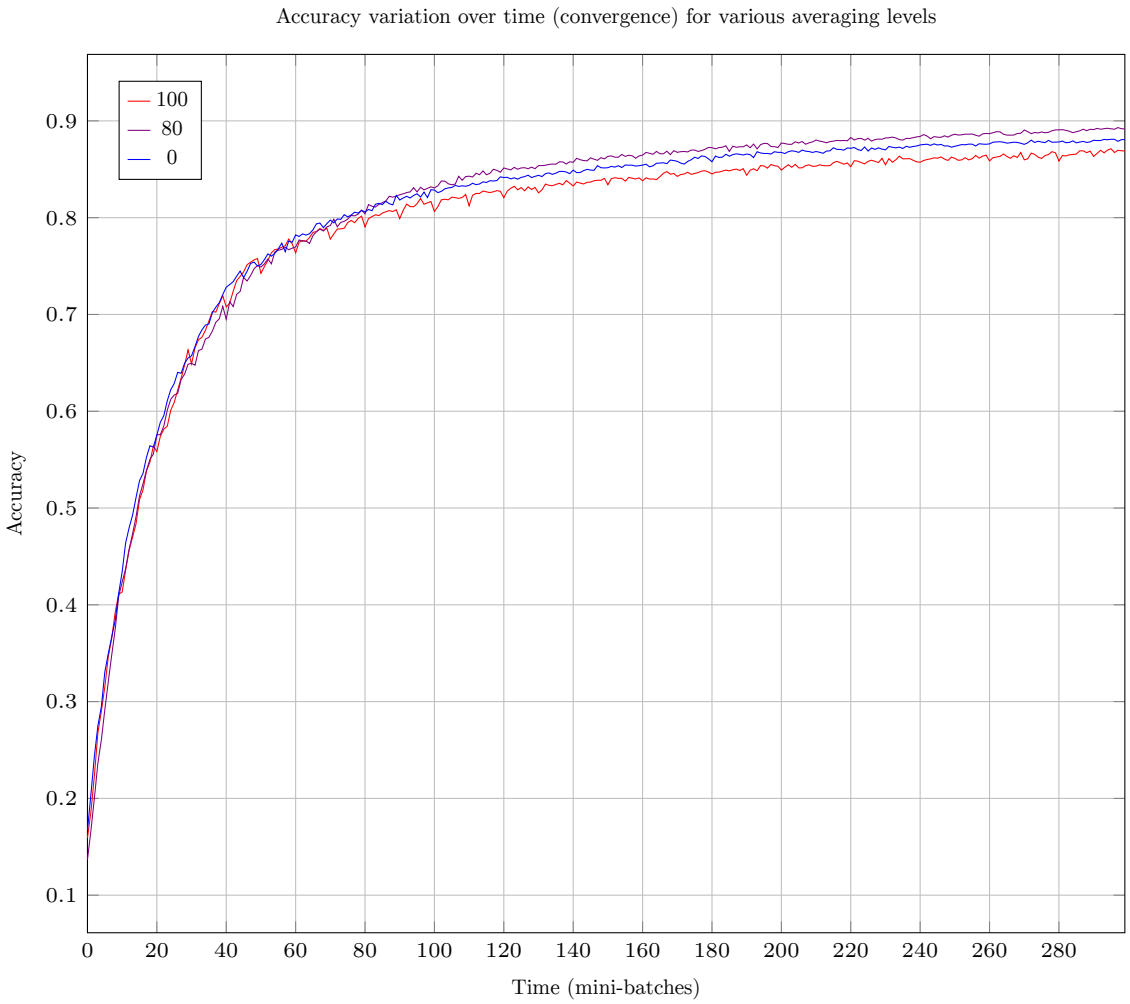


Figure 18: Convergence on modified MNIST - Accuracy over time (convergence) for various averaging levels

V.3.f Difference rate

Here we want to see how the efficiency of different averaging level is affected by the level of difference between peers' tasks.

We tested different averaging levels for different difference rates. The difference rate being the number of digits affected by a different permutation between peers, ranging from 0 to 10. Training sets are independent and of size 3500; mini-batch size is 500. We used 16 peers. Each line corresponds to a different averaging level. The averaging process was done after every mini-batch. 30 mini-batches were used for training. Differences between peers are induced by applying the cycle $(10-r...9)$ where r is the difference rate (or no permutation if $r = 0$, $r = 1$ being impossible with this system) to 7 of the 16 peers output.

Results are median of 10 runs. The error bars corresponds to the second and third quintiles. The results are presented in Figure 22.

We observe that, while it seems to be the best when there is no difference (the gap with 80 and 60 is too low to officially conclude), 100 averaging is the only scheme severely affected by the difference rate, loosing more than 50% of accuracy. For other schemes, 80 and 60 are the best for low difference rate but the gap is significantly reduced when the difference rate increases. For 10, 80 seems to be inferior to other partial averaging scheme, with a level of accuracy similar to no averaging.

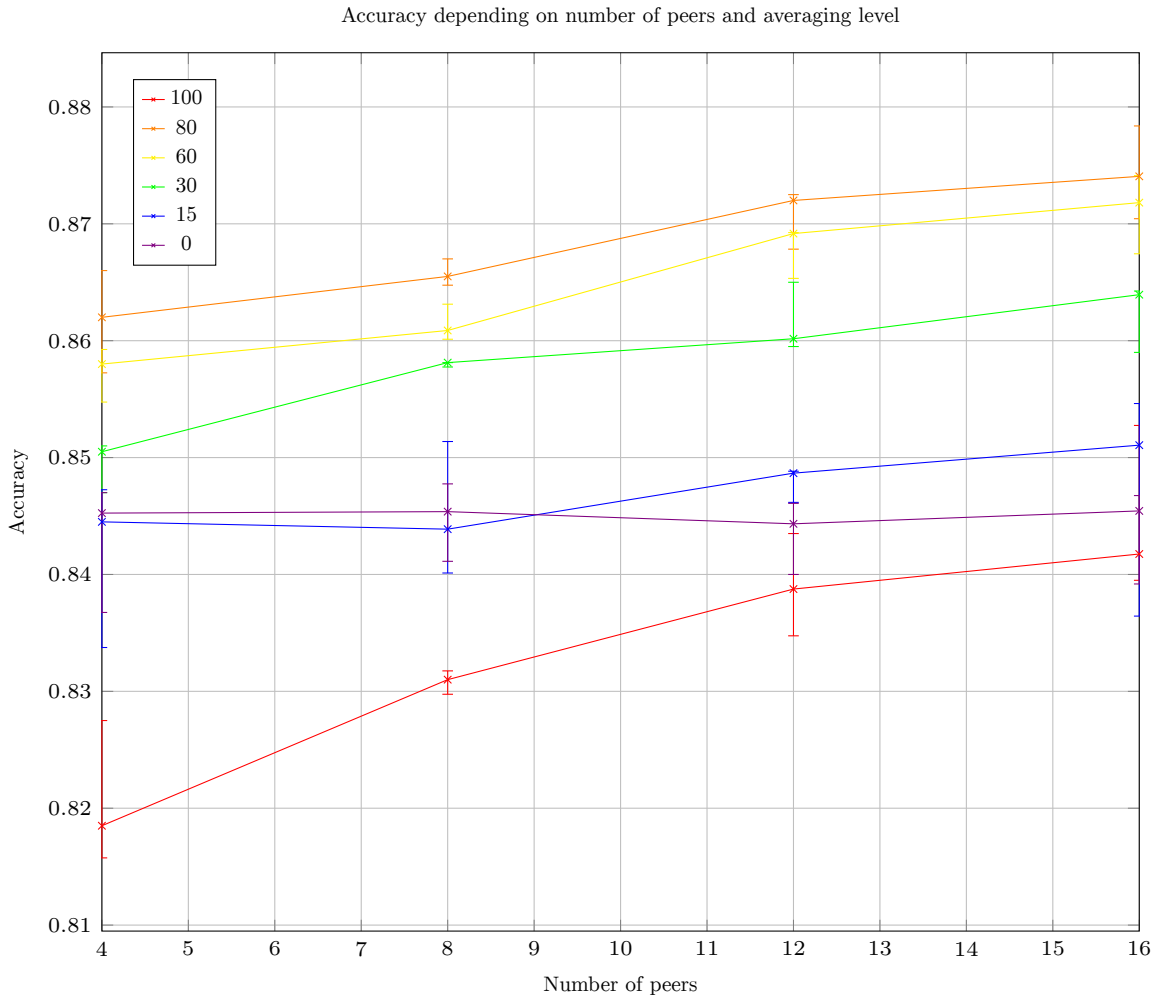


Figure 19: Number of peers on modified MNIST - Accuracy for various numbers of peers and averaging levels

V.3.g Number of peers with permutation

Here we want to see how the efficiency of different averaging level is affected by the proportion of peers with a non-trivial permutation.

