A $\mathcal{L}_{\text{ML}}$ IS A LOWER-BOUND TO THE MARGINAL LIKELIHOOD

In this section, we show that the objective in equation 2 is a lower-bound on the marginal likelihood, under a mild assumption on each approximate posterior $q_k(\boldsymbol{w})$. The aim is to approximate:

$$\log p(\mathcal{D}|\psi) = \sum_{k=1}^{C} \log p(\mathcal{D}_k|\mathcal{D}_{1:k-1}, \psi) \tag{5}$$

Our partitioned approximation is given by:

$$\sum_{k=1}^{C} \mathbb{E}_{q_{k-1}(\boldsymbol{w})} \left[\log p(\mathcal{D}_k|\boldsymbol{w}, \psi)\right] \tag{6}$$

We can get the equation for the gap between quantities in 5 and 6:

$$\begin{aligned}
\text{gap} &= \sum_{k=1}^{C} \log p(\mathcal{D}_k|\mathcal{D}_{1:k-1}, \psi) - \sum_{k=1}^{C} \mathbb{E}_{q_{k-1}(\boldsymbol{w})} \left[\log p(\mathcal{D}_k|\boldsymbol{w}, \psi)\right] && (7)\\
&= \sum_{k=1}^{C} \mathbb{E}_{q_{k-1}(\boldsymbol{w})} \left[\log p(\mathcal{D}_k|\mathcal{D}_{1:k-1}, \psi) - \log p(\mathcal{D}_k|\boldsymbol{w}, \psi)\right] && (8)\\
&= \sum_{k=1}^{C} \mathbb{E}_{q_{k-1}(\boldsymbol{w})} \left[\log \frac{p(\mathcal{D}_k|\mathcal{D}_{1:k-1}, \psi)}{p(\mathcal{D}_k|\boldsymbol{w}, \psi)}\right] && (9)\\
&= \sum_{k=1}^{C} \mathbb{E}_{q_{k-1}(\boldsymbol{w})} \left[\log \frac{\overbrace{p(\boldsymbol{w}|\mathcal{D}_{1:k}, \psi) p(\mathcal{D}_k|\mathcal{D}_{1:k-1}, \psi)}^{p(\boldsymbol{w}, \mathcal{D}_k|\mathcal{D}_{1:k-1})} p(\boldsymbol{w}|\mathcal{D}_{1:k-1}, \psi)}{p(\boldsymbol{w}|\mathcal{D}_{1:k}, \psi) \underbrace{p(\mathcal{D}_k|\boldsymbol{w}, \psi) p(\boldsymbol{w}|\mathcal{D}_{1:k-1}, \psi)}_{p(\boldsymbol{w}, \mathcal{D}_k|\mathcal{D}_{1:k-1})}}\right] && (10)\\
&= \sum_{k=1}^{C} \mathbb{E}_{q_{k-1}(\boldsymbol{w})} \left[\log \frac{p(\boldsymbol{w}|\mathcal{D}_{1:k-1}, \psi)}{p(\boldsymbol{w}|\mathcal{D}_{1:k}, \psi)}\right] && (11)\\
&= \sum_{k=1}^{C} D_{\text{KL}} \left[q_{k-1}(\boldsymbol{w}) \| p(\boldsymbol{w}|\mathcal{D}_{1:k}, \psi)\right] - D_{\text{KL}} \left[q_{k-1}(\boldsymbol{w}) \| p(\boldsymbol{w}|\mathcal{D}_{1:k-1}, \psi)\right] && (12)
\end{aligned}$$

We now make two assumptions

- $D_{\text{KL}} \left[q_{k-1}(\boldsymbol{w}) \| p(\boldsymbol{w}|\mathcal{D}_{1:k}, \psi)\right] \geq D_{\text{KL}} \left[q_k(\boldsymbol{w}) \| p(\boldsymbol{w}|\mathcal{D}_{1:k}, \psi)\right]$. This is motivated from the fact that $q_k(\boldsymbol{w})$ is trained on all data chunks $\mathcal{D}_{1:k}$ so it is expected to be a better approximation to the posterior $p(\boldsymbol{w}|\mathcal{D}_{1:k})$, compared to $q_{k-1}(\boldsymbol{w})$ which is only trained on $\mathcal{D}_{1:k-1}$.
- $D_{\text{KL}} \left[q_{C-1}(\boldsymbol{w}) \| p(\boldsymbol{w}|\mathcal{D}_{1:C}, \psi)\right] \geq D_{\text{KL}} \left[q_0(\boldsymbol{w}) \| p(\boldsymbol{w})\right]$. Since we are free to choose the approximate posterior before seeing any data — $q_0(\boldsymbol{w})$—, we can set it to be equal to the prior $p(\boldsymbol{w})$ which, together with the positivity of the KL divergence, trivially satisfies this assumption.

Therefore, by rearranging Eq. 12 and using our two assumptions we have that the gap is positive

$$\begin{aligned}
\text{gap} = &-D_{\text{KL}} \left[q_0(\boldsymbol{w}) \| p(\boldsymbol{w})\right] + D_{\text{KL}} \left[q_{C-1}(\boldsymbol{w}) \| p(\boldsymbol{w}|\mathcal{D}_{1:C}, \psi)\right] + \\
&\sum_{k=1}^{C} D_{\text{KL}} \left[q_{k-1}(\boldsymbol{w}) \| p(\boldsymbol{w}|\mathcal{D}_{1:k}, \psi)\right] - D_{\text{KL}} \left[q_k(\boldsymbol{w}) \| p(\boldsymbol{w}|\mathcal{D}_{1:k}, \psi)\right] \geq 0,
\end{aligned} \tag{13}$$

and our approximation is a lower bound to the marginal likelihood, *i.e.*,

$$\log p(\mathcal{D}|\psi) \geq \sum_{k=1}^{C} \mathbb{E}_{q_{k-1}(\boldsymbol{w})} \left[\log p(\mathcal{D}_k|\boldsymbol{w}, \psi)\right]. \tag{14}$$

B PARTITIONED NETWORKS AS A SPECIFIC APPROXIMATION TO THE MARGINAL LIKELIHOOD

In this section of the appendix, we show that the partitioned neural networks we presented in the paper are a particular instance of the approximation to the marginal likelihood shown in equation 2.

Consider a dataset $\mathcal{D}$ comprised of C shards, i.e. $\mathcal{D} = (\mathcal{D}_1, \dots, \mathcal{D}_C)$, along with a model, e.g., a neural network, with parameters $\boldsymbol{w} \in \mathbb{R}^{D_w}$, a prior $p(\boldsymbol{w}) = \prod_{j=1}^{D_w} \mathcal{N}(\boldsymbol{w}_j|0, \lambda)$ and a likelihood $p(\mathcal{D}|\boldsymbol{w}, \psi)$ with hyperparameters ψ. Assuming a sequence over the dataset chunks, we can write out the true marginal likelihood as

$$\log p(\mathcal{D}|\psi) = \sum_k \log p(\mathcal{D}_k|\mathcal{D}_{1:k-1}, \psi) = \sum_k \log \mathbb{E}_{p(\boldsymbol{w}|\mathcal{D}_{1:k-1},\psi)} \left[p(\mathcal{D}_k|\boldsymbol{w}, \psi)\right] \tag{15}$$

$$\geq \sum_k \mathbb{E}_{p(\boldsymbol{w}|\mathcal{D}_{1:k-1},\psi)} \left[\log p(\mathcal{D}_k|\boldsymbol{w}, \psi)\right]. \tag{16}$$

