Are Neural Nets Modular? Inspecting Their Functionality Through Differentiable Weight Masks

Róbert Csordás [1] **Sjoerd van Steenkiste** [1] **Jürgen Schmidhuber** [1 2]

Abstract

Neural networks (NNs) whose subnetworks implement reusable functions are expected to offer numerous advantages, e.g., compositionality through efficient recombination of functional building blocks, interpretability, preventing catastrophic interference by separation, etc. Understanding if and how NNs are modular could provide insights into how to improve them. Current inspection methods, however, fail to link modules to their function. We present a novel method based on learning binary weight masks to identify individual weights and subnets responsible for specific functions. This powerful tool shows that typical NNs fail to reuse submodules, becoming redundant instead. It also yields new insights into known generalization issues with the SCAN dataset. Our method also unveils class-specific weights of CNN classifiers, and shows to which extent classifications depend on them. Our findings open many new important directions for future research.

1. Introduction

Modularity is an important organization principle in both artificial (Baldwin & Clark, 2000; Schmidhuber, 1990) and biological (Clune et al., 2013; von Dassow & Munro, 1999; Lorenz et al., 2011) systems. It provides a natural way of achieving compositionality, which seems essential for systematic generalization, one of the areas where typical artificial neural networks (NNs) do not yet perform well (Fodor et al., 1988; Marcus, 1998; Lake & Baroni, 2018; Hupkes et al., 2019).

Recently, NNs with explicitly designed modules have demonstrated superior generalization cabilities (Chang et al., 2019; Bahdanau et al., 2019; Clune et al., 2013; Kirsch et al., 2018; Andreas et al., 2016). An implicit assumption behind such models is that NNs without hand-designed modularity do not become modular by themselves. In contrast, it was recently found how certain types of modular structures emerge in standard NNs (Watanabe, 2019; Filan et al., 2020). However, these methods use proxies such as activation statistics or connectivity to identify modules, making it unclear whether they correspond to functional blocks. For example functionally consistent modules might have units with different activation statistics and vice versa, making such analysis inappropriate for inspecting functional modularity. Establishing a better connection between modules and their function remains essential for obtaining a powerful analysis tool that helps to understand and alleviate the shortcomings of current NNs.

Here we analyze functional modularity in common neural architectures: recurrent NNs (RNNs) and feedforward NNs (FNNs) including convolutional NNs (CNNs). We define modules as a subnetworks needed to perform a specific function (functional modules). We train an NN on its original task (e.g., classify images into 10 classes), then freeze its weights. Now the original task is modified in line with a specific functionality of interest (e.g., leave one class out and solve the remaining 9). On this modified task, we train probabilistic, binary, but differentiable masks for all weights (while the NN's weights remain frozen). The result is a binary mask for each modified task, exhibiting the subnetwork necessary to perform it.

To investigate whether the discovered functional modules support compositionality, we analyze whether the NN has two desirable properties: (**P1**) *it uses different modules for very different functions*, and (**P2**) *it uses the same module for identical functions* that may have to be performed multiple times. **We experimentally show that many typical NNs exhibit P1 but not P2.** By analyzing transfer learning on permuted MNIST, we provide additional insights into this issue. We offer a possible explanation: simple data routing between modules in standard NNs often would be highly desirable, but it is hard to learn by the NN's weight matrix which typically must also learn to satisfy the conflicting goal of transforming the data in useful ways. Standard NNs have no bias towards separating the conceptually different

[1]The Swiss AI Lab, IDSIA / USI / SUPSI [2]NNAISENSE. Correspondence to: Róbert Csordás <robert@idsia.ch>.

2020 ICML Workshop on Human Interpretability in Machine Learning (WHI 2020).

goals of data transformation and routing.

We show that the modules discovered by typical NNs do not tend to encourage compositional solutions. For example, we analyze encoder-decoder LSTMs on the SCAN dataset (Lake & Baroni, 2018) designed to test systematic generalization based on textual commands. We show that new weights are required to deal with new command combinations, even those combinations governed by already known rules. The new weights are responsible solely for these new combinations, indicating that the learned modules are non-compositional. Instead they are very pattern recognition-like and fail to perform the more symbolic manipulation required for generalization on SCAN.

Finally, we study whether functional modules are present in CNNs, which are thought to rely heavily on shared features. Surprisingly, we can identify subsets of weights solely responsible for single classes. When removing such a subset, the performance on its class drops significantly. By analyzing the resulting confusion matrices, we identify classes relying on similar features.

2. Discovering Modular Structure via Weight-Level Introspection

Current methods for discovering modular structure in NNs are based on clustering individual *units* based on their similarity (Watanabe, 2019; Filan et al., 2020). Such unit-level analysis might be misleading though. Units can be shared even when their weights, which perform the actual computation, are not. Indeed, units can be viewed as mere "wires" for transmitting information. Imagine a task where operations on numbers should be performed, conditioned on the input. The first hidden layer should represent the number after the first operation, regardless of which one it was. Hence a weight level analysis is critical.

The method proposed in this paper inspects pre-trained NNs by applying masks to their frozen weights. Masks are trained on modified tasks depending on the specific functionalities we want to analyze, e.g., sub-tasks of the original problem, or tasks based on different data splits, to test generalization. The trained masks identify subnetworks responsible for the corresponding functionalities.

First the NN is trained on the original, unmodified task. Once converged, its weights are frozen. Next we train the mask on the modified task. For each weight, the mask defines a probability of keeping it. All N weights are handled independently of each other. We use $i \in [1, N]$ to denote the weight index. The mask's probabilities are represented as logits $l_i \in \mathbb{R}$, which are initialized to keep the weights with high probability (90%). To restrict arbitrary scaling of the weights, we binarize masks before applying them. For binarization, we use the Gumbel-Sigmoid with straight-through estimator, which we derive from Gumbel-Softmax (Jang et al., 2017; Maddison et al., 2017) in Appendix A. A sample $s(l_i) \in [0, 1]$ from the mask can be drawn as:

$$s(l_i) = \sigma\left(\frac{1}{\tau}\left(l_i - \log\frac{\log U_1}{\log U_2}\right)\right), \tag{1}$$

where U_1, U_2 are i.i.d. samples from the uniform distribution $\mathrm{U}(0, 1)$, $\tau \in (0, \infty)$ is the temperature and $\sigma(x) = \frac{1}{1+e^{-x}}$ is the sigmoid function. Next, we can use a straight-through estimator to obtain a binarized sample:

$$m_i = [\mathbb{1}_{s(l_i)>0.5} - s(l_i)] + s(l_i), \tag{2}$$

where $\mathbb{1}_x$ is the indicator function and $[x]$ is an operator for preventing gradient flow. The masks are applied element-wise:

$$w_i' = w_i * m_i. \tag{3}$$

The goal of the masking is to remove weights that are not required to implement the target function. Thus, logits l_i should be regularized, such that the probability for weight w_i to be active is small unless w_i is necessary for the task. This is achieved by adding the term $r = \alpha \sum_i l_i$ to the loss. $\alpha \in [0, \infty)$ is a hyper-parameter responsible for the strength of the regularization.

At the end of the training process, deterministic binary masks $M_i \in \{0, 1\}$ are created by thresholding masks at 50% probability: $M_i = \mathbb{1}_{\sigma(l_i)>0.5}$. The mask M shows the subnetwork required for the task. Because the probability mass is usually concentrated at 0 and 1, thresholding is safe (See Fig. 6 in Appendix).

