Appendix

A. Experimental details

The code for this work was directly adapted from the official MixMo [11] codebase: https://github.com/alexrame/mixmo-pytorch.

We followed similar experimental settings on CIFAR 100 as MixMo [11] and present here the adapted setting description:

We used standard architecture WRN-28-w, with a focus on $w = 2$. We re-use the **hyper-parameters** configuration from MIMO [2] with batch repetition 2 (bar2). The optimizer is SGD with learning rate of $\frac{0.1}{b} \times \frac{\text{batch-size}}{128}$, batch size 64, linear warmup over 1 epoch, decay rate 0.1 at steps $\{75, 150, 225\}$, l_2 regularization 3e-4. We follow standard MSDA practices [14] and set the maximum number of epochs to 300. Our experiments ran on a single NVIDIA 12Go-TITAN X Pascal GPU.

All experiments were run three times on three fixed seeds from the same version of the codebase. Qualitative results presented in Fig. 3 and Fig. 5 are obtained by visualizing results for the first set of random seeds. Quantitative results presented in Tab. 1 are given in the form of $mean \pm std$ over the three runs.

B. Complementary adjustments to MIMO procedures in MixShare

MIMO methods use a number of auxiliary procedures to train strong subnetworks. However, as MixShare differs significantly from standard MIMO frameworks, it does not use these frameworks to the same extent.

CutMix probability in the input block MixMo [11] only uses cutmix mixing in its input block about half the time, using a basic summing operation on the two encoded inputs the rest of the time. This is because the model will use a summing operation at test time. Therefore, the use of cutmix at training induces an strong train/test gap that needs to be bridged by the use of summing during training.

We cannot afford to use summing half the time as unmixing relies on the use of cutmix in the input block. However, since our two encoders are very similar (due to our kernel alignment), cutmix and summing (or averaging) behave very similarly and the train/test gap is therefore minimal.

Input Repetition A slight train/test gap still remains however since the model is rarely presented the same image as input to both subnetworks at training time. We solve this by reprising a procedure introduced in the seminal MIMO paper [2]: input repetition. In our case, we ensure 10% of inputs of our batches are made of repetition of the same image during training.

Loss rebalancing MixMo [11] introduced a re-weighting function of the subnetwork training losses that rescales the mixing ratios used in the inputs block. These ratios are rescaled to be less lopsided (closer to an even 50/50 split) before being applied to their relevant subnetwork losses. This rescaling is necessary as it ensures all parameters receive sufficient training signal.

We however find in our experiments it is more beneficial to do away with this re-balancing and keep the original mixing ratios, which we explain by the large amount of features shared between subnetworks. Since features are shared, we do not need to worry about some features receiving too little training signal.

C. A more nuanced discussion on kernel alignment

While MixShare uses the exact same initialization of the encoder kernels for simplicity, it is interesting to note much weaker versions of kernel alignment are sufficient to obtain similar results.

Indeed, we found in our experiments that initializing the kernels to be simply co-linear is more than enough to ensure proper feature sharing. In fact, this leads to the exact same performance as using the same initialization and the encoder kernels quickly converge to similar values.

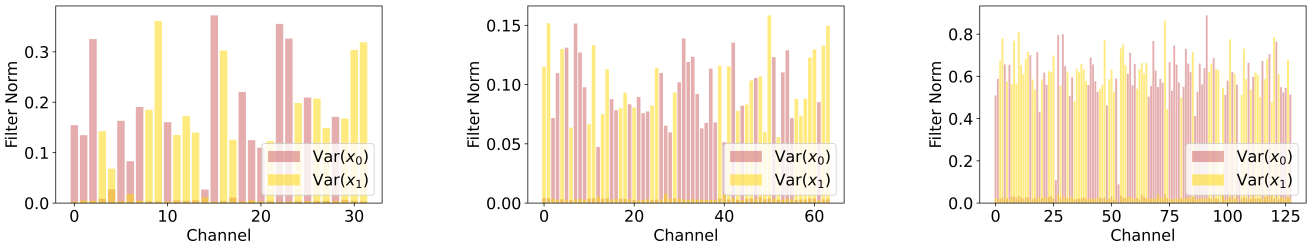
This further validates our intuition that MIMO models need a "common language" to benefit from sharing features: all that is required is for encoder kernels to extract the same "type" of features.

D. Analysis of subnetwork features within the core network

Sec. 2 studies what features each subnetwork uses in the input block and output block of the multi-input multi-output model. Studying the importance of features within the core networks for each subnetworks is more difficult as it is not possible to consider the model weights. Reprising an analysis conducted in the Appendix of [11], we identify the influence of intermediate features on subnetworks with the variance of the feature with respect to the relevant input.

For the first subnetwork, if we consider the intermediate feature map (at one point in the network f) $\mathcal{M}_{int} = f_{int}(\mathcal{D}_{test}, d)$, such that $\mathcal{M}_{int}$ is of shape $N \times C \times H \times W$ with $\mathcal{D}_{test}$ the test set, d a fixed input, N the size of the test set, C the number of intermediate feature maps and $H \times W$ the spatial coordinates. We compute the importance of each of the C feature map with respect to the first subnetwork as $Mean(Var(\mathcal{M}_{int}, dim = 0), dim = (1, 2))$. The importance of intermediate features for the second subnetwork is obtained similarly by considering $\mathcal{M}_{int} = f_{int}(d, \mathcal{D}_{test})$.

Fig. 6 shows the resulting feature importance maps at after each of the three residual blocks in the core network.



(a) Feature variance after block 1.

(b) Feature variance after block 2.

(c) Feature variance after block 3.

Figure 6. Checking the variance of feature maps w.r.t. the two inputs at different levels of the network shows clear separation of features in standard multi-input multi-output architectures.

As can be observed, the subnetworks remain consistently separated in the core network.