Since the true posteriors $p(\boldsymbol{w}|\mathcal{D}_{1:j}, \psi)$ for $j \in \{1, \dots, C\}$ are intractable, we can use variational inference to approximate them with $q_{\phi_j}(\boldsymbol{w})$ for $j \in \{1, \dots, C\}$, with ϕ_j being the to-be-optimized parameters of the j'th variational approximation. Based on the result from Appendix A, when $q_{\phi_j}(\boldsymbol{w})$ are optimized to match the respective posteriors $p(\boldsymbol{w}|\mathcal{D}_{1:j}, \psi)$, we can use them to approximate the marginal likelihood as

$$\log p(\mathcal{D}|\psi) \geq \sum_k \mathbb{E}_{q_{\phi_{k-1}}(\boldsymbol{w})} \left[\log p(\mathcal{D}_k|\boldsymbol{w}, \psi)\right]. \tag{17}$$

Partitioned networks correspond to a specific choice for the sequence of approximating distribution families $q_{\phi_k}(\boldsymbol{w})$. Specifically, we partition the parameter space $\boldsymbol{w}$ into C chunks, i.e., $\boldsymbol{w}_k \in \mathbb{R}^{D_{wk}}$, such that $\sum_k D_{wk} = D_w$, and we associate each parameter chunk $\boldsymbol{w}_k$ with a data shard $\mathcal{D}_k$. Let $r_{\phi_k}(\boldsymbol{w}_k)$ be base variational approximations over $\boldsymbol{w}_k$ with parameters ϕ_k. Each approximate distribution $q_{\phi_k}(\boldsymbol{w})$ is then defined in terms of these base approximations, i.e.,

$$q_{\phi_k}(\boldsymbol{w}) = \left(\prod_{j=1}^{k-1} r_{\phi_j}(\boldsymbol{w}_j)\right) r_{\phi_k}(\boldsymbol{w}_k) \left(\prod_{m=k+1}^{K} r_0(\boldsymbol{w}_m)\right) \tag{18}$$

where $r_0(\cdot)$ is some base distribution with no free parameters. In accordance with the assumptions in appendix A, we can then fit each $q_{\phi_k}(\boldsymbol{w})$ by minimising the KL-divergence to $p(\boldsymbol{w}|\mathcal{D}_{1:k}, \psi)$ – the posterior after seeing k chunks:

$$D_{\mathrm{KL}}\left[q_{\phi_k}(\boldsymbol{w}) \| p(\boldsymbol{w}|\mathcal{D}_{1:k}, \psi)\right] = -\mathbb{E}_{q_{\phi_k}(\boldsymbol{w})}[\log p(\mathcal{D}_{1:k}|\boldsymbol{w}, \psi)] + D_{\mathrm{KL}}\left[q_{\phi_k}(\boldsymbol{w}) \| p(\boldsymbol{w})\right]$$

$$+ \log p(\mathcal{D}_{1:k}|\psi) \tag{19}$$

$$\tag{20}$$

Finding the optimum with respect to ϕ_k:

$$\arg\min_{\phi_k} \; D_{\mathrm{KL}}\left[q_{\phi_k}(\boldsymbol{w}) \| p(\boldsymbol{w}|\mathcal{D}_{1:k}, \psi)\right] = \tag{21}$$

$$= \arg\min_{\phi_k} -\mathbb{E}_{q_{\phi_k}(\boldsymbol{w})}[\log p(\mathcal{D}_{1:k}|\boldsymbol{w}, \psi)] + D_{\mathrm{KL}}\left[q_{\phi_k}(\boldsymbol{w}) \| p(\boldsymbol{w})\right] \tag{22}$$

$$= \arg\min_{\phi_k} -\mathbb{E}_{q_{\phi_k}(\boldsymbol{w})}[\log p(\mathcal{D}_{1:k}|\boldsymbol{w}, \psi)]$$

$$+ D_{\mathrm{KL}}\left[\left(\prod_{j=1}^{k-1} r_{\phi_j}(\boldsymbol{w}_j)\right) r_{\phi_k}(\boldsymbol{w}_k) \left(\prod_{m=k+1}^{K} r_0(\boldsymbol{w}_m)\right) \| \prod_{i}^{K} p(\boldsymbol{w}_i)\right] \tag{23}$$

$$= \arg\min_{\phi_k} -\mathbb{E}_{q_{\phi_k}(\boldsymbol{w})}[\log p(\mathcal{D}_{1:k}|\boldsymbol{w}, \psi)] + D_{\mathrm{KL}}\left[r_{\phi_k}(\boldsymbol{w}_k) \| p(\boldsymbol{w}_k)\right]. \tag{24}$$

We can now obtain partitioned networks by assuming that $r_{\phi_k}(\boldsymbol{w}_k) = \mathcal{N}(\boldsymbol{w}_k|\phi_k, \nu\mathbf{I})$ for $k \in \{1, \dots, C\}$, $r_0(\mathbf{w}) = \mathcal{N}(\boldsymbol{w}|\hat{\boldsymbol{w}}, \nu\mathbf{I})$, with $\hat{\boldsymbol{w}}$ being the parameters at initialization (i.e., before we

update them on data) and taking $\nu \to 0$, i.e., in machine-precision, the weights are deterministic. As noted in Section I.1, we scale the weight-decay regularizer for ϕ_k (whenever used) differently for each partition k, such that it can be interpreted as regularization towards a prior. In the experiments where we do not regularize ϕ_k according to $p(\mathbf{w}_k)$ when we optimize them, this implicitly corresponds to $\lambda \to \infty$ (i.e. the limiting behaviour when the variance of $p(\mathbf{w})$ goes to infinity), which makes the contribution of the regularizer negligible.

C Partitioning Schemes

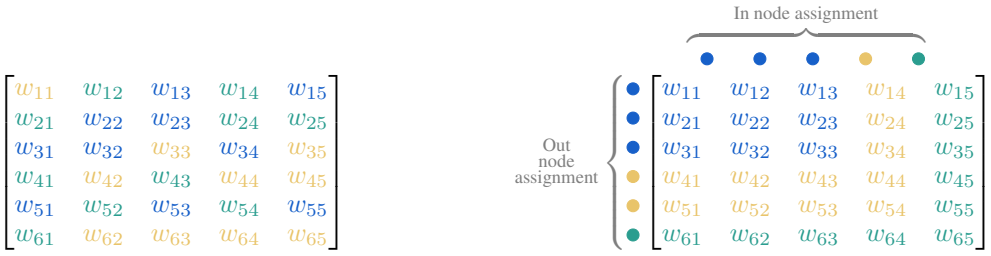
There are several ways in which we could aim to partition the weights of a neural network. Throughout the experimental section 5, we partition the weights by assigning a fixed proportion of weights in each layer to a given partition at random. We call this approach **random weight partitioning**.

We also experimented with other partitioning schemes. For example, we tried assigning a fixed proportion of a layer's outputs (*e.g.*, channels in a convolution layer) to each partition. All weights in a given layer that a specific output depends on would then be assigned to that partition. We call this approach **node partitioning**. Both approaches are illustrated in Figure 4.

One benefit of the node partitioning scheme is that it makes it possible to update multiple partitions with a single batch; This is because we can make a forward pass at each linear or convolutional layer with the full network parameters $\boldsymbol{w}$, and, instead, mask the appropriate inputs and outputs to the layer to retrieve an equivalent computation to that with $\boldsymbol{w}_s^{(k)}$. The gradients also need to be masked on the backward pass adequately. No such simplification is possible with the random weight partitioning scheme; if we were to compute a backward pass for a single batch of examples using different subnetworks for each example, the memory overhead would grow linearly with the number of subnetworks used.

In initial experiments, we found both *random weight partitioning* and *node partitioning* performed similarly. In the experimental section 5, we focused on the former, as it's easier to reason about with relation to *e.g.*, dropout.

Throughout this work, partitioning happens prior to initiating training, and remains fixed throughout. It might also be possible to partition the network parameters dynamically during training, which we leave for future work.