We evaluate the resulting subnetwork by applying the mask, whose interpretation is task-dependent. For example, to show that the resulting subset of weights *is* responsible for a particular subtask (A) but not for another (B), apply the mask trained on A and test on both. A performance drop is expected on task B only. Otherwise, if a module *is exclusively* needed only for a particular subtask but not for the other, we can invert the masks and test on both tasks. The inverted masks are expected to perform well on the complementary task, but not on the original one. However, this mask inversion method is limited to analyzing entirely disjoint weights. Weight sharing between tasks can be analyzed through the overlap of their corresponding weight masks (see Appendix B.1).

The precise experimental details of all our experiments are left to the Appendix. Mean and standard deviations shown in the figures are calculated over 10 runs.

Conditions for the validity of the analysis. Choosing the regularization hyperparameter α is critical for getting valid conclusions. Too low α might yield the false impression that no modules exist or that they share more weights

than they really do. Too strong regularization may degrade the performance on the target function, discarding essential weights. Fortunately, there is a consistent heuristic for selecting the correct α, which follows from training a set of masks on the full task. We increase α as long as the performance does not start to drop. Then we reduce α slightly until the performance is adequate. The method is not very sensitive to the *exact* value α, and it transfers well between different network sizes (Fig. 7 in Appendix). We find that it is less critical but still important to tune the learning rate and the number of steps of the mask training process. We always check the validity of the selected hyperparameters by training a set of masks on the full, unmodified problem. We expect to see only a slight drop in performance.

Note that underfitting NNs tend to share more weights. Throughout our experiments, we found a large enough network is essential to avoid false conclusions about the reasons behind the sharing. See Fig. 9 in Appendix on how weight sharing changes with the network capacity.

3. Analyzing Module Properties

Let us reconsider P1 and P2 (see intro) in more detail, as they reflect the advantages of modular compositional design. According to our definition, the NN is not modular without P1. Moreover, disjoint modules prevent catastrophic interference (McCloskey & Cohen, 1989; Rosenbaum et al., 2019) of different functions, because changing the weights of one of the modules does not affect the others. Regarding P2, reuse has multiple advantages. It increases data efficiency to the extent that all relevant pieces are processed by the same module, which thus receives more training. Therefore, it also helps with generalization.

We construct synthetic datasets to verify whether NNs have natural inductive biases supporting P1 and P2. The experiments are kept as simple as possible while targeting the specific property of interest. The inputs of compositional modules can come from multiple sources. Consider arithmetic expressions of the form $a * b$ and $(c + d) * e$: the first operand of the multiplier can come directly from the input or from the output of the adder. The same holds for the outputs. Consider the four possible combinations: "same input/output (I/O), same function" is standard for training an NN on a single task. Our method is not of interest for this case because there are no sub-tasks to drive mask learning. Much more interesting are: "shared I/O, different function" to inspect property P1 (section 3.1), and "separate I/O, same function" to analyze P2 (section 3.2). We ignore the case "same I/O, different function" because the results of the two experiments above make it unlikely to provide new insights.

Using our tool, we can draw surprising conclusions about the modularity of typical NNs, which tend to satisfy P1 but not P2. Our experiments suggest that weight sharing across tasks is mostly driven by shared I/O rather than task similarity.

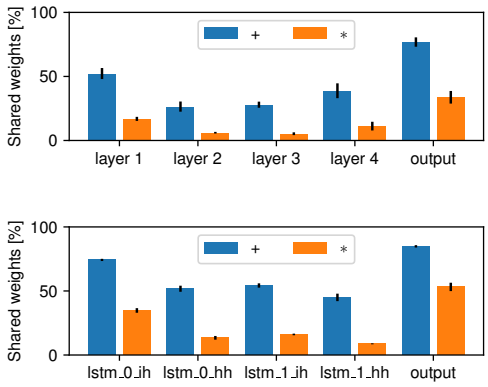


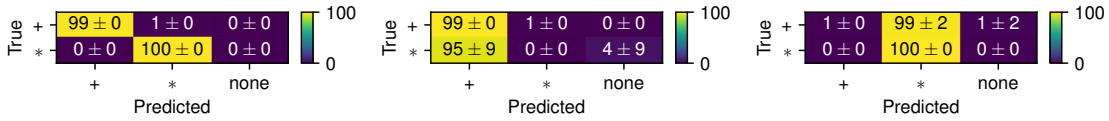
Figure 1. Percentage of shared weights on the addition/multiplication task per layer. FNN (top), LSTM (bottom)

3.1. Addition/Multiplication Experiments

The addition/multiplication dataset is designed to test the bias towards separating different functions (P1). It defines a single shared input and output. Two-digit numbers are encoded as one-hot vectors, each representing a digit. Each input is defined by two operands and a specification of the operation (a two-element one-hot vector), resulting in a 42-dimensional input vector. The target is a two-digit modulo 100 result, yielding 20 output elements.

First, we train the network to perform this task without any masking. Once performance is nearly perfect, we freeze its weights. We perform two stages of mask training: first, we train a set of masks on addition (multiplication examples excluded), then we repeat this procedure for multiplication. Note again that mask training cannot change the weights: they can only be kept or removed.

We analyze an FNN and an LSTM (Hochreiter & Schmidhuber, 1997) on this task. For LSTM, we repeat the full input for a fixed number of timesteps. The result is the output of the final step. No loss is applied at intermediate steps. Regardless of the architecture, we found the same general tendencies: There is more sharing in the input and output layers, less in the hidden layers (Fig. 1). We found that multiplication uses 3.8 times more weights than addition (Fig. 11 in Appendix), explaining the smaller proportion of its weights being shared. We conclude that there appears to be some bias in the network towards implementing different functions using distinct weights. On the other hand, the current amount of separation might still be inadequate to prevent interference and catastrophic forgetting. Increased sharing in I/O layers could be due to a switching/routing



(a) Mask trained on both $+$ and $*$ (b) Mask trained on $+$ (c) Mask trained on $*$

Figure 2. Analysis of FNN performance degradation on the addition/multiplication task. The y-axis shows the operation specified in the input. The x-axis shows the actual operation performed. "none" means the predicted number is neither the result of addition nor multiplication. The network ignores the operator specification and performs the one corresponding to the mask instead.

procedure used to select which operation to perform.

We also analyzed how performance breaks down on the task for which the mask was not trained (Fig. 2). Here the behavior of the FNN and the RNN differ. The FNN tends to ignore the function description in the input, and performs the operation corresponding to what the mask was trained on. The LSTM, on the other hand, tends to produce outputs that are invalid for both operations (Fig. 10 in Appendix), suggesting that it learned a solution where the two operations are more intertwined.

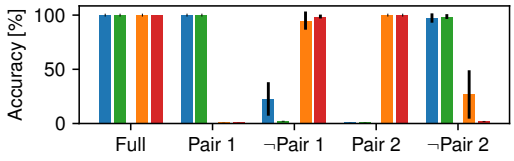


Figure 3. Double-addition task: accuracy on pair 1 and pair 2 with FNN and pair 1 and pair 2 with LSTM. The x-axis shows on what the applied mask was trained on. $\neg$ denotes an inverted mask.