We tested different averaging levels for number of peers with permutation. Training sets are independent and of size 3500; mini-batch size is 500. We used 16 peers. Each line corresponds to a different averaging level. The averaging process was done after every mini-batch. 30 mini-batches were used for training. The peers with a non-trivial permutation have 8 and 9 permuted.

Results are median of 20 runs (more precise than 10, since results were close). The error bars corresponds to the second and third quintiles. The results are presented in Figure 23.

We observe that full averaging's accuracy significantly drops when the number of permuted peers increases. Full averaging is the most accurate with 0 peers with permutations (logical) but is only third with 2 and last for 4 to 8. As one could expect, no averaging is not affected by the number of peers with permutation. More interestingly, partial averaging schemes do not seem to be much affected by the number of peers with permutation; only a limited drop is observed for 30, 60 and 80.

V.3.h Training set size

Here we want to see how the efficiency of different averaging level is affected by the size of each peer's training set.

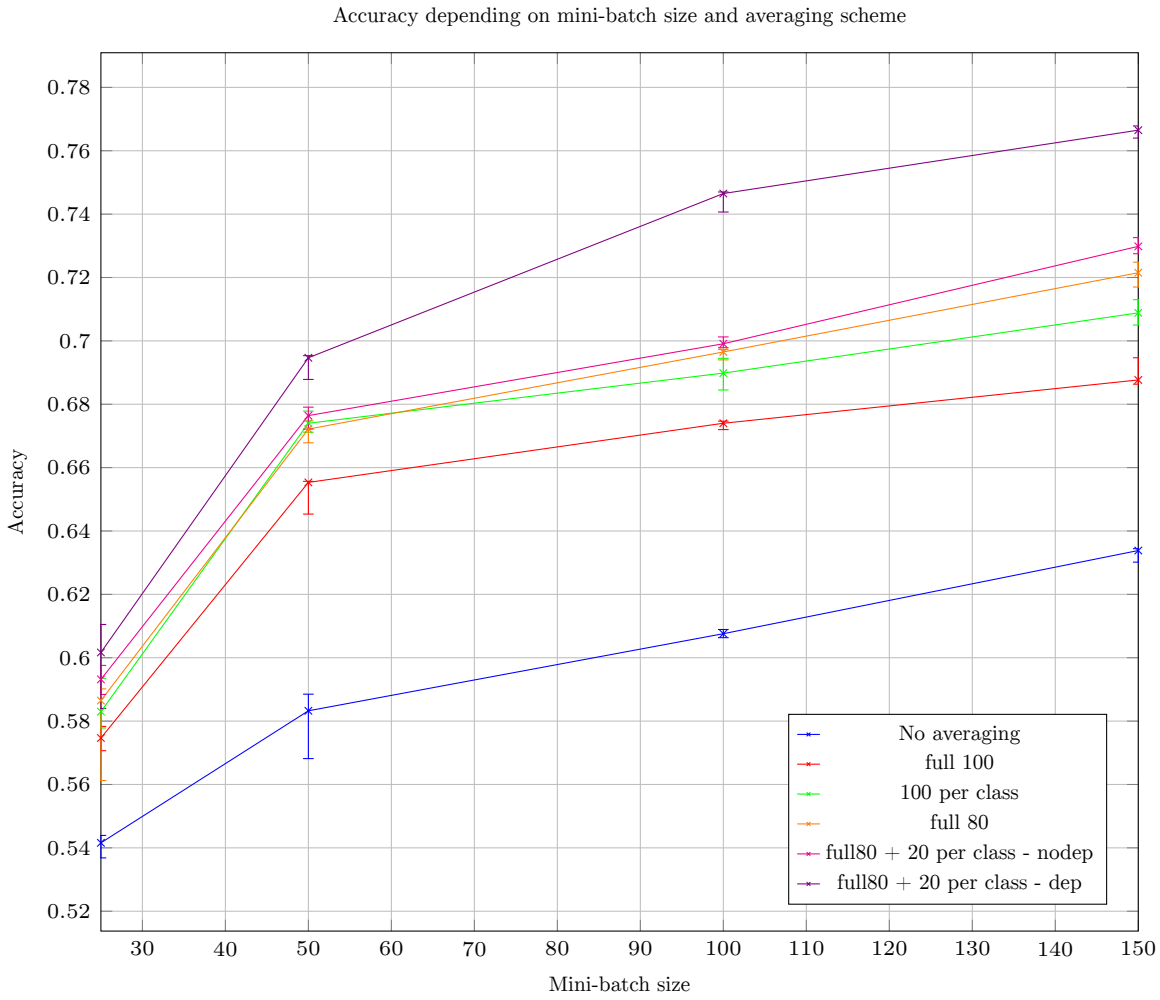


Figure 20: Semi-local averaging on modified MNIST - Accuracy for various mini-batch sizes and averaging schemes

We tested different averaging levels for different training set sizes (ranging from 250 to 3500). The mini-batch size is fixed at 200. We used 16 peers. Each line corresponds to a different averaging level. The averaging process was done after every mini-batch. 50 mini-batches were used for training. Differences between peers are induced by permuting digits 8 and 9 for 7 of the 16 peers.

Results are median of 10 runs. The error bars corresponds to the second and third quintiles. The results are presented in Figure 24.

We see that the lower the averaging level is, the more the training set size is important. 100 averaging is the best for 250 set size, third behind 80 and 60 for 500, bets only 0 for 1000 and last after that. 80 is the best averaging level, except for very small training sets (size 250); 60 being second.

V.3.i Different layouts

Due to the way we induce differences between peers, a simple post-treatment consisting in a permutation inversion would allow 100 averaging to outperform any scheme. We did not try to use this fact before to improve our averaging schemes to remain general. In this test, we evaluate how a partial averaging scheme crafted more specifically for this problem will perform.

Here we test different averaging schemes and layouts for different difference rates. The first four schemes are used with a 784=300=100=10 network layout (4 layers), last 4 784=300=100=10=10 (5 layers). Each line corresponds to a different averaging scheme. Training sets are independent and

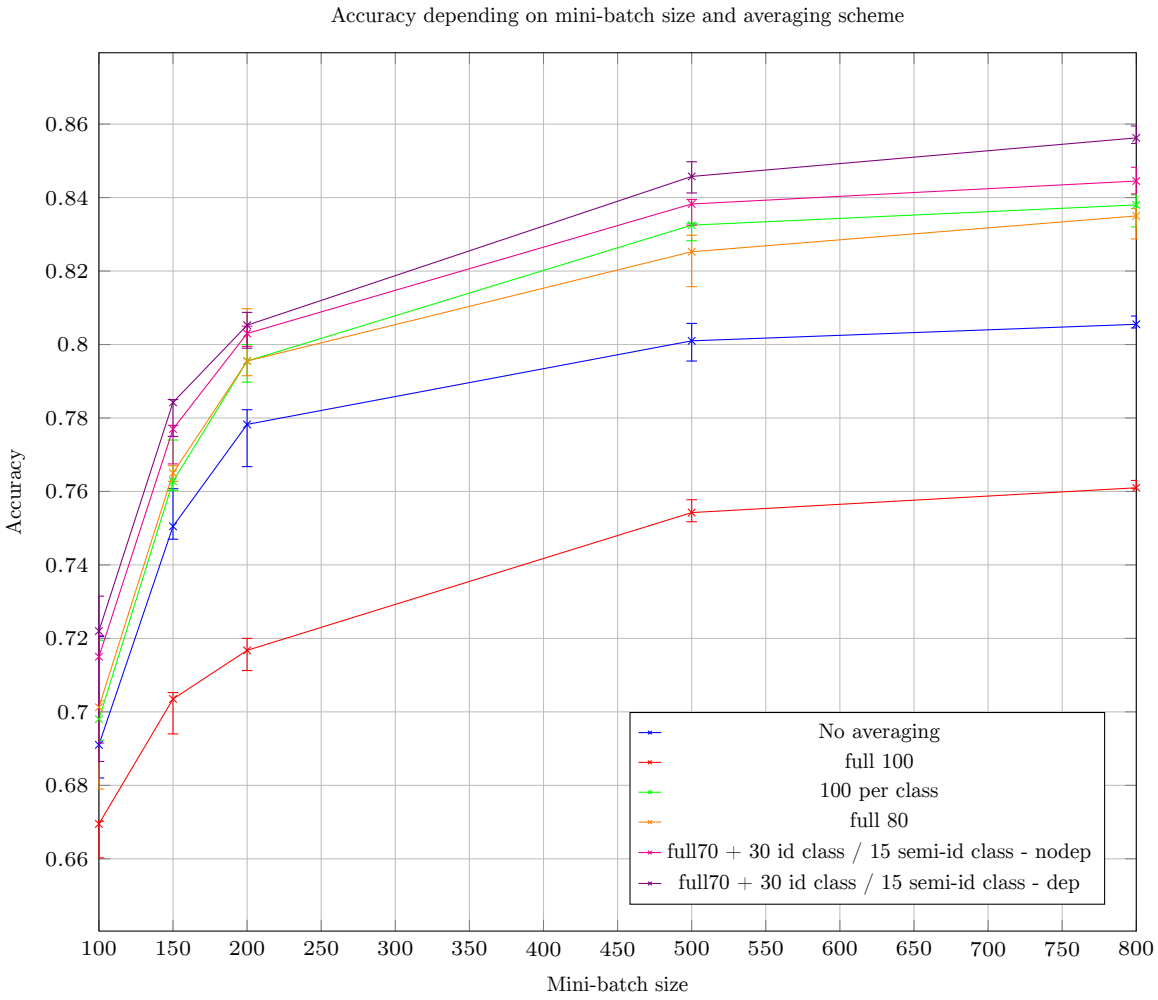


Figure 21: Semi-local averaging with local models on modified MNIST - Accuracy for various mini-batch sizes and averaging schemes