(a) Random weight partitioned

(b) Node partitioned

Figure 4: Figures showing how the weights within a single weight matrix $W \in \mathbb{R}^{6\times5}$ for a linear layer would be partitioned.

D Scalability

In the paper, we claim that our method is scalable compared to Schwöbel et al. (2021) and Immer et al. (2022). What constraints the scalability of the mentioned prior works, however, is different.

For the Last Layer Marginal Likelihood, although the approach works on small datasets such as PCAM (Veeling et al., 2018) and MNIST, the authors report that they were unable to learn invariances

on larger datasets such as CIFAR10. In (Schwöbel et al., 2021, section 7), they explore the issue of scalability in more detail, and showcase that last layer marginal likelihood is insufficient.

Differentiable Laplace performs well, even on more complex datasets, such as CIFAR10. Their scalability, however, is limited by the computational and memory complexity of their method, which we go into in more detail in the section below.

D.1 Complexity Analysis

First, we consider the scalability of our algorithm in terms of computational and memory complexity. In particular, we show that our method scales much more favourably compared to Differentiable Laplace (Immer et al., 2022).

We present our analysis for a feed-forward model of depth L, with layer widths D[8]. In order to directly compare to Immer et al. (2022) and Benton et al. (2020), we consider the complexities in the invariance learning setup (Benton et al., 2020; van der Wilk et al., 2018) with S augmentation samples. In other experiments, hyperparameter optimization setups, S can be taken to be 1. The notation is summarized in Table 5.

N	Number of datapoints in dataset $\mathcal{D}$
N_B	Batch size
S	Number of augmentation samples[9]
C	Output size (number of classes)
D	Feedforward network layer widths
L	Feedforward network depth
P	Number of parameters (s.t. $\mathcal{O}(P) = \mathcal{O}(LD^2 + DC)$)

Table 5: Notation for complexity analysis.

We consider the computational and memory costs of 1) obtaining a gradient with respect to the parameters 2) obtaining a gradient with respect to the hyperparameters, and 3) computing the value of the model/hyperparameter selection objective for each method. All analysis assumes computation on a Monte-Carlo estimate of the objective on a single batch of data.

In Tables 6 and 7, we assume that $C < D$, and hence, for the clarity of comparison, sometimes fold a factor depending C into a factor depending on D if it's clearly smaller. This hiding of the factors was only done for Differentiable Laplace, which is the worst scaling method.

D.1.1 Computational Complexity

	Param. Backward	Hyperparam. Backward	Hyperparam. Objective
Partitioned	$\mathcal{O}(N_B PS)$	$\mathcal{O}(N_B PS)$	$\mathcal{O}(N_B PS)$
Augerino	$\mathcal{O}(N_B PS)$	$\mathcal{O}(N_B PS)$	$\mathcal{O}(N_B PS)$
Diff. Laplace	$\mathcal{O}(N_B PS)$	$\mathcal{O}(N_B PS + NCP + NCDLS + LD^3)$	$\mathcal{O}(NPS + NCP + NCDLS + LD^3)$

Table 6: **Computational Complexities**. The two terms highlighted for Augerino can be computed in a single backward pass. For Differentiable Laplace, the terms in blue can be amortised over multiple hyperparameter backward passes. That is why, in their method, they propose updating the hyperparameters once every epoch on (possibly) multiple batches of data, rather than once on every batch as is done with Partitioned Networks and Augerino.

[8]This is for the ease of comparison. Same upper bound complexities will hold for a network of variable sizes D_ℓ for $\ell \in [L]$, where $D = \max_\ell D_\ell$

[9]Only relevant for invariance learning.

D.1.2 MEMORY COMPLEXITY

The memory complexities for Partitioned Networks, Augerino, and Differentiable Laplace are shown in Table 7. Crucially, the memory required to update the hyperparameters for Differentiable Laplace scales as $\mathcal{O}(N_B S L D^2 + P)$, with a term depending on the square of the network widths. This can become prohibitively expensive for larger models, and is likely the reason why their paper only considers experiments on architectures with widths up to a maximum of 256.

	Param. Backward	Hyperparam. Backward	Hyperparam. Objective
Partitioned	$\mathcal{O}(N_B S L D + P)$	$\mathcal{O}(N_B S L D + P)$	$\mathcal{O}(N_B S D + P)$
Augerino	$\mathcal{O}(N_B S L D + P)$	$\mathcal{O}(N_B S L D + P)$	$\mathcal{O}(N_B S D + P)$
Diff. Laplace	$\mathcal{O}(N_B S L D + P)$	$\mathcal{O}(N_B S L D^2 + P)$	$\mathcal{O}(N_B S L D^2 + P)$

Table 7: **Memory Complexities**. Differences are highlighted in red.

D.2 PRACTICAL SCALABILITY

A complexity analysis in big-$\mathcal{O}$ notation as provided by us in the previous sections allows to understand scalability in the limit, but constant terms that manifest in practice are still of interest. In this section we aim present real timing measurements for our method in comparison to Augerino and Differential Laplace, and elaborate on what overhead might be expected with respect to standard neural network training.

The empirical timings measurements on an NVIDIA RTX 3080-10GB GPU are shown in Table 8. We used a batch-size of 250, 200 for the MNIST and CIFAR10 experiments respectively, and 20 augmentation samples, just like in our main experiments in Table 1 and Figure 3. As can be seen, the overhead from using a partitioned network is fairly negligible compared to a standard forward and backward pass. The one difference compared to Augerino is, however, the fact that a separate forward-backward pass needs to be made to update the hyperparameters and regular parameters. This necessity is something that can be side-stepped with alternative partitioning schemes, as preliminarily mentioned in appendix C, and is an interesting direction for future research.

Method		MNIST CNN	CIFAR10 $^{\text{fix}}_{\text{up}}$ResNet-8	CIFAR10 $^{\text{fix}}_{\text{up}}$ResNet-14
Augerino		$\times 1$	$\times 1$	$\times 1$
Diff. Laplace†	Param.	$\times 1$	$\times 1$	$\times 1$
	Hyperparam.	$\times 2015.6$	$\times 18.2$	-
Partitioned	Param.	$\times 1.08$	$\times 1.17$	$\times 1.21$
	Hyperparam.	$\times 1.08$	$\times 1.08$	$\times 1.09$

Table 8: Relative empirical time increase with respect to a regular parameter update during standard training. † The timing multipliers with respect to the baseline for $^{\text{fix}}_{\text{up}}$ResNet-8 are taken from the timings reported in (Immer et al., 2022, Appendix D.4). On the ResNet-14, we get an out-of-memory error during the hyperparam. update step with Differentiable Laplace on the NVIDIA RTX 3080-10GB GPU when running with the official codebase (Immer and van der Ouderaa).

Memory Overhead Our proposed method's memory consumption scales in the same way as Augerino or vanilla neural network training. There is a minor constant memory overhead due to having to store the assignment of weights to partitions. In general, only $\log C$ bits per parameter are necessary to store the partition assignments, where C is the number of chunks. In our implementation, we only consider $C < 2^8$, and hence store the assignments in byte tensors. This means that the partitioned models require extra 25% memory for storing the parameters (when using 32bit floats to represent the parameters).

If the "default" weight values (i.e. those denoted $\hat{\boldsymbol{w}}_i$ in Figure 1) are non-zero, there is an additional overhead to storing those as well, which doubles the memory required to store the parameters. We observed there was no difference in performance when setting default weight values to 0 in architectures in which normalisation layers are used (i.e. most modern architectures). As such, we would in general recommend to set the default weight values to 0. However, we found setting default values to the initialised values to be necessary for stability of training deep normalisation-free architectures such as the $\frac{\text{fix}}{\text{up}}$ architectures (Zhang et al., 2019) we used to compare with Differentiable Laplace. As their method is not compatible with BatchNorm, we used these architectures in our experiments, and hence used non-zero default values.