3.2. Double-Addition Experiments

The double-add experiment is designed to test property P2. The task is to perform modulo 100 addition twice by reusing weights. The locations of inputs and outputs differ for the two instances, simulating a more realistic scenario of different data sources within the network when composing modules dynamically, without considering the additional problem of finding the right composition. Since the operation is the same and the data distributions are exact matches, this is the simplest setup that encourages sharing. With inputs a, b, c and d, the network should output $a + b$ and $c + d$. The encoding is the same as in section 3.1, resulting in 80 input and 40 output units.

We first train the network until convergence on the full task, then freeze its weights. We train a set of masks on $a + b$, followed by $c + d$. We analyze both FNN and LSTM, taking care to avoid activation interference. When both operations have to be performed simultaneously, sharing is impossible. Thus, for the FNN, we perform two forward passes. In each pass, we feed only one pair of numbers to the network (either a, b or c, d), while zeroing out the other. With LSTM, we investigate two different settings. In the first case, both pairs are presented together for a fixed number of steps, and the result is the last output. Hence, the LSTM is allowed to schedule the execution of the operations freely. In the second case, we feed a single pair for multiple steps with the other zeroed out, and then read its output. This procedure is then repeated for both pairs without resetting the state, which removes any incentive for solving them simultaneously since only one pair is seen at a time.

The results of all experiments are consistent: weight sharing is low (Fig. 12 in the Appendix). To verify how independent the modules are, we invert masks trained on pair A, removing all units needed for A, including the shared ones. We test the resulting network on pair B, where the performance decreases only slightly, confirming that they are independent (Fig. 3). No difference between the two LSTM variants is apparent (Fig. 13 in Appendix).

These observations show that P2 is violated even in this simple case. Realistic scenarios tend to be more complex as the data distribution for different instances of the operation might be different (with overlaps), providing even fewer incentives to share. Furthermore, comparing the results to those of section 3.1, it is apparent that sharing depends more on the location of the inputs/outputs than on the similarity of the performed operations. This behavior is undesired and calls for further research.

3.3. A Potential Explanation for Lack of Weight Sharing

Experiment in section 3.2 (and appendix E) consistently show that the internal parts of NNs resist to share computations. We hypothesize that the reason is the difficulty of routing in standard NNs. Inputs and outputs must be routed to different sources/targets to reuse modules in different compositions. In routing networks (Rosenbaum et al., 2019; Kirsch et al., 2018; Chang et al., 2019), this is done through hand-designed mechanisms. Without them, routing can only occur through the weights of the NN. But weight matrix-based transformations normally change the data represen-

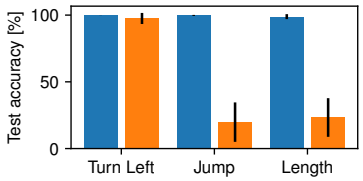
tations unless the weights have a special structure and the units have appropriate activation functions. This structure may be difficult to learn. In fact, our NNs should learn to dynamically change the composition in a problem-dependent way, which is an additional level of complexity. Our experiments suggest that typical NNs find it hard to learn to represent data in the same way on the different paths of information flow and process them by a single module. Instead, they prefer to learn a new set of weights for every use case of a function given enough capacity. We argue that this is an important issue that limits generalization and data efficiency. Research on inductive biases is needed to reduce such redundancy regardless of network size and task difficulty.

4. Analyzing Generalization Issues on SCAN Dataset

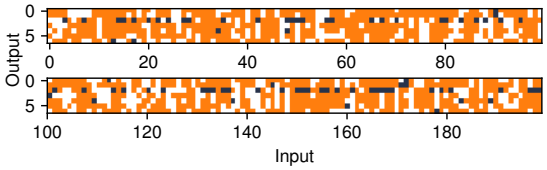
The failure of typical NNs at more systematic generalization is one of the central issues of current-day NN research. The SCAN dataset (Lake & Baroni, 2018) is designed for analyzing this property. It consists of compositional commands (e. g. "jump thrice"), to be translated into primitive output moves (e. g. "JUMP JUMP JUMP"). The dataset comes in different types of splits. The "simple" split is i.i.d. The "length" split has shorter training samples than test samples, but their structure is identical. Finally, the "add primitive" splits have a particular command presented in the training set but no compositions of this command with others.

It was previously shown that typical NNs have generalization issues regarding the "length" and some of the "add primitive" splits (Lake & Baroni, 2018). The reason for this is unclear, however. There might be two explanations. (a) The NN might have learned an algorithm to solve the problem, but the quality of the learned representations are insufficient. For the "length" split, noise might accumulate in the NN's state. For the "add primitive" split, the NN might be unable to form an analogy between the problematic primitive and the well-performing ones, because of limited variety in the training data. In either case, the NN has limited pressure to improve since the learned representations are sufficient to solve the entire training set. (b) Alternatively, the NN might not have learned the correct algorithm for the solution. It might have learned components of it, but it still needs new weights to solve a new problem of the same kind. Only in this case, we argue, has the NN failed to leverage the compositional nature of the problem.

We use the baseline 2 layer LSTM encoder-decoder model from (Lake & Baroni, 2018) (for details refer to Appendix F). We train the model on the "simple" split until convergence and freeze its weights. The learned representations and weights are capable of solving each split, which we verify at the end of the training phase. For each split, we train a different set of masks. We measure the performance of the network on the corresponding test split. The word embeddings are excluded from the masking process and remain unmodified after initial training.



(a) Performance on different splits



(b) Weights removed from last layer on "Add jump" split

Figure 4. Results of experiments on SCAN. (a) Accuracy on split shown on x-axis with masks trained on the full problem and with masks trained on split shown on x-axis. (b) Elements removed (dark blue) from the weight masks of the last layer trained on the "Add jump" split. Output 2 corresponds to "JUMP". Broken into 2 lines for better fit. The white elements correspond to weights not needed for any of the splits. Most elements corresponding to "JUMP" are removed.

Our experiments support explanation (b). We found a big generalization gap on the "length" and the "add jump" splits, whereas the "add turn left" caused only a slight degradation (Fig. 4a). The results are consistent with earlier findings (Lake & Baroni, 2018). The crucial additional insight gained with our tool's help comes from the fact that we started with a network whose representations and weights can already solve the full problem. Just by removing weights, the performance degrades on samples excluded from the training set, even though all the tokens and rules are still seen in different combinations. Thus, the solution learned on the full dataset does not embody the correct algorithm, confirming that the root of the problem is explanation (b).

By inspecting the weights we draw additional conclusions. In case of the "add jump" split, the most apparent difference is in the output layer. Most weights corresponding to "JUMP" are removed (Fig. 4b). This suggests that the network learned to detect patterns of cases when "JUMP" should be the output, and the last layer puzzles them together. In contrast to this pattern recognition approach, the generalizing algorithm for solving SCAN and other reasoning problems requires a variable manipulation-like solution (Garnelo & Shanahan, 2019; Lake et al., 2017). For the length split, the interpretation is not obvious.