of size 3500; mini-batch size is 500. We used 16 peers. The averaging process was done after every mini-batch. 50 mini-batches were used for training (longer training, since some layouts are deeper than usual). Differences between peers are induced by applying the cycle $(10 - r...9)$ where r is the difference rate (or no permutation if $r = 0$, $r = 1$ being impossible with this system) to 7 of the 16 peers output.

Results are median of 10 runs. The error bars corresponds to the second and third quintiles. The results are presented in Figure 25.

Like previously, complete averaging schemes see their accuracy drop severely when the difference rate increases. The 80 variant used on 784=300=100=10=10 has an accuracy close to no averaging on the same layout, lower for high difference rates. 784=300=100=10=10 layout seems less efficient than 784=300=100=10 in general, but with a 784-300-100-10-0 averaging scheme, it outperforms no averaging on 784=300=100=10 and even (but not much significantly) 784-250-80-10 for high difference rates. 784-300-100-0 appears to be the best scheme in general, beating any other scheme significantly, except in the no difference case and being less affected than 784-250-80-10 by difference rate.

It is important to remember that the 784-300-100-0 was the best here due to the particular nature of the differences between peers, a permutation. This scheme was crafted specifically for this problem, which is a luxury, we could not have done this if the exact nature of the differences between peers was not precisely known. Moreover, this kind of layout is not easy to generalize to semi-local models.

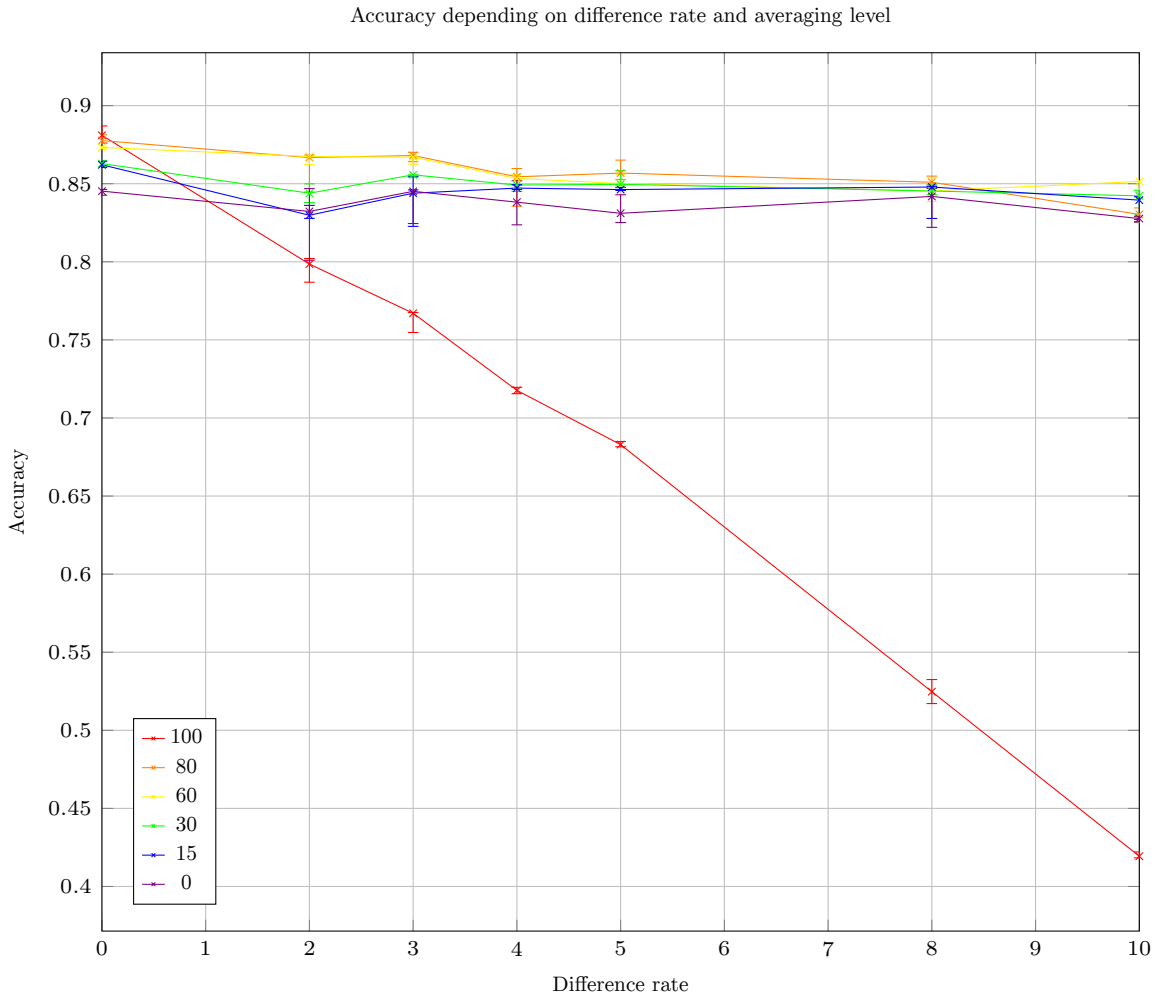


Figure 22: Difference rate on modified MNIST - Accuracy for various difference rates and averaging levels

V.4 Multi-Layer Perceptron - VSN

We finally test MLP on a significantly different task: vehicle recognition from sensors. We use a dataset from [DH04]. This dataset contains data from sensors (notably seismic and acoustic sensors) produced while some (military) vehicle passes near the sensor.

The task proposed is to recognize the class of vehicle (*assault amphibious* or *dragon wagon*) from each sensor's data (binary classification). To allow a MLP to perform this task, the data from each set of sensors (node) is first transformed into 50 seismic and 50 acoustic features (100 total inputs). This process (based on Fast Fourier Transform) is described in the original paper ([DH04]).

For our distributed multi-task setup, we consider each node as a peer. The tasks are similar, since all peer what to classify the same kinds of vehicles from the same kind of data, but different due to the location of sensors influencing their output.

For this experiment we use 16 peers, each with a training set of 50 samples. The test set contains 200 samples. The mini-batch size is 25. We use an MLP with a 100=50=20=1 layout.

The "averaging level" is the number of neurons in the global model (shared by all peers), others neurons being in local models, in the first hidden layer, which contains a total of 50 neurons. The levels we test are 0, 7, 15, 30, 40 and 50. Some neurons from the second hidden layer are also averaged, the exact numbers are, respectively, 0, 3, 6, 12, 16, 20. Each line corresponds to a different mini-batch number (from 100 to 400). The averaging process was done after every mini-batch.