Lastly, if the default weight values are set to the (random) initialisation values, it is possible to write a cleverer implementation in which only the random seeds are stored in memory, and the default values are re-generated every time they are need in a forward and a backward pass. This would make the memory overhead from storing the default values negligible.

E Note on Augerino

In replicating Augerino (Benton et al., 2020) within our code-base and experimenting with the implementation, we discovered a pathological behaviour that is partly mirrored by the authors of Immer et al. (2022). In particular, note that the loss function (Benton et al., 2020, Equation (5)) proposed by the authors is problematic in the sense that for any regularization strength $\lambda > 0$, the optimal loss value is negative infinity since the regularization term (negative L2-norm) is unbounded. In our experiments we observe that for a sufficiently-large value of λ and after a sufficient number of iterations, this behaviour indeed appears and training diverges. In practice, using Augerino therefore necessitates either careful tuning of λ, clipping the regularisation term (a method that introduces yet another hyperparameter), or other techniques such as early stopping.

In the open-source repository for the submission (Benton et al.), it can be seen that on many experiments the authors use a "safe" variant of the objective, in which they clip the regulariser (without pass-through of the gradient) once the l_∞-norm of any of the hyperparameters becomes larger than an arbitrary threshold. Without using this adjustment, we found that the Augerino experiments on MNIST crashed every time with hyperparameters diverging to infinity.

F Sensitivity to partitioning

F.1 Sensitivity in terms of final performance

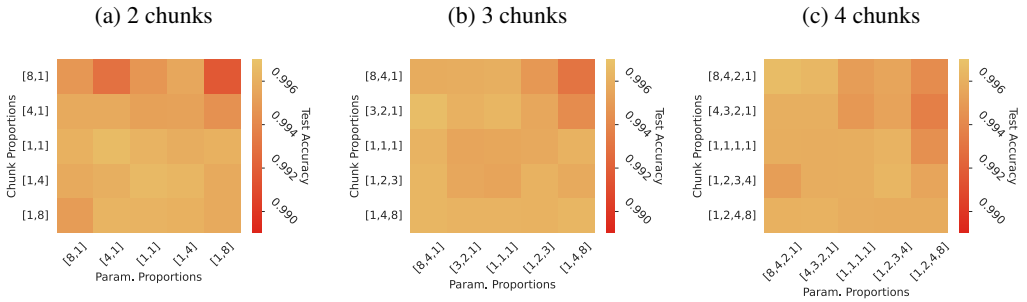


Figure 5: Learning affine augmentations on **MNIST** with a CNN fit on all data. x- and $y-$ ticks denote the ratios of parameters/datapoints assigned to each partition/chunk respectively.

Partitioned networks allow for learning hyperparameters in a single training run, however, they introduce an additional hyperparameter in doing so: the partitioning scheme. The practitioner needs to choose the number of chunks C, the relative proportions of data in each chunk, and the relative proportions of parameters assigned to each of the C partitions $\boldsymbol{w}_k$. We investigate the sensitivity to the partitioning scheme here. We show that our results are fairly robust to partitioning through a grid-search over parameter partitions and chunk proportions on the affine augmentation learning task on MNIST with the CNN architecture we use throughout this work.

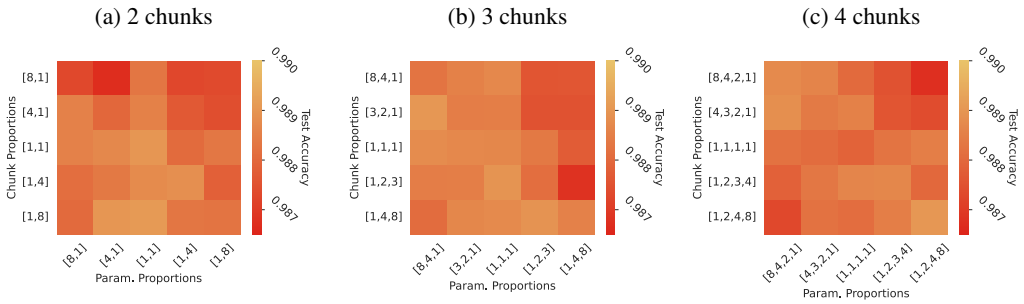


Figure 6: Learning affine augmentations on **RotMNIST** with a CNN fit on all data. x- and $y-$ ticks denote the ratios of parameters/datapoints assigned to each partition/chunk respectively.

Figure 5 and Figure 6 show the test accuracy for a choice of chunk and parameter proportions across two, three and four chunks. The proportions are to be read as un-normalized distributions; for example, chunk proportions set to $[1, 8]$ denotes that there are $8\times$ as many datapoints assigned to the second compared to the first. Each configuration was run with 2 random seeds, and we report the mean across those runs in the figure. The same architecture used was the same as for the main MNIST experiments in section 5 (see Appendix I.4 for details).

We observe that for various partition/dataset-chunking configurations, all models achieve fairly similar final test accuracy. There is a trend for models with a lot of parameters assigned to later chunks, but with few datapoints assigned to later chunks, to perform worse. While these results show a high level of robustness against the choice of additional hyperparameters introduced by our method, these results do show an opportunity or necessity for choosing the right partitioning scheme in order to achieve optimal performance.

F.2 Sensitivity in terms of hyperparameters found

To compare how the different partitioning schemes qualitatively impact the hyperparameters that the method identifies, we also retrain vanilla models from scratch using the hyperparameter values found using partitioned networks. Namely, we take the final value of the hyperparameters learned with partitioned networks with a given partitioning scheme, and plot the final test set accuracy of a vanilla neural network model trained from scratch with those hyperparameters. The results are shown in Figures 7 and 8.

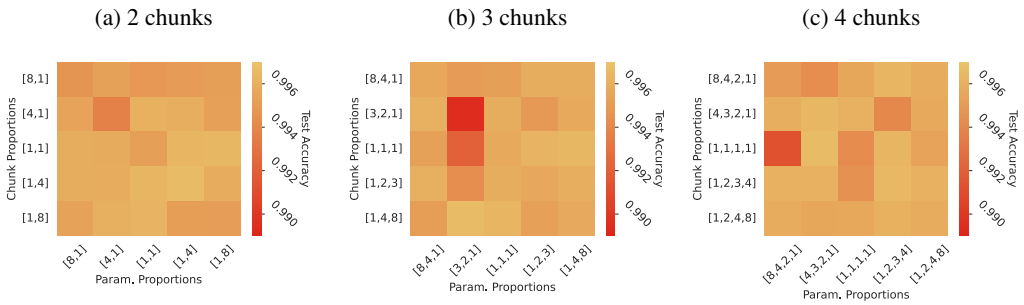


Figure 7: Standard neural network trained on **MNIST** with a CNN fit on all data, with hyperparameters found using partitioned networks with chunk and parameter proportions corresponding to those in Figure 5. x- and $y-$ ticks denote the ratios of parameters/datapoints assigned to each partition/chunk respectively.

G How good are the hyperparameters found?