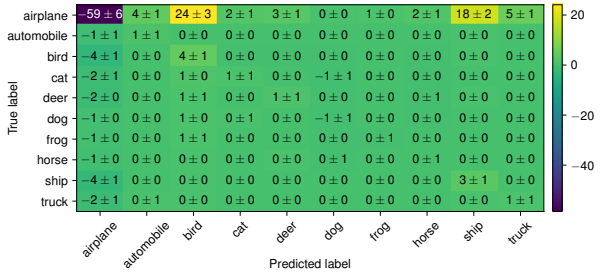


Figure 5. Absolute changes in the normalized confusion matrix on CIFAR 10, when features exclusive to "airplane" are removed [%]. The target class suffers a significant performance drop. Airplanes will be classified as birds or ships, probably because they rely heavily on the blue background.

5. Analyzing Convolutional Neural Networks

Findings from section 3.2 suggest that typical neural networks are biased against weight sharing. We investigate this effect in CNNs, which are believed to rely a lot on shared feature extractors (Zeiler & Fergus, 2014). Our method is able to highlight the set of weights solely responsible for individual classes and not shared by any other. First, we train a CNN on CIFAR 10 (details in Appendix G) and freeze its weights. Then we train a set of control masks on all classes. Next, we train a set of masks on all classes except some target class. This setup ensures that weights solely responsible for the target class will be the only ones missing from the resulting masks. We repeat this for all classes. The mask of the last classifier layer is only trained on the full task with all classes. The other stages copy it and keep it unmodified throughout the training process. If masks for the last layer were learned, all units responsible for the target class would be removed from the output projection, dropping the performance of the target class to zero, making it impossible to draw conclusions about the importance of class-exclusive features.

We compute the confusion matrix on the full validation set at the end of each training stage. We calculate the absolute difference between the confusion matrices with and without the target class removed, highlighting how the removal changes the classification. Interestingly, although relatively few weights are removed (Fig. 16 in Appendix), the true positive rate of the target class suffers a significant drop of up to 60%. This drop shows a heavy reliance on class-exclusive, non-shared features found by our method. The drop on non-target classes is insignificant and probably the result of noise introduced by the mask sampling process.

Analyzing the difference in misclassification rates provides insights as well: as the true positive rate drops, some classes are predicted more frequently. These rely on similar shared features. For example, removing "airplane" causes airplane images to be classified as "birds" and "ships" (Fig. 5). Indeed, they have a very distinct shared feature: the background color is blue. More insightful conclusions can be drawn for other classes (see Fig 18 in Appendix). Overall, the findings are in line with sections 3.2 and E: the network seems to resist modular weight sharing.

6. Related Work

Among the few attempts to identify modules in NNs Filan et al. (2020) is most related to ours. It identifies clusters of neurons with strong internal and weak external connectivity. Others detect communities (Watanabe et al., 2018) or cluster units hierarchically based on activation statistics (Watanabe, 2019). All of these methods, however, are unit-based, leaving them vulnerable to the issues discussed in section 2. Mutual information-based methods can help to detect important pathways in NNs (Davis et al., 2020). The above methods, however, neither ground the discovered modules in their functionality, nor analyze whether they support compositionality. PackNet (Mallya & Lazebnik, 2018) uses weights masks for continual learning. But its masks are obtained by pruning at the end of the training phase, when the NN must be retrained, and its tasks are learned sequentially, preventing possible co-adaptation of features. Moreover, the method was not used to analyze NNs and their modularity. Numerous papers analyze systematic generalization (Lake & Baroni, 2018; Bahdanau et al., 2019; Hill et al., 2020; Hupkes et al., 2019). However, they leave open the question whether lack of generalization is due to inferior quality of learned representations or to non-compositionality of learned algorithms.

7. Conclusion

Our new method for inspecting modularity in neural networks (NNs) is the first to identify modules by their functionality, making it easy to interpret what NNs really do. It is also a powerful tool for analyzing how the discovered modules of NNs share or separate weights based on functional properties. Unfortunately, it shows that in typical current NNs, weight sharing between modules responsible for solving certain tasks does not really reflect task similarity (as desired) but can mostly be explained by rather trivial shared I/O interfaces of solution-implementing modules. With disjoint module interfaces, typical RNNs and CNNs exhibit a consistent pattern of failing to share weights. Although functional modules should play an essential role in learning compositional and generalizable solutions, our case study on the SCAN dataset explains generalization problems of typical NNs by their failure to learn modular, compositional algorithms. We show that these problems are not merely due to insufficient quality of learned data representations. Our findings open many new directions for future research.

8. Acknowledgements

The authors wish to thank Aleksandar Stanić, Francesco Faccio, Kazuki Irie, Louis Kirsch and the anonymous reviewers for their constructive feedback. We are also grateful to NVIDIA Corporation for donating a DGX-1 as part of the Pioneers of AI Research Award and to IBM for donating a Minsky machine. This research was supported by an European Research Council Advanced Grant (no: 742870). We are also thankful for Weights & Biases for making their tool freely accessible for members of academia.

References

Andreas, J., Rohrbach, M., Darrell, T., and Klein, D. Neural module networks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

Bahdanau, D., Murty, S., Noukhovitch, M., Nguyen, T. H., de Vries, H., and Courville, A. Systematic generalization: What is required and can it be learned? In *International Conference on Learning Representations (ICLR)*, 2019.

Baldwin, C. Y. and Clark, K. B. *Design rules / the power of modularity*. MIT Press, 2000.

Chang, M., Gupta, A., Levine, S., and Griffiths, T. L. Automatically composing representation transformations as a means for generalization. In *International Conference on Learning Representations (ICLR)*, 2019.

Clune, J., Mouret, J.-B., and Lipson, H. The evolutionary origins of modularity. *Proceedings of the Royal Society B: Biological Sciences*, 280(1755), Mar 2013. ISSN 1471-2954. doi: 10.1098/rspb.2012.2863.

Davis, B., Bhatt, U., Bhardwaj, K., Marculescu, R., and Moura, J. M. On network science and mutual information for explaining deep neural networks. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 8399–8403, 2020.

Fernando, C., Banarse, D., Blundell, C., Zwols, Y., Ha, D., Rusu, A. A., Pritzel, A., and Wierstra, D. Pathnet: Evolution channels gradient descent in super neural networks. *arXiv preprint arXiv:1701.08734*, 2017.

Filan, D., Hod, S., Wild, C., Critch, A., and Russell, S. Neural networks are surprisingly modular. *arXiv preprint arXiv:2003.04881*, 2020.

Fodor, J. A., Pylyshyn, Z. W., et al. Connectionism and cognitive architecture: A critical analysis. *Cognition*, 28 (1-2):3–71, 1988.

Garnelo, M. and Shanahan, M. Reconciling deep learning with symbolic artificial intelligence: representing objects and relations. *Current Opinion in Behavioral Sciences*, 29:17–23, 2019.

Golkar, S., Kagan, M., and Cho, K. Continual learning via neural pruning. In *NeurIPS 2019 Workshop Neuro AI*, 2019.

Hill, F., Lampinen, A. K., Schneider, R., Clark, S., Botvinick, M., McClelland, J. L., and Santoro, A. Environmental drivers of systematicity and generalization in a situated agent. In *International Conference on Learning Representations (ICLR)*, 2020.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Hupkes, D., Dankers, V., Mul, M., and Bruni, E. The compositionality of neural networks: integrating symbolism and connectionism. *arXiv preprint arXiv:1908.08351*, 2019.