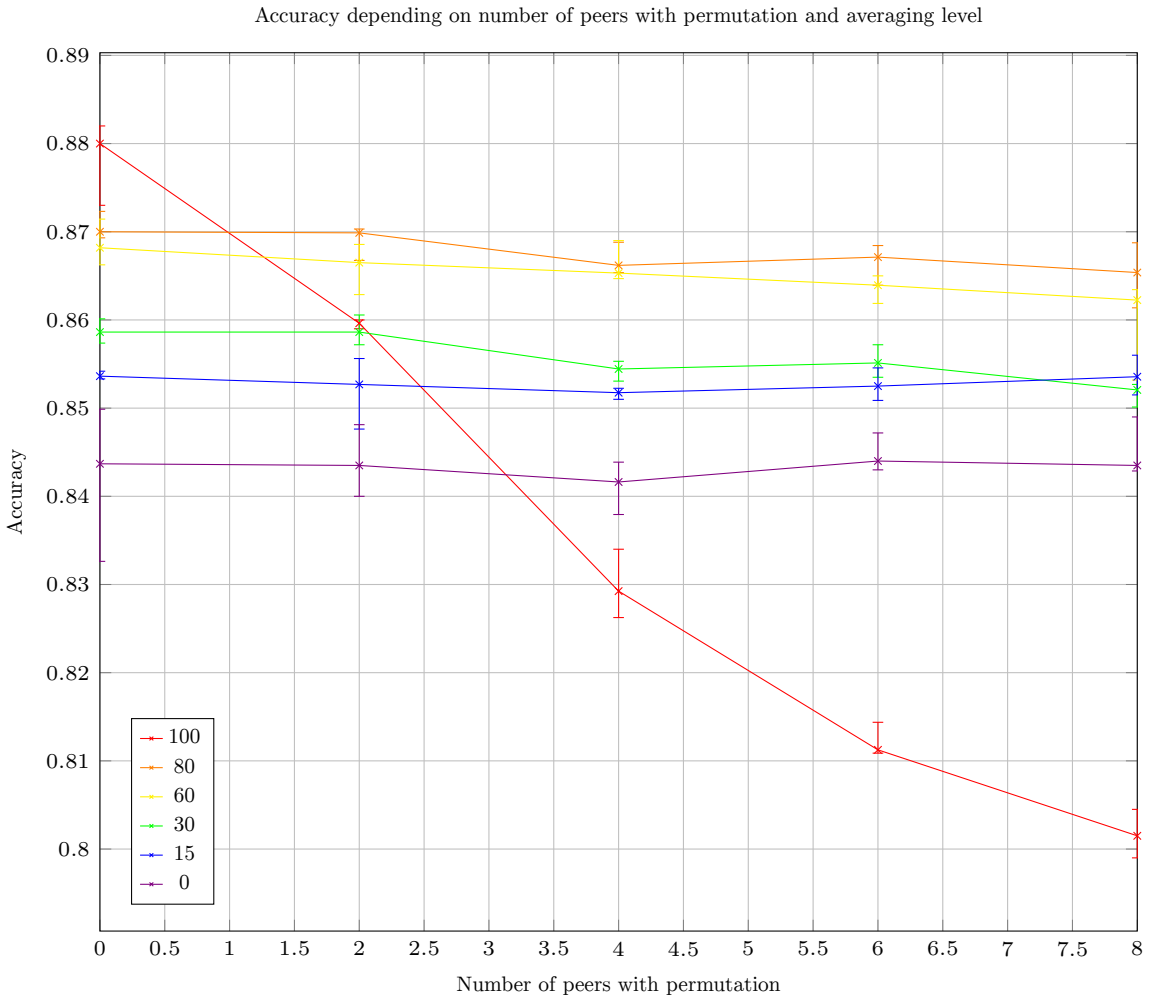


Figure 23: Number of peers with permutation on modified MNIST - Accuracy for various numbers of peers with permutation and averaging levels

Results are median of 10 runs. The error bars corresponds to the second and third quintiles. The results are presented in Figure 26.

This test's results are very similar to our other MLP tests. We get an optimum around 80% averaging.

V.5 Associative neural network

We test associative neural networks on the following task.

We generate binary vectors of size n ($\in 0, 1^n$) with $2m$ 1s (and $n-2m$ 0s). The task of the neural network is to be able to complete each learned vector from an input with only m 1s. For example, the learned vector could be $[0, 0, 1, 0, 1, 1, 0, 0, 1]$, the neural network will be given $[0, 0, 1, 0, 0, 1, 0, 0, 0]$ as input and should be able to return the learned vector.

For training, we give the network the full vectors as input; for testing, we give the network a partial input with only m 1's and observe the output. To complicate the task, we add noise to the vectors: a fixed number z of random bits are flipped (1 becomes 0 and 0 becomes 1) in all vectors (training and testing) before feeding them to the network.

For accuracy evaluation, the task is considered successfully accomplished if the number of matching 1's between the expected output and the actual output is greater than or equal to the number of non-matching 1's. More formally ($\mathbf{c}$ is expected output, $\mathbf{a}$ is actual output), a test is considered successful

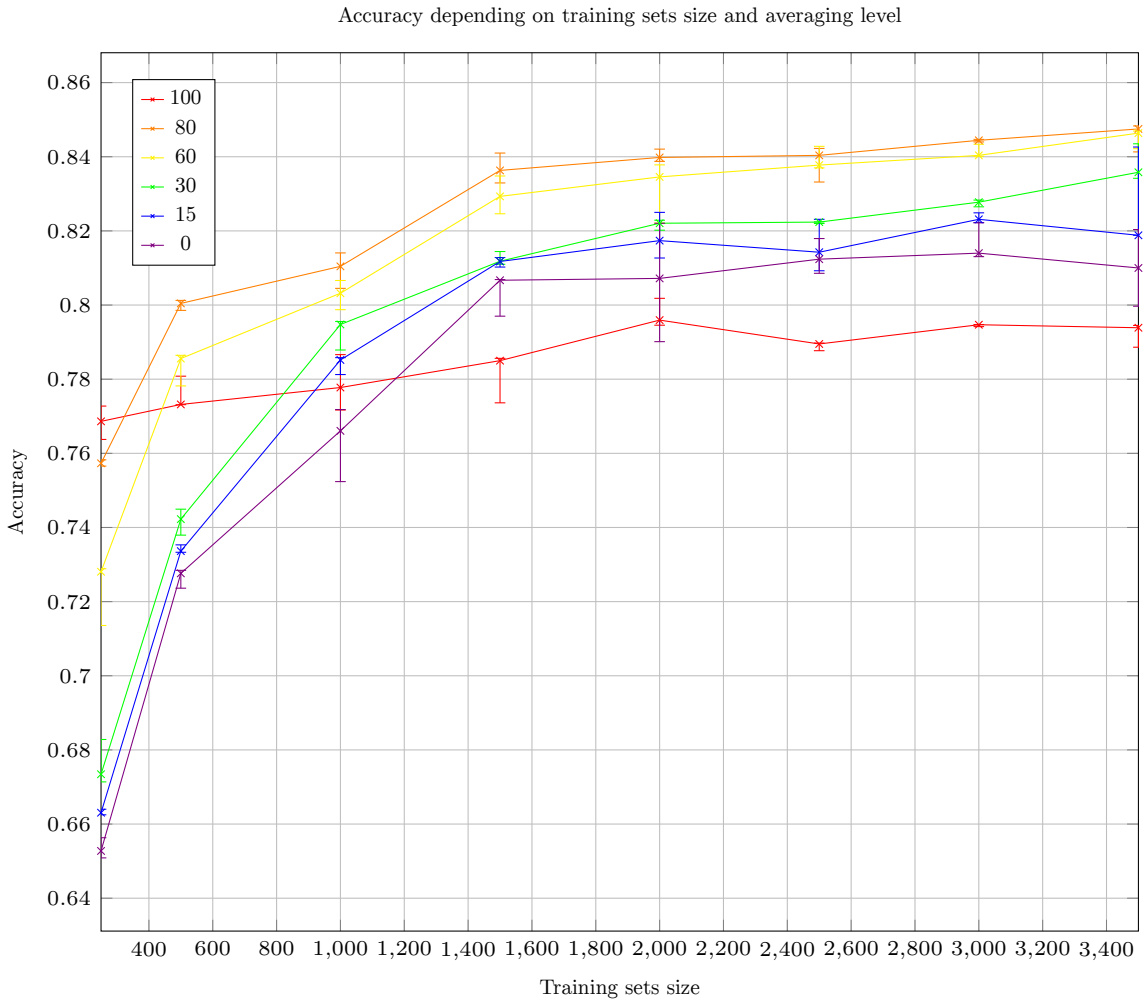


Figure 24: Training set size on modified MNIST - Accuracy for various training sets sizes and averaging levels

if and only if $\mathbf{c} \wedge \mathbf{a}$ contains at least as many 1's as $\mathbf{c} \oplus \mathbf{a}$.

To generate differences between peers, we simply changed, for some peers, half of the 1's that are not in the testing input. For example, from $[0,0,1,0,0,1,0,0,0]$, some peers will have to find $[0,0,1,0,1,1,0,0,1]$ and others $[1,0,1,0,0,1,1,0,0]$.

Our vector generator works in the following way: it takes as input an integer $k \in \mathbb{N}$ and has a periodicity parameter s. The 1 of the vector that will be given as input for both training and testing corresponds to the coordinates $3m(k[s]) + i$ with i ranging from 0 to $m-1$. The 1 of the vector that will be given as input only for training (that should be guessed for testing) corresponds to the coordinates $3m(k[s]) + i$ with i ranging from m to $2m-1$ in the general case and, in the modified case (when we want to make a peer different), with i ranging from $2m$ to $3m-1$. To add noise, we simply flip bits with coordinates $((k/s) + i(n/z))[n]$ ($/$ representing integer division) with i ranging from 0 to $z-1$.