Here we show that the hyperparameters found by partitioned networks are also a good set of hyperparameters for vanilla neural networks retrained from scratch. This section expands on the

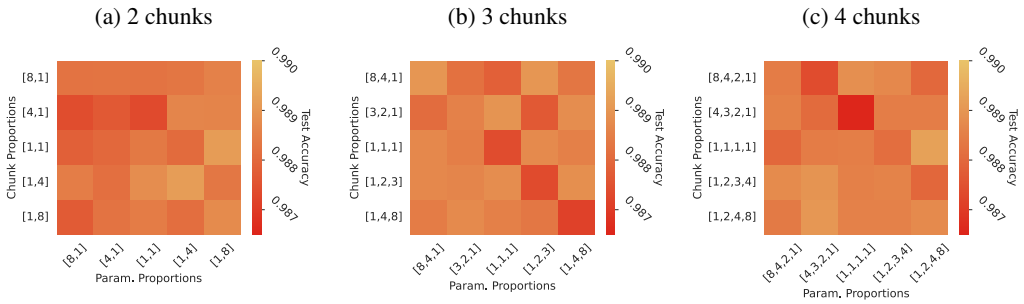


Figure 8: Standard neural network trained on **RotMNIST** with a CNN fit on all data, with hyperparameters found using partitioned networks with chunk and parameter proportions corresponding to those in Figure 6. x- and $y-$ ticks denote the ratios of parameters/datapoints assigned to each partition/chunk respectively.

experiment in section F.2. To validate this claim, we conducted a fairly extensive hyperparameter search on the affine augmentation learning task on RotMNIST; we trained 200 models by first sampling a set of affine augmentation parameters uniformly at random from a predefined range[10], and then training a neural network model (that averages across augmentation samples at train and test time, as described in Benton et al. (2020)) with standard neural training with those hyperparameters fixed throughout.

In Figure 9, we plot the final test-set performance of all the models trained with those hyperparameters sampled from a fixed range. Alongside, we show the hyperparameters and test-set performance of the partitioned networks as they progress throughout training. The partitioned networks consistently achieve final test-set performance as good as that of the best hyperparameter configurations identified through extensive random sampling of the space. We also show the test-set performance of neural network models, trained through standard training, with hyperparameters fixed to the final hyperparameter values identified by the partitioned networks. The hyperparameters identified by partitioned networks appear to also be good for regular neural networks; the standard neural networks with hyperparameters identified through partitioned training also outperform the extensive random sampling of the hyperparameter space. Furthermore, Figure 9 shows that partitioned networks do learn full rotation invariance on the RotMNIST task, i.e. when full rotation invariance is present in the data generating distribution.

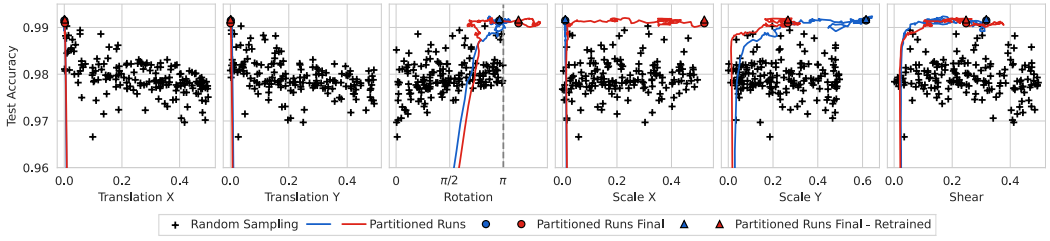


Figure 9: The test-set performance plotted alongside (1D projections of) affine augmentation hyperparameters on the RotMNIST task with MNIST-CNN. Final test-set accuracies are shown for the hyperparameters sampled randomly for a neural network model trained through standard training with those hyperparameters fixed (+). For multiple partitioned networks runs, the plot shows the progression of the identified hyperparameters and the test-set performance through the training run (—), as well as the final hyperparameters and test-set performance (●). Lastly, the plot also shows the final test-set accuracies of models trained through standard training on the final hyperparameters identified through partitioned training (▲).

[10]The ranges were: $\mathrm{Uniform}(0, \pi)$ for the maximum rotation, and $\mathrm{Uniform}(0, \frac{1}{2})$ for all the remaining affine augmentation parameters (maximum shear, maximum $x-$ and $y-$translation, and maximum $x-$ and $y-$ scale).

H LIMITATIONS

As mentioned in the main text, our method improves upon existing work, but also comes with its own limitations.

Complexity Inherent to our method — as presented in e.g. Figure 1 — is the necessity for an additional forward-backward pass to update the hyperparameters. Consequently, hyperparameter optimization has additional costs which, however, are significantly less than the computational costs of existing work, as we discuss in more detail in Appendix D.1 and the experimental section. Furthermore, empirically, partitioned networks usually require more training iterations to converge.

Performance Assuming the optimal hyper-parameters are given, training the full, non-partitioned networks based on those optimal values can be expected to yield better performance compared to the final model found by partitioned training. Partitioning the network inherently constrains the network capacity, causing some loss of performance. Opportunities for alleviating this performance loss while still enjoying single-run hyperparameter optimization through partitioned training will be left to future work. These include for example adjusting training rounds or increasing network capacity in the first place.

Partitioning While partitioned networks allows for automatic optimization of, intuitively, hard to tune hyperparameters, such as augmentation parameters, they come with the additional limitation of requiring to partition both the data and the model. This introduces an additional hyperparameter, namely, the partitioning strategy. While our default strategy of assigning more parameters and data to the first chunk works reasonably well on all of the experiments we consider, if one targets obtaining the best possible performance on a given task, the partitioning strategy might need additional tuning. We provide some empirical results about the sensitivity to partitioning in appendix F.1

I EXPERIMENTAL DETAILS

I.1 PARTITIONED TRAINING

Partitioned parameter update scheduling The gradient computation of Equation 3, as described in the main text, requires that the data-points for updating a given subnetwork $\boldsymbol{w}_s^{(k)}$ come from the appropriate dataset chunks $(\boldsymbol{x}, y) \in \mathcal{D}_{1:k}$ for a chunk k. Depending on the partitioning scheme (Appendix C), evaluating different subnetworks for different chunks can or cannot be done in a single mini-batch. More specifically, the **random weight-partitioning** we chose for our experiments requires a separate mini-batch per subnetwork (in order to keep the memory cost the same as for standard neural network training). An immediate question arising from a chunked dataset and several partitions is to define the order and frequency of updates across subnetworks. In our experiments we define (non-uniform) splits of the training dataset $\mathcal{D}$ across the C chunks, which requires a tailored approach to sampling the data. More specifically, for a given (normalized) ratio of chunk-sizes $[u_1, \ldots, u_C]$, each iteration of partitioned training proceeds as follows:

1. Sample a partition index $k \sim \text{Cat}(u_1, \ldots, u_C)$
2. Sample a mini-batch $\tilde{\mathcal{D}}$ of examples uniformly from $\mathcal{D}_{1:k}$.
3. Evaluate $\log p(\tilde{\mathcal{D}}|\boldsymbol{w}_s^{(k)}, \psi)$ using subnetwork $\boldsymbol{w}_s^{(k)}$ and
4. compute the (stochastic) gradient wrt. partition parameters $\boldsymbol{w}_k$ (Eq. 3).
5. Update partition parameters $\boldsymbol{w}_k$ using an optimizer, such as SGD or Adam.

This sampling scheme results in a data-point $(\boldsymbol{x}, y) \in \mathcal{D}_k$ from earlier chunks to be sampled more often. Concretely, the probability that an example in chunk k will be sampled is $\propto \sum_{i \leq k} u_i$. This is done so that each partition $\boldsymbol{w}_k$ is updated with equal probability on each of the examples in $\mathcal{D}_{1:k}$ As a result, we use *with replacement* sampling for the partitioned network training throughout the experimental section.

Gradient optimization of partitioned parameters A consequence of per-partition updates with the random weight partitioning scheme (appendix C) is that, for a chosen partition $\boldsymbol{w}_k$ to update, all other partitions do not receive a gradient update. In other words, the gradient at each iteration is sparse. Consequently, many off-the-shelve momentum-based optimizers will not account correctly. Specifically, we implement modifications to the PyTorch Paszke et al. (2019) provided optimizers that allow us to track per-partition momenta, number of steps, etc. Note that this creates a disconnect between the number of iterations across all partitions and the number of iterations per-partition. Doing so, however aligns the computational cost of training the partitioned network parameters with the cost of training regular neural network parameters. Regardless, we do not alter the way learning-rate schedulers behave in our experiments and anneal learning-rates according to the total number of iterations. Similarly, we report the total number of iterations when comparing against baselines that update all network-parameters per iteration.