Jang, E., Gu, S., and Poole, B. Categorical reparametrization with gumbel-softmax. In *International Conference on Learning Representations (ICLR)*, 2017.

Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A. A., Milan, K., Quan, J., Ramalho, T., Grabska-Barwinska, A., et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526, 2017.

Kirsch, L., Kunze, J., and Barber, D. Modular networks: Learning to decompose neural computation. In *Advances in Neural Information Processing Systems 31*, pp. 2408–2418, 2018.

Kolouri, S., Ketz, N., Zou, X., Krichmar, J., and Pilly, P. Attention-based structural-plasticity. *arXiv preprint arXiv:1903.06070*, 2019.

Lake, B. M. and Baroni, M. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. In *Proc. International Conference on Machine Learning (ICML)*, volume 80, pp. 2879–2888, 2018.

Lake, B. M., Ullman, T. D., Tenenbaum, J. B., and Gershman, S. J. Building machines that learn and think like people. *Behavioral and brain sciences*, 40, 2017.

Lorenz, D. M., Jeng, A., and Deem, M. W. The emergence of modularity in biological systems. *Physics of life reviews*, 8(2):129–160, Jun 2011.

Maddison, C. J., Mnih, A., and Teh, Y. W. The concrete distribution: A continuous relaxation of discrete random variables. In *International Conference on Learning Representations (ICLR)*, 2017.

Mallya, A. and Lazebnik, S. Packnet: Adding multiple tasks to a single network by iterative pruning. In *Proc.*

Conference on Computer Vision and Pattern Recognition (CVPR), pp. 7765–7773, Salt Lake City, UT, USA, June 2018.

Marcus, G. F. Rethinking eliminative connectionism. *Cognitive psychology*, 37(3):243–282, 1998.

McCloskey, M. and Cohen, N. J. Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*, volume 24, pp. 109–165. Elsevier, 1989.

Rosenbaum, C., Cases, I., Riemer, M., and Klinger, T. Routing networks and the challenges of modular and compositional computation. *arXiv preprint arXiv:1904.12774*, 2019.

Schmidhuber, J. Towards compositional learning with dynamic neural networks. Technical Report FKI-129-90, Institut für Informatik, Technische Universität München, 1990.

von Dassow, G. and Munro, E. Modularity in animal development and evolution: elements of a conceptual framework for evodevo. *The Journal of experimental zoology*, 285(4):307—325, December 1999.

Watanabe, C. Interpreting layered neural networks via hierarchical modular representation. In *International Conference on Neural Information Processing*, pp. 376–388, 2019.

Watanabe, C., Hiramatsu, K., and Kashino, K. Modular representation of layered neural networks. *Neural Networks*, 97:62–73, 2018.

Zeiler, M. D. and Fergus, R. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pp. 818–833. Springer, 2014.

A. From Gumbel-Softmax to Gumbel-Sigmoid

In what follows, we use the notation by Jang et al. (Jang et al., 2017): k is the number of categories, class probabilities are π_i, elements of sample vector $y \in \mathbb{R}^k$ from the Gumbel-Softmax distribution (Jang et al., 2017) (also called Concrete distribution) are denoted by y_i. We will refer to $l_i = \log \pi_i$ as logits. We show how to sample from the Gumbel-Sigmoid distribution, the special case of $k = 2$, $l_2 = 0$ of the Gumbel-Softmax distribution.

First, we show that the sigmoid is equivalent to a first element $y_1 \in \mathbb{R}$ of the output vector of the two element softmax with $l_1 = x$, $x \in \mathbb{R}$ and $l_2 = 0$:

$$\sigma(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1} = \frac{e^x}{e^x + e^0} = \frac{e^{l_1}}{e^{l_1} + e^{l_2}} = y_1 \tag{4}$$

According to Jang et al. (Jang et al., 2017), sample vector $y \in \mathbb{R}^k$ from the Gumbel-Softmax distribution can be drawn as follows:

$$y_i = \frac{e^{\frac{1}{\tau}(l_i + g_i)}}{\sum_{j=1}^{k} e^{\frac{1}{\tau}(l_j + g_j)}} \tag{5}$$

where $g_i \sim \text{Gumbel}(0, 1)$ are independent samples from the Gumbel distribution. We are interested in the special case of a sigmoid, which we showed to be equivalent to the y_1 in $k = 2, l_2 = 0$ case:

$$y_1 = \frac{e^{\frac{1}{\tau}(l_1 + g_1)}}{e^{\frac{1}{\tau}(l_1 + g_1)} + e^{\frac{1}{\tau} g_2}} \tag{6}$$

Which can be rearranged:

$$y_1 = \frac{1}{1 + e^{-\frac{1}{\tau}(l_1 + g_1 - g_2)}} = \sigma\left(\frac{1}{\tau}(l_1 + g_1 - g_2)\right) \tag{7}$$

Writing out the inverse transformation sampling formula for $g_i \sim \text{Gumbel}(0, 1)$; $g_i = -\log(-\log U_i)$, were $U_i \sim \text{U}(0, 1)$ are independent samples from the uniform distribution, we get:

$$y_1 = \sigma\left(\frac{1}{\tau}(l_1 - \log(-\log U_1) + \log(-\log U_2))\right) \tag{8}$$

$$y_1 = \sigma\left(\frac{1}{\tau}\left(l_1 - \log\frac{\log U_1}{\log U_2}\right)\right) \tag{9}$$

Finally, by renaming $s(l) = y_1$ and $l = l_1$ (we have just a single logit), we obtain the sampling formula for Gumbel-Sigmoid:

$$s(l) = \sigma\left(\frac{1}{\tau}\left(l - \log\frac{\log U_1}{\log U_2}\right)\right) \tag{10}$$

A.1. Straight-Through Estimator

Samples from the Gumbel-Softmax distribution can directly be converted to a sample from the categorical distribution as:

$$c_i = \mathbb{1}_{i = \arg\max_i y_i} \tag{11}$$

Which gives rise to the straight-through estimator which can provide hard samples while permitting gradient flow ($[x]$ is an operator for blocking gradient flow):

$$c_i = [\mathbb{1}_{i = \arg\max_i y_i} - y_i] + y_i \tag{12}$$

Since in the Gumbel-Sigmoid we have $k = 2$ categories and $\sum_i y_i = 1$, $\arg\max$ can be replaced by checking whether $y_i > 0.5$:

$$c_i = [\mathbb{1}_{y_i > 0.5} - y_i] + y_i \tag{13}$$

B. Implementation Details

Our method is implemented in PyTorch, and available on https://github.com/xdever/modules. We use the Adam optimizer, a batch size of 128, a learning rate of 10^{-3}, and a gradient clipping of 1. To obtain better mask quality, we use 4 samples of masks per batch, meaning that each quarter of the batch uses a different mask sample. For non-LSTM networks, we use the ReLU activation function. The Gumbel-sigmoid always has a temperature of $\tau = 1$.

All figures in this paper show mean and unbiased standard deviations calculated over 10 runs with different seeds.

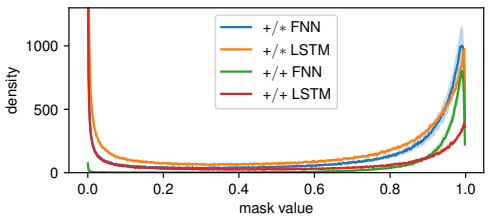
For most of our experiments the regularization coefficient α is specified as $\beta = b\alpha$, where b is the batch size.