For this test, we use 15 peers, 2 of them having 1 out of 3 vectors modified. Each vector is 50 bits long, with $m = 8$ (non-zero bits) and $z = 2$ (noise). The periodicity of the generators is 3, the batch size is 9 ($k \in [\![0, 9[\![$) and the mini-batch size is 3. The tests consider 100 consecutive values of k (starting at 1000). The learning rate is 0.1 and the activation threshold for neurons is 0.5. The visible layer has 50 neurons (the length of an input/output vector), while the hidden layer comprises 400 neurons.

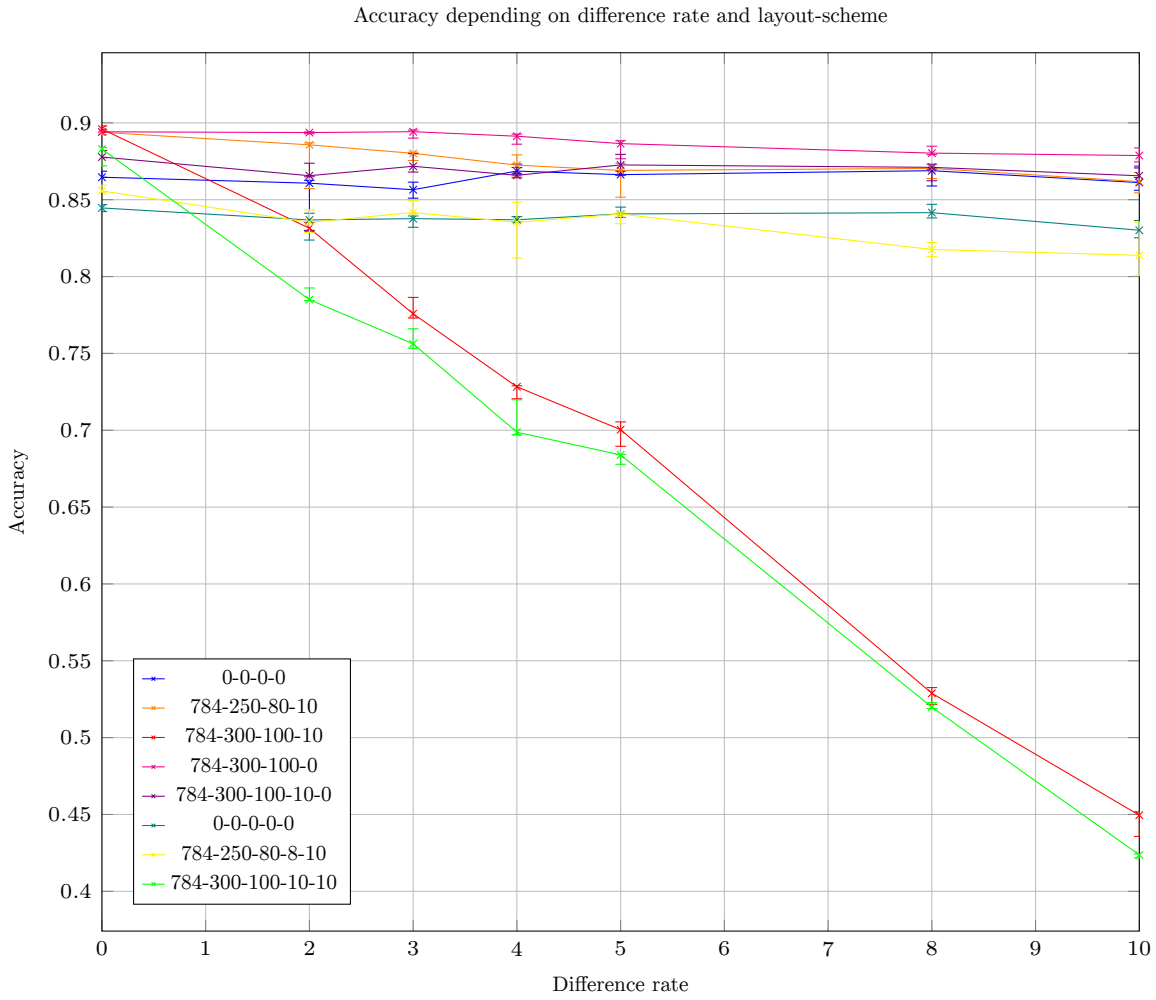


Figure 25: Different layouts on modified MNIST - Accuracy for various difference rates and layouts-schemes

We compare no averaging at all, complete averaging of both layers (visible and hidden) and several partial averaging schemes. For partial averaging schemes, the visible layer was completely averaged and 100, 200, 300 or 350 neurons of the hidden layer were averaged. Each line corresponds to a particular number of mini-batches used: 10, 20, 30 or 40. Averaging was done after each mini-batch.

Results are median of 100 runs. The error bars corresponds to the second and third quintiles. The results are presented in Figure 27.

We observe here that an averaging level of 300 (over 400) seems optimal for lower numbers of mini-batches (10,20), while 350 is for higher numbers (30,40). Except for 40 mini-batches, partial averaging schemes are always the best. Complete averaging was the worst for all tests but the 40 mini-batches one. For 40 mini-batches, complete averaging, while still worse than partial 350, is better than most partial averaging schemes. For all numbers of mini-batches, the optimal averaging level remained around 300-350.

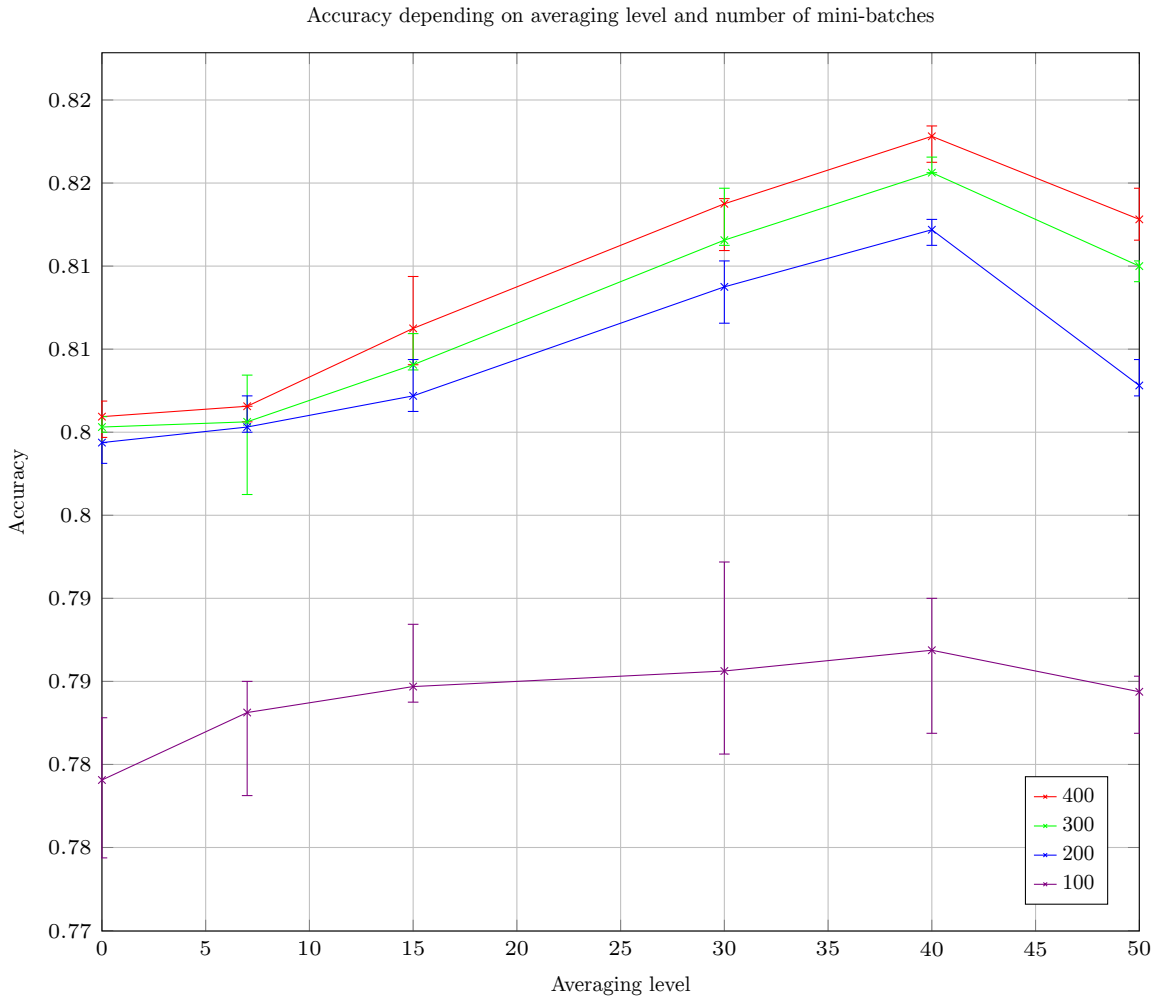


Figure 26: Averaging on VSN - Accuracy for various averaging levels and numbers of mini-batches