While a simple gradient-accumulation scheme across mini-batches would result in a single gradient across all partitions, this approach inherently clashes with non-uniform partitioning $[u_1, \dots, u_C]$. Instead, we chose to sequentially apply gradients computed on a single partition, as described in the previous paragraphs. A further advantage of this approach is that learning progress made by updating partition $\boldsymbol{w}_k$ immediately influences (and can improve) the prediction of subnetworks $\boldsymbol{w}_s^{(k)}, \boldsymbol{w}_s^{(k+1)}, \dots, \boldsymbol{w}_s^{(C)}$.

Gradient optimization of hyperparameters Our partitioned network scheme makes it easy to compute stochastic gradients of the hyperparameter objective $\mathcal{L}_{\text{ML}}$ in Eq. 4 using batch gradient descent optimization methods. After every update to a randomly sampled network partition (see previous paragraph), we update hyperparamters ψ as follows:

- sample a dataset chunk index $k \sim \text{Cat}(\frac{u_2}{Z}, \dots, \frac{u_C}{Z})$. Ratios are re-normalized to exclude $\mathcal{D}_1$.
- sample a mini-batch $\tilde{\mathcal{D}}$ of examples uniformly from $\mathcal{D}_k$ (Note the choice of $\mathcal{D}_k$ instead of $\mathcal{D}_{1:k}$).
- Evaluate $\log p(\tilde{\mathcal{D}}|\boldsymbol{w}_s^{(k-1)}, \psi)$ using subnetwork $\boldsymbol{w}_s^{(k-1)}$ and
- compute the (stochastic) gradient wrt. hyperparameters ψ (Eq. 4).
- Update partition parameters ψ using an optimizer, such as SGD or Adam.

The above sampling procedure yields an unbiased estimate of gradients in eq. 4.

The fact that we optimize hyperparameters with gradients based on data from a single chunk at a time is again a consequence of the random weight-partitioning scheme for the partitioned networks. It is possible to compute gradients wrt. ψ for mini-batches with examples from multiple chunks at a time. With the random weight partitioning scheme, this would result in an increased memory overhead. Lastly, we could also accumulate gradients from different chunks, similarly to Immer et al. (2022), and this would likely result in a lower-variance estimate per update .

It is also possible to reduce the computational overhead of evaluating two mini-batches per iteration (one for updates to $\boldsymbol{w}_k$, one for ψ) as we do in our experiments by interleaving hyperparameter updates at less frequent intervals. We leave an exploration of these design choices to future work. Throughout all experiments, except those in the federated settings (see section J), we use the same batch-size for the hyperparameter udpates as for the regular parameter updates.

Weight-decay For partitioned networks, whenever using weight-decay, we scale the weight decay for earlier partitions with the reciprocal of the number of examples in chunks used to optimize them, following the diagonal Gaussian prior interpretation of weight-decay. This makes the training compatible with the variational interpretation in Appendix B.

I.2 Partitioned affine transformations

In Appendix C we described how we realize partitioned versions of fully-connected and convolutional layers. Design choices for other parameterized network layers used in our experiments are described below.

Normalization layers It is common-place in most architectures to follow a normalization layer (such as BatchNorm (Ioffe and Szegedy, 2015), GroupNorm (Wu and He, 2018)) with an element-wise or channel-wise, affine transformation. Namely, such a transformation multiplies its input $\boldsymbol{h}$ by a scale vector$\boldsymbol{s}$ and adds a bias vector $\boldsymbol{b}$: $\boldsymbol{o} = \boldsymbol{h} * \boldsymbol{s} + \boldsymbol{b}$. For random weight-partitioned networks, we parameterize such affine transformations by defining separate vectors $\{\boldsymbol{s}_1, \ldots, \boldsymbol{s}_C\}$ and $\{\boldsymbol{b}_1, \ldots, \boldsymbol{b}_C\}$ for each partition; the actual scale and bias used in a given subnetwork $\boldsymbol{w}_s^{(k)}$ are $\boldsymbol{s}_s^{(k)} = \prod_{i \in \{1,\ldots,k\}} s_i$ and $\boldsymbol{b}_s^{(k)} = \sum_{i \in \{1,\ldots,k\}} b_i$ respectively. This ensures that the final affine transformation for each subnetwork $\boldsymbol{w}_s^{(k)}$ depends on the parameters in the previous partitions $[1, \ldots, k-1]$. Doing so increases the parameter count for the partitioned networks in architectures that use those normalization layers by a negligible amount.

Scale and bias in FixUp networks The FixUp paper (Zhang et al., 2019) introduces extra scales and biases into the ResNet architecture that transform the entire output of the layers they follow. We turn these into "partitioned" parameters using the same scheme as that for scales and biases of affine transformations following normalization layers.

For partitioned networks, through-out the paper, we match the proportion of parameters assigned to each partition k in each layer to the proportion of data examples in the corresponding chunk $\mathcal{D}_k$.

I.3 Architecture choices

Input selection experiments We use a fully-connected feed-forward neural network with 2 hidden layers of size $[256, 256]$, and with GeLU (Hendrycks and Gimpel, 2016) activation functions. We initialise the weights using the Kaiming uniform scheme (He et al., 2015). For partitioned networks, we use the random-weight partitioning scheme.

Fixup Resnet For all experiments using FixUp ResNets we follow Immer et al. (2022); Zhang et al. (2019), and use a 3-stage ResNet with channel-sizes $(16, 32, 64)$ per stage, with identity skip-connections for the residual blocks as described in He et al. (2016). The residual stages are followed by average pooling and a final linear layer with biases. We use 2D average pooling in the residual branches of the downsampling blocks.We initialize all the parameters as described in Zhang et al. (2019).

Wide ResNet For all experiments using a Wide-ResNet-N-D (Zagoruyko and Komodakis, 2016), with N being the depth and D the width multiplier, we use a 3 stage ResNet with channel-sizes $(16D, 32D, 64D)$. We use identity skip-connections for the residual blocks, as described in He et al. (2016), also sometimes known as ResNetV2.

ResNet-50 We use the "V2" version of Wide ResNet as described in (Zagoruyko and Komodakis, 2016) and replace BatchNormalization with GroupNormalization using 2 groups. We use the 'standard' with with $D = 1$ and three stages of 8 layers for a 50-layer deep ResNet.

We use ReLU activations for all ResNet experiments throughout.

MNIST CNN For the MNIST experiments, we use the same architecture as Schwöbel et al. (2021) illustrated in the replicated Table 9.

Table 9: CNN architecture for MNIST experiments

Layer	Specification
2D convolution	channels=20, kernel size=$(5, 5)$, padding=2, activation=ReLU
Max pooling	pool size=$(2, 2)$, stride=2
2D convolution	channels=50, kernel size=(5,5), padding=2, activation=ReLU
Max pooling	pool size=$(2, 2)$, stride=2
Fully connected	units=500, activation=ReLU
Fully connected	units=50, activation=ReLU
Fully connected	units=10, activation=Softmax

I.4 TRAINING DETAILS

Learning affine augmentations For the parametrization of the learnable affine augmentation strategies, we follow prior works for a fair comparison. More specifically, for our MNIST based setup we follow the parametrization proposed in Schwöbel et al. (2021) whereas for our CIFAR10 based setup we use the generator parametrization from Immer et al. (2022).