B.1. Calculating the Degree of Weight Sharing

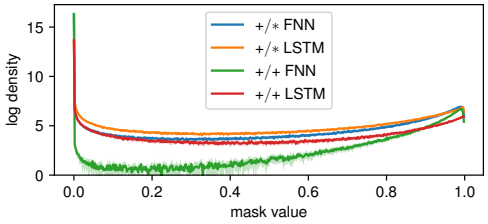
Given a target mask $t_i \in \{1, 0\}$ and a reference $r_i \in \{1, 0\}$, the proportion $s(t, r) \in [0, 1]$ of t shared with r is calculated by:

$$s(t, r) = \frac{\sum_i t_i r_i}{\sum_i t_i} \tag{14}$$

Reference r represents the weights used by other tasks. If there are more than two tasks, r is the logical OR of all of them, except for t.

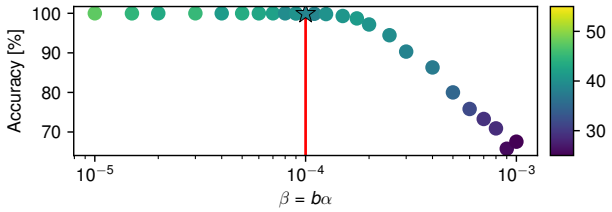


(a) Linear scale, small values cut off

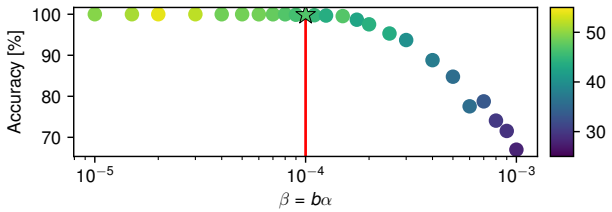


(b) Log scale, all values

Figure 6. Histogram (normalized as an 500-bin PDF) of mask values on different tasks. (a) Shown on linear scale. Values < 0.0002 (bottom 10% of the first bin) are removed from calculation because their number vastly exceeds the number of kept weights for most of the networks, making the details invisible. (b) All masks (without small values removed) on log-scale.



(a) Big network



(b) Medium network

Figure 7. Sensitivity analysis for hyperparameter $\beta = b\alpha$ (b is the batch size) on addition/multiplication experiments. Note the logarithmic x-axis. The color shows the amount of total sharing [%]. Red line and the star indicates the value chosen for our experiments. Each point is a mean of 10 independent seeds. The network is not very sensitive to the exact choice of β. (a) Big network, with 5 layers of size 2000. (b) Medium network, with 5 layers of size 800. It can be seen that the hyperparameter transfers well between networks sizes.

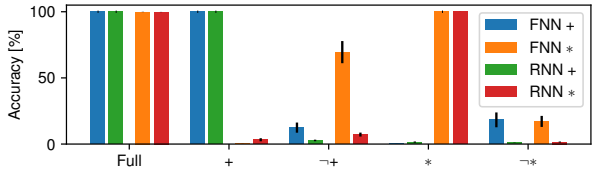


Figure 8. Accuracy of addition/multiplication task on $+$ and $*$ with FNN and $+$ and $*$ with LSTM. The x-axis shows on what the applied mask was trained on. $\neg$ denotes an inverted mask.

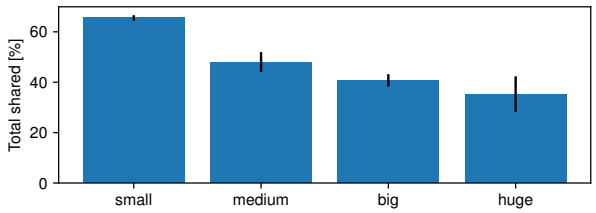


Figure 9. Addition/multiplication task: the total proportion of shared weights for the "add" operation for different network sizes. "small" means a 4 layer network with hidden sizes of 400, 400, 200, "medium" 5 layers / hidden sizes of 800, "big" 5 layers / 2000, "huge" 5 layers / 4000.

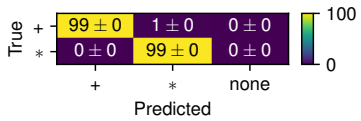
C. Addition/Multiplication Experiments

Since our experiments show that modulo 100 multiplication requires lots of weights, we use reasonably big networks for this experiment. The FNN is 5 layers deep, each layer having 2000 units. The LSTM has a state size of 256 because further increases cause overfitting. We train the network for 20k steps before freezing. Each mask training phase takes an additional 20k steps. Mask training uses a learning rate of 10^{-2}, 10x higher than the one used for the weights, and regularizer $\beta = 10^{-4}$. LSTM uses 3 time steps, where the input is repeated over every step. The dataset is generated by sampling numbers and operations randomly.

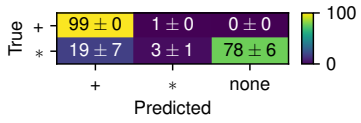
Fig. 10 shows that unlike in the FNN case, if we test the LSTM on the operation opposite to what the mask was trained on, the network performs invalid operations in roughly 70% of the cases. Compare this to the FNN case in Fig. 2.

Fig. 9 shows that while the network with 5 layers and 2000 units is already way oversized for the task (the small, 3 layer networks of 400, 400, 200 can solve the task), sharing still changes with increased network size.

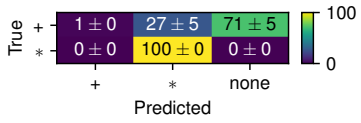
Fig. 8 shows limited separability of addition and multiplication tasks. Given the high proportion of sharing, especially in the input and output layers, the results are as expected: inverted masks do not perform well.



(a) Mask trained on both + and ∗

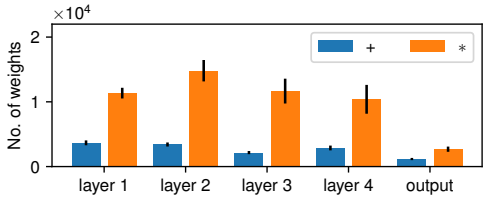


(b) Mask trained on +

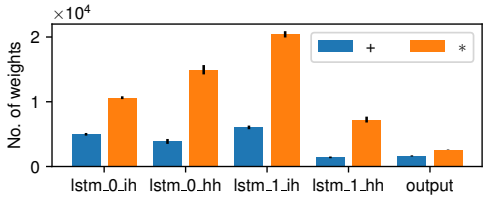


(c) Mask trained on ∗

Figure 10. Analysis of the RNN's performance degradation in the addition/multiplication task. The y-axis shows the operation specified in the input. The x-axis shows the actual operation performed. "none" means the predicted number is neither the result of addition nor multiplication. The network performs invalid operations in most of the cases.

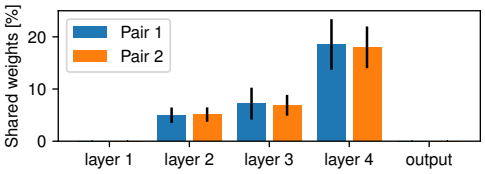


(a) FNN

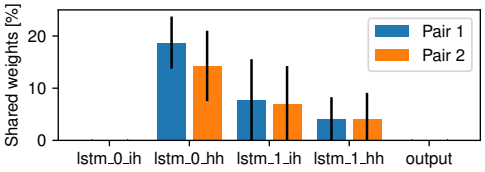


(b) LSTM

Figure 11. Addition/multiplication taks: number of weights per operation for each layer in (a) feedforward network, (b) LSTM.