V.6 General analysis of results

Tests recapitulation ("Classic" is 784=300=100=#classes)								
Task [NN]	Peers	Batch size	MB size	#MB	Avg every	Layout	Averaging	Difference [#peers]
FEMNIST-A [MLP]	16	360	180	Varies	1	Classic	Varies	Natural
FEMNIST-M [MLP]	16	360	180	Varies	1	Classic	Varies	Natural
FEMNIST-D [MLP]	16	100	50	Varies	1	Classic	Varies	Natural
FEMNIST-A [MLP]	16	360	180	Varies	1	Classic	Varies	Natural
FEMNIST-M [MLP]	16	360	180	Varies	1	Classic	Varies	Natural
FEMNIST-D [MLP]	16	100	50	Varies	1	Classic	Varies	Natural
FEMNIST-A [MLP]	Varies	350	175	200	1	Classic	Varies	Natural
FEMNIST-M [MLP]	Varies	350	175	200	1	Classic	Varies	Natural
FEMNIST-D [MLP]	Varies	100	50	200	1	Classic	Varies	Natural
MNIST [MLP]	16	3500	Varies	30	1	Classic	Varies	(89) [7]
MNIST [MLP]	16	3500	100	30	10	Classic	Varies	(89) [7]
MNIST [MLP]	Varies	3500	500	30	1	Classic	Varies	(89) [1/4]
MNIST [MLP]	12	200	Varies	100	1	Classic	Varies	$\mathfrak{S}_3$ [2 each]
MNIST [MLP]	4	1000	Varies	50	1	Classic	Varies	(89) [1] + (67)(89) [1]
MNIST [MLP]	16	3500	500	30	1	Classic	Varies	Varies [7]
MNIST [MLP]	16	3500	500	30	1	Classic	Varies	(89) [Varies]
MNIST [MLP]	16	Varies	200	50	1	Classic	Varies	(89) [7]
MNIST [MLP]	16	3500	500	50	1	Varies	Varies	Varies [7]
VSN [MLP]	16	50	25	Varies	1	100=50=20=1	Varies	Natural
SBV compl [Assoc]	15	9	3	Varies	1	50<>400	Varies	Half 1s [2]

Our experiments show that our method effectively enables multi-task learning with neural networks.

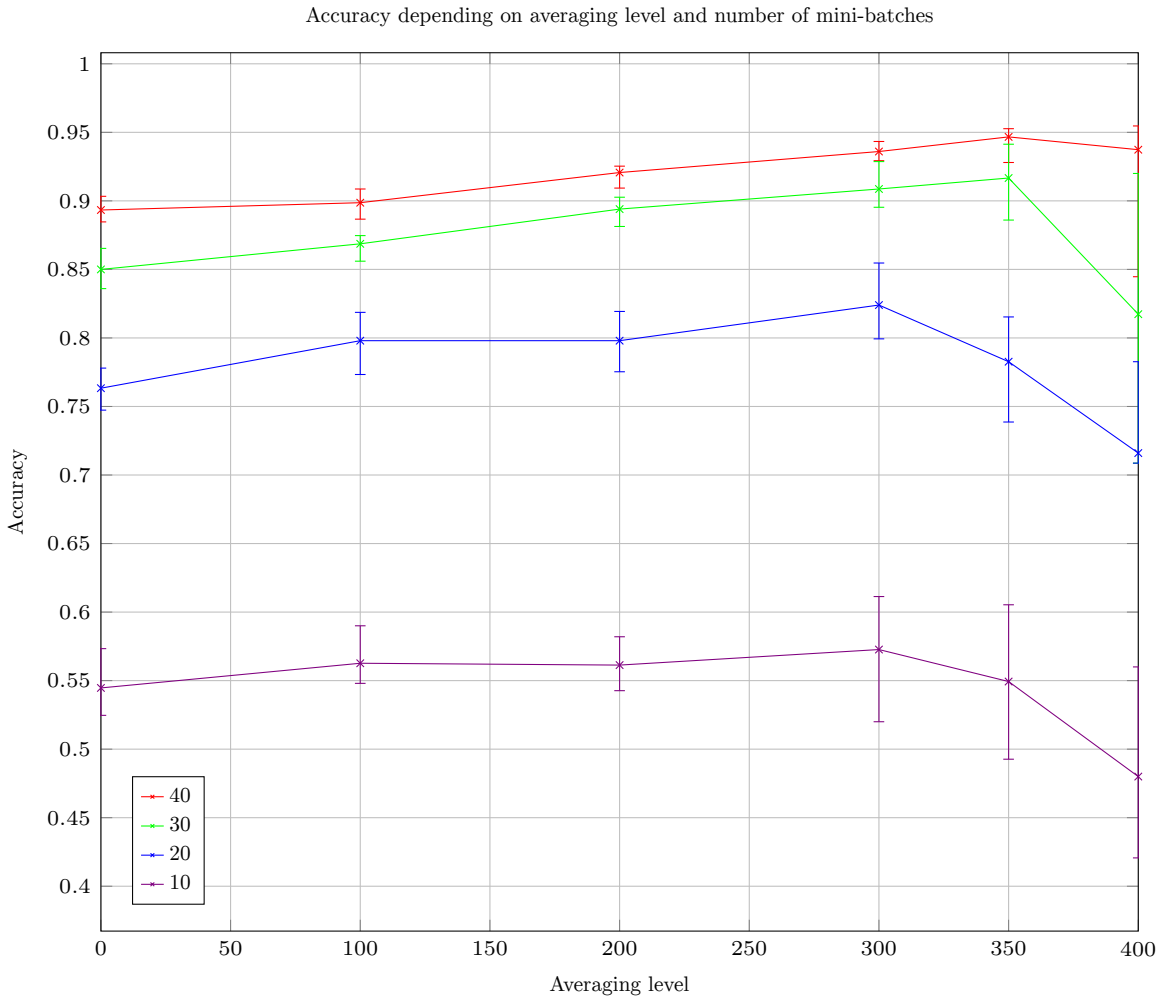


Figure 27: Associative neural network - Accuracy for various averaging levels and number of mini-batches

From the presented results, we can make the following detailed statements.

Partial model averaging yields better results than no averaging or complete averaging on different tasks, synthetic and real, with a variety of parameters, different kinds of neural networks and learning algorithms.

Among the considered tasks, a high level of averaging (60-80%) usually turns out to be the best solution and the level of difference between peers' tasks only has a low influence on the optimal averaging level.

Semi-local averaging and the associated dependency system further improve performance when groups of peers share more similar functions than the whole system.

Multi-task learning with MLP benefits in FEMNIST from the possibility to share portions of layers rather than just complete layers, while more task-specific layouts, averaging limited to specific layers, have proved to be more efficient in modified MNIST but require more knowledge about the differences between peers. A mix of both kinds of schemes could be interesting in certain cases.

While those tests are done on simple cases, they allowed us to examine numerous parameters changes and their effect on the efficiency of our method. This makes this paper a good base for applying our work to real use cases.

VI Related work

Transfer learning was introduced in [PMK91]. While the term "transfer learning" is often used to refer to a particular case of multi-task learning with neural networks, involving reusing pretrained generic first layers with task-specific final layers, transfer learning is actually a much more generic notion. Transfer learning refers to any kind of utilization of knowledge from a certain learning process for another. This is not limited to neural networks and the whole multi-task learning field is actually a subfield of transfer learning [PY09].

A number of researchers have addressed the problem of distributed [Rec+11; Smi+16], decentralized [Bel+18] or federated learning [Kon+16] while focusing on single tasks. Most of these works consider (Stochastic) Gradient Descent, even if some approaches, like federated model averaging [Bre+17], can in principle be applied to other learning algorithms. In the context of model averaging, a recent contribution [Kam+18] suggests that changing the frequency of (model) synchronization depending on the obtained accuracy can reduce communication overhead. We plan to consider this possibility as future work.