Input selection experiments For the model selection (non-differentiable) input selection experiments, we train all variants with Adam with a learning rate of 0.001 and a batch-size of 256 for 10000 iterations. For both Laplace and partitioned networks, we do early stopping based on the marginal likelihood objective ($\mathcal{L}_{\text{ML}}$ for partitioned networks). We use weight-decay 0.0003 in both cases. For the post-hoc Laplace method, we use the diagonal Hessian approximation, following the recommendation in (Immer et al., 2021). For partitioned networks, we divide the data and parameters into 8 chunks of uniform sizes. We plot results averaged across 3 runs.

Mask learning for input selection experiment We use the same optimizer settings as for the input selection experiment. We train for 30000 iterations, and optimize hyperparameters with Adam with a learning rate of 0.001. We divide the data and parameters into 4 uniform chunks.

MNIST experiments We follow Schwöbel et al. (2021), and optimize all methods with Adam with a learning rate of 0.001, no weight decay, and a batch-size of 200. For the partitioned networks and Augerino results, we use 20 augmentation samples. We use an Adam optimizer for the hyperparameters with a learning rate of 0.001 (and default beta parameters).

For Augerino on MNIST, we use the "safe" variant, as otherwise the hyperparameters and the loss diverge on every training run. We elaborate on this phenomenon in Appendix E. Otherwise, we follow the recommended settings from (Benton et al., 2020) and Immer et al. (2022), namely, a regularization strength of 0.01, and a learning rate for the hyperparameters of 0.05.

For both MNIST and CIFAR experiments, we found it beneficial to allocate more data to either the earlier, or the later, chunks. Hence, we use 3 chunks with $[80\%, 10\%, 10\%]$ split of examples for all MNIST and CIFAR experiments.

CIFAR variations experiments We again follow Immer et al. (2022), and optimize all ResNet models with SGD with a learning rate of 0.1 decayed by a factor of $100\times$ using Cosine Annealing, and momentum of 0.9 (as is standard for ResNet models). We use a batch-size of 250. We again use Adam for hyperparameter optimization with a learning rate of 0.001 (and default beta parameters). We train our method for $[2400, 8000, 12000, 20000, 40000]$ iterations on subsets $[1000, 5000, 10000, 20000, 50000]$ respectively for CIFAR-10, just as in (Immer et al., 2022). For all methods, we used a weight-decay of $1e-4$. For partitioned networks, we increase the weight decay for earlier partitions with the square root of the number of examples in chunks used to optimize them, following the diagonal Gaussian prior interpretation of weight-decay. We use 3 chunks with $[80\%, 10\%, 10\%]$ split of examples.

For RotCIFAR-10 results, we noticed our method hasn't fully converged (based on training loss) in this number of iterations, and so we doubled the number of training iterations for the RotMNIST results. This slower convergence can be explained by the fact that, with our method, we only update a fraction of the network parameters at every iteration.

TinyImagenet experiments Our experiments with TinyImagenet (Le and Yang, 2015) closely follow the setting for the CIFAR-10 experiments described above. Images are of size $64x64$ pixels, to be classified into one of 200 classes. The training-set consists of 100000 images and we compare our method against baselines on subset of $[10000, 50000, 100000]$ datapoints. For the standard version of TinyImagenet, we train for $[80000, 80000, 40000]$ steps respectively and for the rotated version of TinyImagenet we train for 120000 steps for all subset sizes. We tuned no other hyper-parameters compared to the CIFAR-10 setup and report our method's result for a partitioning with $[80\%, 20\%]$ across 2 chunks after finding it to perform slightly better than a $[80\%, 10\%, 10\%]$ split across 3 chunks in a preliminary comparison.

Fine-tuning experiments For the fine-tuning experiments in table 2, we trained a FixUp ResNet-14 on a subset of 20000 CIFAR10 examples, while optimizing affine augmentations (following affine augmentations parameterization in (Benton et al., 2020)). We used the same optimizer settings as for all other CIFAR experiments, and trained for 80000 iterations, decaying the learning rate with Cosine Annealing for the first 60000 iterations. For fine-tuning of validation-set optimization models, we used SGD with same settings, overriding only the learning rate to 0.01. We tried a learning rate of 0.01 and 0.001, and selected the one that was most favourable for the baseline based on the test accuracy.

We also tried training on the full CIFAR-10 dataset, but found that all methods ended up within a standard error of each other when more than 70% of the data was assigned to the first chunk (or training set, in the case of validation set optimization). This indicates that CIFAR-10 is sufficiently larger that, when combined with affine augmentation learning and the relatively small ResNet-14 architecture used, using the extra data in the 2nd partition (or the validation set) results in negligible gains.

I.5 Datasets

Input selection synthetic dataset For the input selection dataset, we sample 3000 datapoints for the training set as described in section 5, and we use a fresh sample of 1000 datapoints for the test set.

RotMNIST Sometimes in the literature, RotMNIST referes to a specific subset of 12000 MNIST examples, whereas in other works, the full dataset with 60000 examples is used. In this work, following (Benton et al., 2020; Immer et al., 2022) we use the latter.

J Federated partitioned training

In this section, we explain how partitioned networks can be applied to the federated setting, as well as the experimental details.

J.1 Partitioned networks in FL

In order to apply partitioned networks to the federated setting, we randomly choose a partition for each client such that the marginal distribution of partitions follows a pre-determined ratio. A given chunk $\mathcal{D}_k$ therefore corresponds to the union of several clients' datasets. Analogous to how "partitioned training" is discussed in the main text and Appendix I, we desire each partition $\boldsymbol{w}_k$ to be updated on chunks $\mathcal{D}_{1:k}$. Equation 3 in the main text explains which data chunks are used to compute gradients wrt. parameter partition $\boldsymbol{w}_k$. An analogous perspective to this objective is visualized by the exemplary algorithm in Figure 1 and asks which partitions are influenced (*i,e.*, updated) by data from chunk $\mathcal{D}_k$: A data chunk $\mathcal{D}_k$ is used to compute gradients wrt. partitions $\boldsymbol{w}_{k:C}$ through subnetworks $\boldsymbol{w}_s^{(k)}$ to $\boldsymbol{w}_s^{(C)}$ respectively. Consequently, a client whose dataset is assigned to chunk $\mathcal{D}_k$ can compute gradients for all partitions $\boldsymbol{w}_{k:C}$.

Updating network partitions Due to the weight-partitioned construction of the partitioned neural networks, it is not possible to compute gradients with respect to all partitions in a single batched forward-pass through the network. Additionally, a change to the partition parameters $\boldsymbol{w}_k$ directly influences subnetworks $\boldsymbol{w}_s^{(k+1)}$ to $\boldsymbol{w}_s^{(C)}$. In order to avoid the choice of ordering indices k to C for the client's local update computation, we update each partition independently while keeping all other partitions initialised to the server-provided values that the client received in that round t: Denote $D_{i,k}$ as the dataset of client i where we keep index k to emphasize the client's assignment to chunk k. Further denote $\boldsymbol{w}_{j,i}^{t+1}$ as the partition $\boldsymbol{w}_j^t$ after having been updated by client i on dataset $D_{i,k}$.

$$\boldsymbol{w}_{j,i}^{t+1} = \arg\max_{\boldsymbol{w}_j} \log p\left(D_{i,k} | (\boldsymbol{w}_1^t, \ldots, \boldsymbol{w}_j^t, \hat{\boldsymbol{w}}_{j+1}^t, \ldots, \hat{\boldsymbol{w}}_{j+C}^t), \psi\right) \quad \forall j \in [k, C], \tag{25}$$

where the details of optimization are explained in the following section. We leave an exploration for different sequential updating schemes to future work. The final update communicated by a client to the server consists of the concatenation of all updated parameter partitions

$\boldsymbol{w}_{\cdot,i}^{t+1} = \text{concat}(\boldsymbol{w}_{k,i}^{t+1}, \ldots, \boldsymbol{w}_{C,i}^{t+1})$. Note that partitions $(\boldsymbol{w}_1^t, \ldots, \boldsymbol{w}_{k-1}^t)$ have not been modified and need not be communicated to the server. The resulting communication reductions make partitioned networks especially attractive to FL as data upload from client to server poses a significant bottleneck. In practice, we expect the benefits of these communication reductions to outweigh the additional computation burden of sequentially computing gradients wrt., to multiple partitions.