(a) FNN



(b) LSTM

Figure 12. Double addition task: percentage of weights shared per operation in case of (a) feedforward network, (b) LSTM. The first and last layers have zero shared weights.

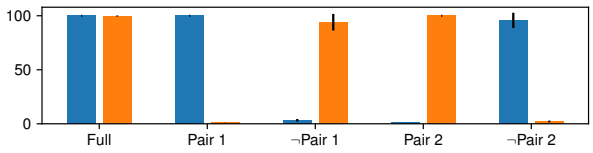


Figure 13. Double addition task: accuracy on pair 1 and pair 2 LSTMs with only one input presented at a time, to prevent interference. The x-axis shows on what the applied mask was trained on. ¬ denotes an inverted mask. Compared to Fig. 3, no improvement is apparent.

D. Double Addition Experiments

The training protocol for double-add experiments is identical to the one in Appendix C, except that the FNN variant uses mask regularizer of $\beta = 4 * 10^{-4}$. LSTM uses 6 steps in total (3 steps per operation). In the full input case, both tuples are presented at the input for all 6 steps, and the output is read from the last step. If only one tuple is presented at a time, the first tuple is shown for the first 3 steps, resulting in an output at the 3rd step, the second tuple is presented for the next 3 steps, resulting in an output at the 6th step.

The degree of weight sharing is shown in Fig. 12.

The LSTM with only one input presented at a time is shown in Fig. 13. No improvement is apparent compared to the setting where the LSTM can schedule the operations freely (Fig. 3).

E. Transfer Learning Experiments

We also analyzed the degree to which P2 is violated in a more complex setting—see section 3.2. We measure the

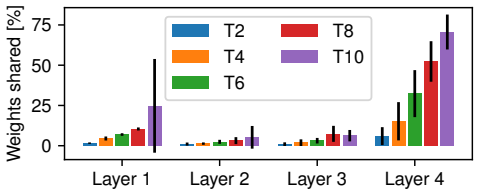


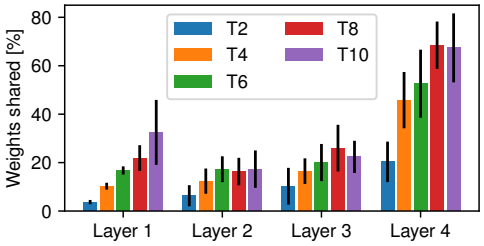
Figure 14. Percentage of weights shared per layer after every second task on permuted MNIST. Each task corresponds to a permutation. The last layer has the lowest capacity, filling up first, forcing subsequent runs to share weights.

amount of transfer in a continual learning setup. A popular benchmark for continual learning is permuted MNIST (Kirkpatrick et al., 2017; Golkar et al., 2019; Kolouri et al., 2019). A sequence of tasks is created by applying task-specific permutations to MNIST images. Spatially close pixels of a given digit may no longer be observed in nearby locations. We sequentially train an FNN on these tasks, switching to a new permutation once adequate performance is reached.

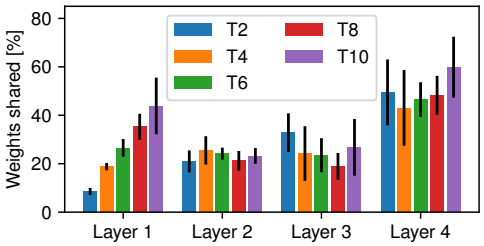
Continual learning is closely related to transfer learning. The new tasks are expected to reuse knowledge learned from previous tasks, resulting in faster convergence and fewer new parameters. Popular methods for continual learning revolve around freezing necessary weights when a new task is added (Fernando et al., 2017; Mallya & Lazebnik, 2018; Golkar et al., 2019). We adapt our method in a similar spirit. Masks and weights are trained simultaneously. At the end of training on a task, we freeze the used weights and save the masks. The free weights are reinitialized, and a new set of masks is allocated for the next task. We obtain a set of masks, each corresponding to a permutation. This process is reminiscent of PackNet (Mallya & Lazebnik, 2018), with the difference of being single-pass, in contrast to PackNet's three-step process (train, prune, retrain).

Since the tasks differ only by the permutation of the inputs, it is sufficient to re-train a new first layer of the network, which undoes the permutation and implements the same transformation as the original. The rest of the net can be reused without modification. However, we found only insignificant weight sharing as long as there is sufficient free space available (Fig. 14). Only once all the capacity is filled up, weights start being shared. This is especially apparent in the last layer of the network.

The initialization of new masks at the beginning of the training phase partially destroys knowledge about the previous stage's connectivity, which we suspect might be the reason for failing to share weights. To test this hypothesis, we initialized elements of new masks corresponding to used weights by a significantly higher probability than those corresponding to new weights. Intuitively, this favours the old,



(a) New weights with $P = 0.5$.



(b) New weights with $P \approx 0.27$.

Figure 15. Percentage of weights shared per layer after every second task on permuted MNIST, for a network with masks initialized to prefer reuse of old weights. Each task corresponds to a permutation. The last layer has the lowest capacity, filling up first, forcing the subsequent runs to share weights. Decreasing probability of new weights forces increased sharing, but it is far from the ideal 100%. Moreover the sharing in the first, permuted layer also increases, which should not be the case.

frozen network and adds new weights with low probability. The mask logits corresponding to weights of the previous task are initialized to 2 (corresponding to $P \approx 0.88$), the logits for newly initialized weights to either 0 ($P = 0.5$, Fig. 15a) or -1 ($P \approx 0.27$, Fig. 15b). Although this improved the sharing, it is still far from the target 100%. It also increased the sharing of the first layer, which should not share at all.

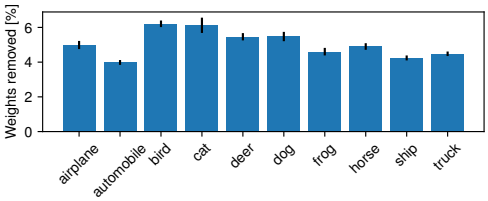
Such observations confirm once more that P2 is not achieved: modules are not reused. The same functionality is redundantly re-learned instead.

E.1. Implementation Details

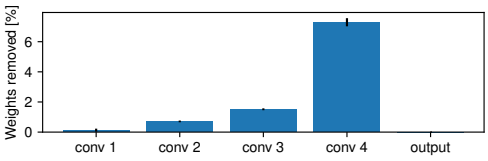
In the transfer learning setup, we train 10 permutations of MNIST with the same network. These experiments use 8 mask samples per batch. Each phase takes 30k steps. The learning rate is 10^{-2}. The network is 4 layers deep, with hidden sizes of 800, 800, 64. We are using a mask loss of $\alpha = 10^{-5}$.

F. SCAN Experiments

We modify the baseline (Lake & Baroni, 2018) by reducing the size of word embeddings to 16 because SCAN has



(a) Per class



(b) Per layer

Figure 16. The percentage of weights removed from a net trained on CIFAR 10: (a) per class (b) per layer. The output layer is excluded from the masking process, thus no weights are removed from it.