The majority of solutions for multi-task learning focus on a local setup, rather than a distributed one [Rud17], but some decentralized solutions exist. The first, to the best of our knowledge, such contribution [OHJ12] proposes a decentralized multi-task learning algorithm limited to linear models, a work later improved in [WKS16] and [Smi+17]. A recent theoretical paper [CST18] proposes the use of kernel methods to learn non-linear models in federated learning. Some researchers have instead proposed a form of decentralized multi-task learning, in which each peer needs to learn a personalized convex model influenced by neighborhood relationships in a collaboration graph [Bel+18]. In a more recent paper, they also presented an algorithm to jointly learn both the personalized convex models and the collaboration graph itself [ZBT19]. In this paper, we do not use a collaboration graph; rather, we express similarities between learning tasks by defining sub-models that are shared with the entire networks (global models) or with a group of peers (semi-local models). Moreover, unlike the above solutions, our approach can optimize non-convex loss functions.

More recently, several preprints have proposed methods for federated multi-task/personalized learning with neural networks. But, none of these works considers a decentralized setup, semi-local models, or flexible layouts like ours. The method for federated multi-task learning proposed in [CB19] relies on a client-server model in which clients operate sequentially, performing their updates one after the other. This negates any speed gains that may result from distribution and results in very poor scalability making this method unsuitable for most practical applications. A better approach, closer to ours, is proposed in [Jia+19], with effective parallelism but still no decentralization or semi-local models. After our own initial preprint publication, in November 2019, other approaches were proposed. An approach that can be considered as a significantly more limited version of our own work, only allowing specific layers to be local (as we show in experiments, this is not always the optimal solution) is proposed in [Ari+19]. Approaches similar to [Jia+19] are presented in [FMO20] and [DKM20].

VII Conclusion

In this work, we introduced an effective solution for distributed multi-task learning with neural networks, applicable with different kinds of learning algorithms and not requiring mandatory prior knowledge about the nature, nor magnitude, of the differences between tasks, while still being able to benefit from such knowledge when available. The simplicity, range of application, and flexibility of our method significantly distinguishes it from existing works.

This work also opens several new directions in different fields. First, in the context of machine learning, we plan to design an algorithm that allows models to be automatically assigned to peers. Second, it would be interesting to reduce communication overhead similarly to [Kam+18]. Third, it would be interesting to apply our method to other kinds of neural networks, like LSTM [HS97; Ger99;

GSC00]. In the context of game theory, it would be interesting to examine the question of what peers gain by participating in the systems, particularly if peers are considered as rational economic agents.

References

[AH15] Subutai Ahmad and Jeff Hawkins. "Properties of Sparse Distributed Representations and their Application to Hierarchical Temporal Memory." In: *CoRR* (2015).

[Ari+19] Manoj Ghuhan Arivazhagan et al. "Federated Learning with Personalization Layers." 2019.

[Bel+18] Aurélien Bellet et al. "Personalized and Private Peer-to-Peer Machine Learning." In: *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics*. Vol. 84. Proceedings of Machine Learning Research. PMLR, 2018, pp. 473–481.

[BFT19] Amaury Bouchra Pilet, Davide Frey, and François Taïani. "Robust Privacy-Preserving Gossip Averaging." In: *Stabilization, Safety, and Security of Distributed Systems*. Vol. 11914. Lecture Notes in Computer Science. Springer International Publishing, 2019, pp. 38–52.

[Bre+17] Hugh Brendan McMahan et al. "Communication-Efficient Learning of Deep Networks from Decentralized Data." In: *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*. Vol. 54. Proceedings of Machine Learning Research. PMLR, 2017, pp. 1273–1282.

[Cal+19] Sebastian Caldas et al. "LEAF: A Benchmark for Federated Settings." 2019.

[CB19] Luca Corinzia and Joachim M. Buhmann. "Variational Federated Multi-Task Learning." 2019.

[Che+17] Min Chen et al. "Disease Prediction by Machine Learning Over Big Data From Healthcare Communities." In: *IEEE Access* 5 (2017), pp. 8869–8879.

[CST18] Sebastian Caldas, Virginia Smith, and Ameet Talwalkar. "Federated Kernelized Multi-Task Learning." In: *SysML Conference 2018*. 2018.

[DH04] Marco F. Duarte and Yu Hen Hu. "Vehicle classification in distributed sensor networks." In: *Journal of Parallel and Distributed Computing* 64.7 (2004), pp. 826–838.

[DKM20] Yuyang Deng, Mohammad Mahdi Kamani, and Mehrdad Mahdavi. "Adaptive Personalized Federated Learning." 2020.

[FMO20] Alireza Fallah, Aryan Mokhtari, and Asuman Ozdaglar. "Personalized Federated Learning: A Meta-Learning Approach." 2020.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[Ger99] Felix A. Gers. "Learning to Forget: Continual Prediction with LSTM." In: *9th International Conference on Artificial Neural Networks: ICANN '99*. 1999, pp. 850–855.

[GSC00] Felix A. Gers, Jürgen Schmidhuber, and Fred Cummins. "Learning to Forget: Continual Prediction with LSTM." In: *Neural Computation* 12.10 (2000), pp. 2451–2471.

[Hop82] John Joseph Hopfield. "Neural networks and physical systems with emergent collective computational abilities." In: *Proceedings of the National Academy of Sciences* 79.8 (1982), pp. 2554–2558.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory." In: *Neural Computation* 9.8 (1997), pp. 1735–1780.

[Jia+19] Yihan Jiang et al. "Improving Federated Learning Personalization via Model Agnostic Meta Learning." 2019.

[JMB05] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. "Gossip-Based Aggregation in Large Dynamic Networks." In: *ACM Transactions on Computer Systems* 23.3 (2005), pp. 219–252.

[Kam+18] Michael Kamp et al. "Efficient Decentralized Deep Learning by Dynamic Model Averaging." In: *Machine Learning and Knowledge Discovery in Databases*. Springer International Publishing, 2018, pp. 393–409.

[Kon+16] Jakub Konečný et al. "Federated Optimization: Distributed Machine Learning for On-Device Intelligence." 2016.

[LB95] Yann LeCun and Yoshua Bengio. "Convolutional networks for images, speech, and time-series." In: *The handbook of brain theory and neural networks*. Vol. 1. MIT Press, 1995, pp. 255–258.

[LeC+98] Yann LeCun et al. "Gradient-based learning applied to document recognition." In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[Nat16] US Government National Institute of Standards and Technology. *NIST Special Database 19*. 2016.

[OHJ12] Róbert Ormándi, István Hegedűs, and Márk Jelasity. "Gossip learning with linear models on fully distributed data." In: *Concurrency and Computation: Practice and Experience* 25.4 (2012), pp. 556–571.

[PMK91] Lorien Y. Pratt, Jack Mostow, and Candace A. Kamm. "Direct Transfer of Learned Information Among Neural Networks." In: *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91)*. 1991, pp. 584–589.

[PY09] Sinno Jialin Pan and Qiang Yang. "A Survey on Transfer Learning." In: *IEEE Transactions on Knowledge and Data Engineering* 22.10 (2009), pp. 1345–1359.

[Rec+11] Benjamin Recht et al. "Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent." In: *Advances in Neural Information Processing Systems 24*. Curran Associates, Inc., 2011, pp. 693–701.

[RHW86] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning Internal Representations by Error Propagation." In: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. Vol. 1. MIT Press, 1986, pp. 318–362.

[Rud17] Sebastian Ruder. "An Overview of Multi-Task Learning in Deep Neural Networks." 2017.

[Shi+16] Weisong Shi et al. "Edge Computing: Vision and Challenges." In: *IEEE Internet of Things Journal* 3.5 (2016), pp. 637–646.

[Smi+16] Virginia Smith et al. "CoCoA: A General Framework for Communication-Efficient Distributed Optimization." In: *Journal of Machine Learning Research* 18 (2016).

[Smi+17] Virginia Smith et al. "Federated Multi-Task Learning." In: *Advances in Neural Information Processing Systems 30*. Curran Associates, Inc., 2017, pp. 4424–4434.

[Smo86] Paul Smolensky. "Information Processing in Dynamical Systems: Foundations of Harmony Theory." In: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. Vol. 1. MIT Press, 1986, pp. 194–281.

[WKS16] Jialei Wang, Mladen Kolar, and Nathan Srerbo. "Distributed Multi-Task Learning." In: *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*. Vol. 51. Proceedings of Machine Learning Research. PMLR, 2016, pp. 751–760.

[ZBT19] Valentina Zantedeschi, Aurélien Bellet, and Marc Tommasi. "Fully Decentralized Joint Learning of Personalized Models and Collaboration Graphs." 2019.

A Advanced formalization

In this section, we present a more formal definition of our partial averaging method and its dependency system. This more generic formalism is defined in such way that it could be expanded to non-neural networks-based learning systems.

As said before, the dependencies are a design choice of engineers under the constraint that the dependency relationship must be an order relationship (antisymmetric and transitive). Since a model does not depend on itself, this order is strict and will be noted $m_2 \prec m_1$ (m_2 depends on m_1).

For this more generic formalism, we use the generic notion of *parameter*, which, in the case of neural network, is a generalization of weights and activation function parameters.