The server receives $\boldsymbol{w}_{\cdot,i}^{t+1}$ from all clients that participates in round t, computes the delta's with the global model and proceeds to average them to compute the server-side gradient in the typical federated learning fashion (Reddi et al., 2020).

Updating hyperparameters The computation of gradients on a client i wrt. ψ is a straight-forward extension of equation 4 and the exemplary algorithm of Figure 1:

$$\nabla_\psi \mathcal{L}_{\text{ML}}\left(D_{i,k}, \psi\right) \approx \nabla_\psi \log p\left(D_{i,k} | \boldsymbol{w}_{s,i}^{(t+1),(k-1)}, \psi\right), \tag{26}$$

where $D_{i,k}$ corresponds to client i's local dataset which is assigned to chunk k and $\boldsymbol{w}_s^{(t+1),(k-1)}$ corresponds to the $(k-1)$'th subnetwork after incorporating all updated partitions $\boldsymbol{w}_{s,i}^{(t+1),(k-1)} = \text{concat}(\boldsymbol{w}_1^t, \ldots, \boldsymbol{w}_{k-1}^t, \boldsymbol{w}_{k,i}^{t+1}, \ldots, \boldsymbol{w}_{C,i}^{t+1})$. Note that we compute a full-batch update to ψ in MNIST experiments and use a batch-size equal to the batch-size for the partitioned parameter updates for CIFAR10.

Upon receiving these gradients from all clients in this round, the server averages them to form a server-side gradient. Conceptually, this approach to updating ψ corresponds to federated SGD.

J.2 Federated setup

Non-i.i.d. partitioning For our federated experiments, we split the $50k$ MNIST and $45k$ CIFAR10 training data-points across 100 clients in a non-i.i.d. way to create the typical challenge to federated learning experiments. In order to simulate *label-skew*, we follow the recipe proposed in Reddi et al. (2020) with $\alpha = 1.0$ for CIFAR10 and $\alpha = 0.1$ for MNIST. Note that with $\alpha = 0.1$, most clients have data corresponding to only a single digit. For our experiments on rotated versions of CIFAR10 and MNIST, we sample a degree of rotation per data-point and keep it fixed during training. In order to create a non-i.i.d partitioning across the clients, we bin data-points according to their degree of rotation into 10 bins and sample using the same technique as for label-skew with $\alpha = 0.1$ for both datasets. Learning curves are computed using the $10k$ MNIST and $5k$ CIFAR10 validation data-points respectively. For the rotated dataset experiments, we rotate the validation set in the same manner as the training set.

Architectures and experimental setup We use the convolutional network provided at Schwöbel et al. (2021) for MNIST and the ResNet-9 (Dys) model for CIFAR10 but with group normalization (Wu and He, 2018) instead of batch normalization. We include (learnable) dropout using the continuous relaxation proposed at Maddison et al. (2016) between layers for both architectures. We select 3 chunks for MNIST with a $[0.7, 0.2, 0.1]$ ratio for both, client-assignments and parameter-partition sizes. For CIFAR10, we found a $[0.9, 0.1]$ split across 2 sub-networks to be beneficial. In addition to dropout logits, ψ encompasses parameters for affine transformations, *i.e.*, shear, translation, scale and rotation. We report results after $2k$ and $5k$ rounds, respectively, and the expected communication costs as a percentage of the non-partitioned baseline.

Shared setting In order to elaborate on the details to reproduce our results, we first focus on the settings that apply across all federated experiments. We randomly sample the corresponding subset of $1.25k$, $5k$ data-points from the full training set and keep that selection fixed across experiments (*i,e.*, baselines and partitioned networks) as well as seeds. The subsequent partitioning across clients as detailed in the previous paragraph is equally kept fixed across experiments and seeds. Each client computes updates for one epoch of its local dataset, which, for the low data regimes of $1.25k$ data-points globally, results in single update per client using the entire local dataset. We averaged over 10 augmentation samples for the forward pass in both training and inference.

MNIST & RotMNIST For $5k$ data-points and correspondingly 50 data-points on average per client, most clients perform a single update step. A small selection of clients with more than 64 data-points

performs two updates per round. For the experiments using the full dataset and a mini-batch size of 64, each client performs multiple updates per round. After initial exploration on the baseline FedAvg task, we select a local learning-rate of $5e-2$ and apply standard SGD. The server performs Adam Reddi et al. (2020) with a learning rate of $1e-3$ for the model parameters. We keep the other parameters of Adam at their standard PyTorch values. We find this setting to generalize to the partitioned network experiments but found a higher learning rate of $3e-3$ for the hyper-parameters to be helpful. We chose the convolutional network from Schwöbel et al. (2021) with (learned) dropout added between layers. The model's dropout layers are initialized to drop 10% of hidden activations. For the baseline model we keep the dropout-rate fixed and found 10% to be more stable than 30%.

CIFAR10 & RotCIFAR10 We fix a mini-batch size of 32, leading to multiple updates per client per round in both, the full dataset regime as well as the $5k$ data-points setting. Similarly to the MNIST setting, we performed an initial exploration of hyperparameters on the baseline FedAvg task and use the same ones on partitioned networks. We used dropout on the middle layer of each block which was initialized to 0.1 for both the baseline and partitioned networks and whereas partitioned networks optimized it with $\mathcal{L}_{ML}$ and the concrete relaxation from Maddison et al. (2016), the baseline kept it fixed. For the server side optimizer we used Adam with the default betas and a learning rate of $1e-2$, whereas for the hyperparameters we used Adam with the default betas and a learning rate of $1e-3$. In both cases we used an $\epsilon = 1e-7$. For the local optimizer we used SGD with a learning rate of $10^{-0.5}$ and no momentum.

J.3 MNIST LEARNING CURVES

In Figure 10 we show learning curves for the three considered dataset sizes on the standard MNIST task. Each learning curve is created by computing a moving average across 10 evaluations, each of which is performed every 10 communication rounds, for each seed. We then compute the average and standard-error across sees and plot those values on the y-axis. On the x-axis we denote the total communication costs (up- and download) to showcase the partitioned networks reduction in communication overhead. We see that especially for the low dataset regime, training has not converged yet and we expect performance to improve for an increased number of iterations.

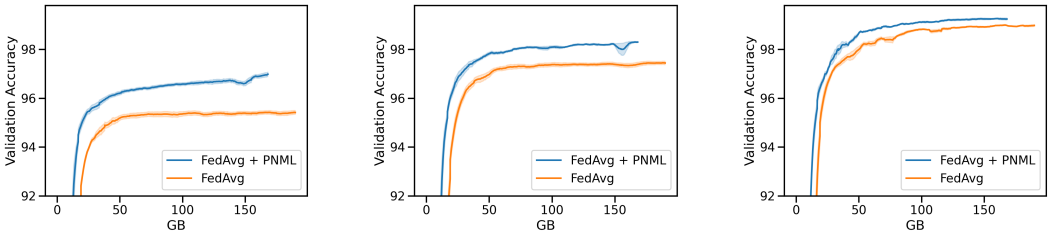


Figure 10: Learning curves for MNIST experiments on $1.25k$, $5k$ and $50k$ data-points respectively.