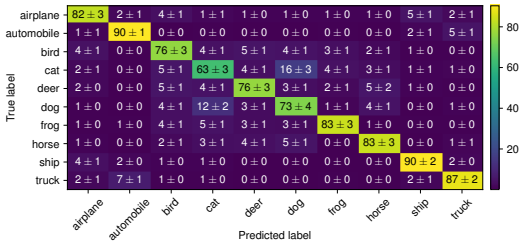


Figure 17. Confusion matrix on CIFAR10 with masks trained on all classes.

only 13 input and 6 output tokens. We find that the old, full size word embeddings are very redundant. So there are many possible input-to-hidden weight configurations, greatly decreasing the probability of sampling one of them, which causes the input-to-hidden layer to be removed by the thresholding procedure. The reduced embedding does not suffer from such problems.

The training procedure uses a batch size of 256 and a gradient clipping of 5. The mask regularizer is $\beta = 3 * 10^{-5}$, the mask learning rate is 10^{-2}. We train the network for 25k steps without masks and for another 25k steps for each set of masks.

G. CNN Experiments on CIFAR10

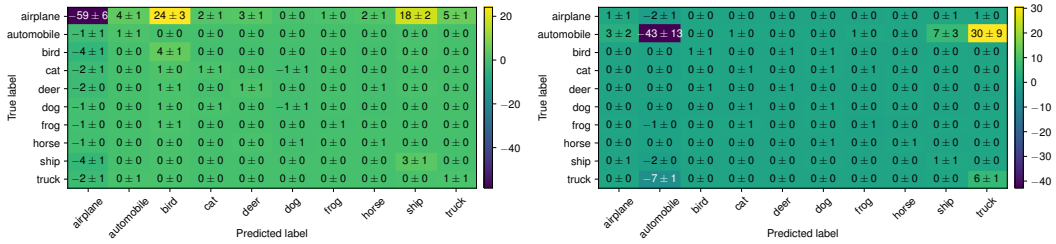
G.1. Detailed Description

We use a learning rate of 10^{-3} and a mask regularizer $\beta = 10^{-4}$. We pre-train the network for 20000 steps, followed by 20000 steps for each of the masks, including the reference. See Table 1 for details of the architecture.

G.2. Analysis of All of the Classes

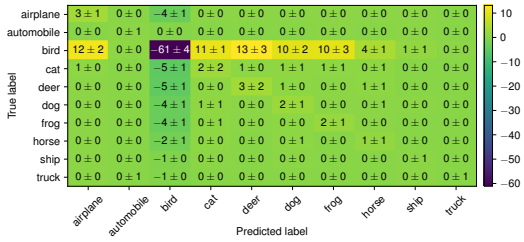
Fig. 18 shows the confusion matrix difference for all classes in CIFAR 10. The most surprising observation is that the decrease in performance for each of the classes is substantial, ranging from 40 to 60%. This shows the heavy reliance on class-exclusive features. The percentage of removed class-specific weights can be seen in Fig. 16.

Analyzing confusion matrix differences yields interesting insights. "Airplane" is confused with"bird" and "ship", probably because of the blue background. Classes "cat" and "dog" tend to be confused with each other—removing exclusive feature detectors for one improves the performance of the other. "Truck" and "car" are highly related, probably because of similar shapes, such as tires, and similar backgrounds, such as the road.

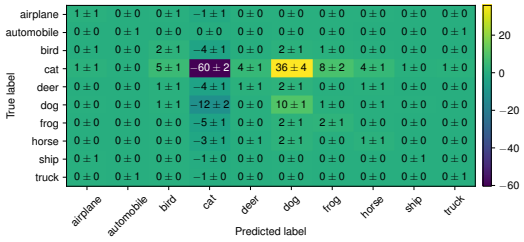


(a) airplane

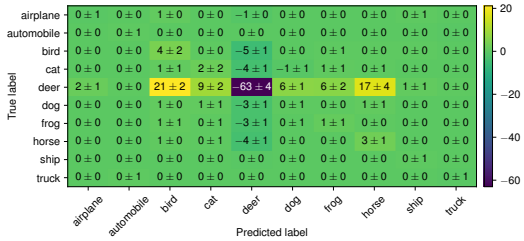
(b) automobile



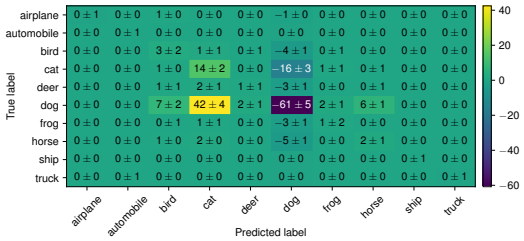
(c) bird



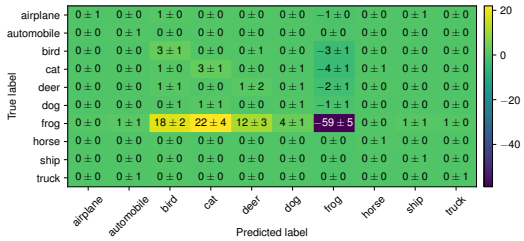
(d) cat



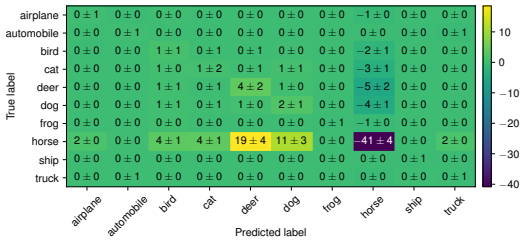
(e) deer



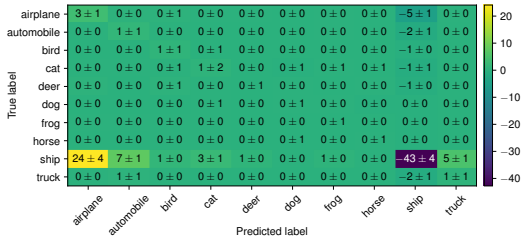
(f) dog



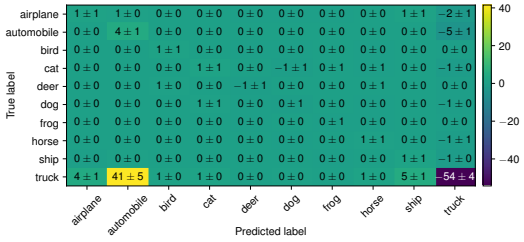
(g) frog



(h) horse



(i) ship



(j) truck

Figure 18. The change in confusion matrix for all CIFAR10 classes, when class indicated by the caption is removed.

Table 1. CNN architecture used for CIFAR 10 experiments

Index	Operation	Inputs	Outputs	Kernel	Padding	Activation	Dropout
1	Conv	3	32	3x3	1	ReLU	-
2	Max pooling	32	32	2x2	0	-	-
3	Conv	32	64	3x3	1	ReLU	-
4	Max pooling	64	64	2x2	0	-	-
5	Conv	64	128	3x3	1	ReLU	0.25
6	Max pooling	128	128	2x2	0	-	-
7	Conv	128	256	3x3	1	ReLU	0.5
8	Spatial average	256	256	-	-	-	-
6	Feedforward	256	10	-	-	Softmax	-