An important feature of neural network is their geometry, all parameters are not equivalent. This fact is largely used by our partial averaging system, so we need to keep it for this more generic formalism. To this end, we introduce the notion of *cell*. In the case of neural network, a cell represents a neuron, though it may be associated with another thing in another class of machine learning systems.

Obviously, the operators creating the models and implementing them locally have to ensure that all models implementation have the same internal topology on all peers, same for the connections between a model's implementation and it's dependencies' implementations. Models' topologies must be defined once before being implemented by peers.

Since we work in a distributed context, we call each peer's parameter vector $\mathbf{z}^p$ and individual parameters z_i^p ($z_i^p \in \mathbb{R}$). Now, we call Π the set of all valid references ("pointer") to some z_i^p ($\sim$ a couple (p, i)). In the following, $\pi \in \Pi$ is some element of Π (π is a generic way to refer to a parameter as a variable, not as its value). Each π is associated with one or more cell(s) (one for activation function parameters, two for weights). We denote $C(\pi)$ the set of cells associated with a parameter. We also denote by $\mu(c)$ the model a cell c is associated with. Now, we can define $M(\pi) = \{\mu(c) | c \in C(\pi)\}$ the set of all models associated with a parameter π, which we could also write $M(\pi) = \mu[C(\pi)]$.

Noting $\mathcal{C}$ the set of all cells and $\mathcal{M}$ the set of all models, we have: $C : \Pi \to 2^{\mathcal{C}}$, $\mu : \mathcal{C} \to \mathcal{M}$ and $M : \Pi \to 2^{\mathcal{M}}$. Taking $Ц : 2^{\mathcal{C}} \to 2^{\mathcal{M}}$ as the set variant of μ: $Ц(X) = \{\mu(c) | c \in X\}$, we can write $M = Ц \circ C$ ($M = \mu \circ C$ would have been an abuse of notation).

Figure 28 summarizes the functions and sets we introduced.

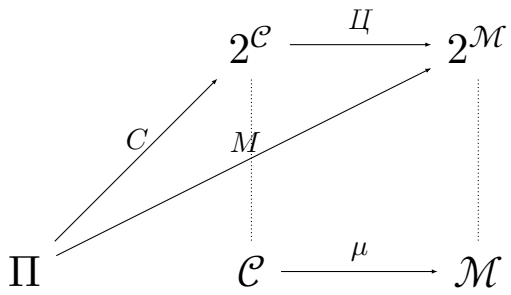


Figure 28: Sets and functions we defined. Π: set of all parameters. $\mathcal{C}$: set of all cells. $\mathcal{M}$: set of all models.

With the assumptions we made one models, we can assume that there is a generic way to index all parameters associated with one model on all peers implementing it, so those weight can effectively be averaged[4].

[4] A possible solution is to index all cells (function $\xi(c)$), then, for each cell, index all the parameters (of the considered model, others must be ignored) associated with the cell (function $\xi_c(\pi)$). Once this is done, you take for each parameter the set $\Xi(\pi) = \{n = 2^{\xi(c)} \times 3^{\xi_c(\pi)} | c \in C(\pi)\}$ (each n is unique due to the unicity of prime numbers decomposition). Then index all parameters based on $\Xi(\pi)$ (the set of all finite sets of integers can be indexed by integers; just take, for each integer n in the set, the nth prime number and multiply all those primes, you have a unique index due to the unicity of prime numbers decomposition).

Now, we want to find a minimal set of models (ideally a singleton) to associate each π with for averaging. To this end, we define the reduced set of models associated with π as follows $R(\pi) = \{m | m \in M(\pi) \wedge \neg\exists m' \in M(\pi), m' \prec m\}$. In a less formal language, we obtain $R(\pi)$ from $M(\pi)$ by removing models which have other models depending on them in the set. For example, if $M(\pi) = \{m_1, m_2, m_3\}$ and $m_2 \prec m_1$ then $R(\pi) = \{m_2, m_3\}$. Let us prove the existence and unicity of $R(\pi)$.

Existence (constructive):

To construct $R(\pi)$ you can simply remove from $M(\pi)$ all elements which are greater than some other element of $M(\pi)$.

Unicity:

Suppose that we have two set different sets $R(\pi)$ and $R(\pi)'$ all $\in M(\pi)$. We can suppose without loss of generality that there is some m such that $m \in R(\pi) \wedge m \notin R'(\pi)$. Then there are two cases. First case, $\exists m' \in M(\pi), m' \prec m$ which is in contradiction with the definition of $R(\pi)$. Second, $\forall m' \in M(\pi), \neg(m' \prec m)$, but in that case, m' should be $\in R'(\pi)$ by definition, so this is also a contradiction.

If $R(\pi)$ is a singleton, the parameter π will be averaged as part of the unique element (model) of $R(\pi)$. If $R(\pi)$ is not a singleton, the less complex solution is to keep π local, but theoretically π could be averaged among all peers implementing all elements (models) of $R(\pi)$. In that case, $R(\pi)$ as a set of models could be considered equivalent to a single model m such that $\forall m' \in R(\pi), m \prec m'$. This would however add significant practical complexity.

Note that mathematically we could have used something stronger than $R(\pi)$, a minimal lower bound of $M(\pi)$ (a minimal set L of elements such that $\forall m \in M(\pi), (m \in L \vee \exists m' \in L, m' \prec m)$) but such a set is not necessarily unique (if you have m_1, m_2, m_3, m_4, with $m_3 \prec m_1, m_3 \prec m_2, m_4 \prec m_1, m_4 \prec m_2$, then $\{m_3\}$ and $\{m_4\}$ both correspond to the definition for set $\{m_1, m_2\}$) and this has not much sens from engineering point of view due to the fact that such a set could include models not associated with any cell associated with π.

Figure 29 gives an example of what could be a neural network with our generic formalism. In this figure we have for example: $C(\pi_1) = \{c_1\}$, $C(\pi_{14}) = \{c_2, c_5\}$, $C(\pi_9) = \{c_1, c_3\}$, $\mu(c_1) = m_1$, $M(\pi_1) = \{m_1\}$, $M(\pi_{14}) = \{m_1\}$, $M(\pi_9) = \{m_2\}$.

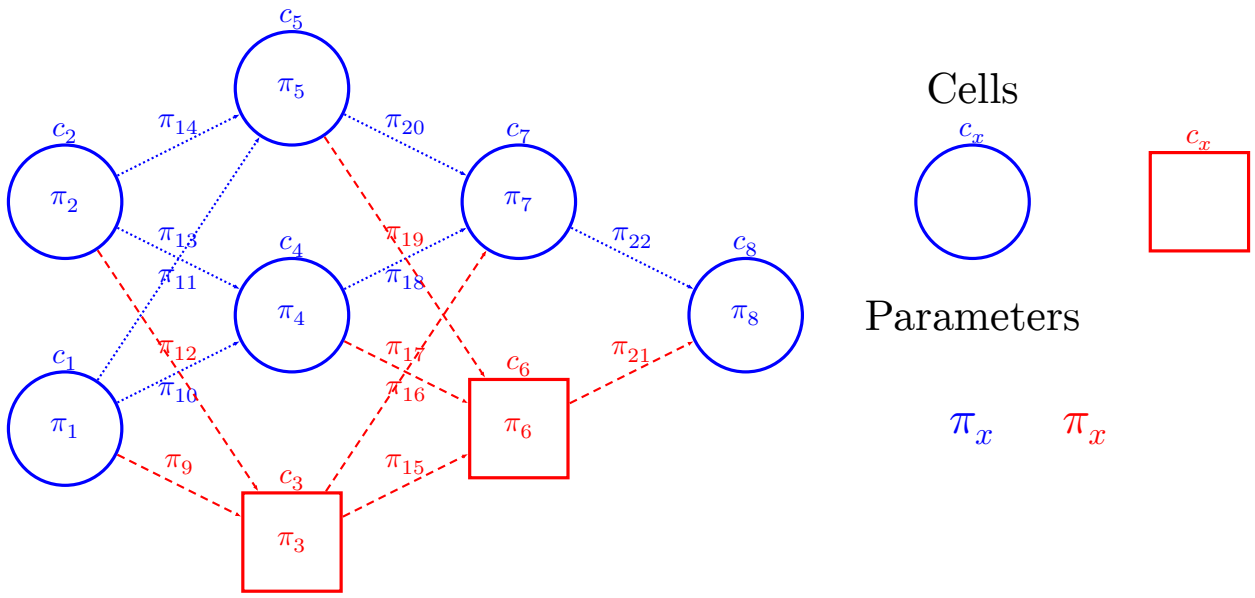


Figure 29: A neural network in our generic formalism. Two models with m_2 depending on m_1. Parameters can be both, activation function parameters (inside cells/neurons) or weights (on edges).