Functional Module Development in Abstract Visual Reasoning Networks

Iyngkarran Kumar

Department of Computer Science, Durham University

Abstract—We study how groups of neural network parameters with specific functionalities (functional modules) develop in neural networks trained on abstract visual reasoning (AVR) problems. To identify the weights that are specialised to a particular task over training, we optimise binary masks on the network weights. Using this tool, we are able to locate weights in a state-of-the-art AVR model that become specialised to the task of perceiving shapes in AVR problems. Further analysis shows that these modules comprise only ten percent of the total weights in the network and that there is a significant degree of parameter sharing between the functional modules responsible for the perception of different shapes. In addition, we make a number of yet unexplained empirical observations about the properties of these functional modules that provide a clear direction for future study.

Index Terms—Computer Vision, Deep Learning, Machine Learning

1 Introduction

Modern deep learning systems are notoriously opaque, offering little insight into how exactly the functionality that the network executes is implemented. Dissolving this black-box nature of artificial neural networks (ANNs) is thus critically important as deep learning technologies are integrated into human society, with the field of neural network interpretability [1], [2] arising to tackle this problem directly. Interpretable neural networks offer many clear benefits to their black-box counterparts, such as the ability for operators to clearly diagnose issues when the systems inevitably behave out of line with human expectations, as well as the potential for making novel insights when examining systems that achieve superhuman performance across domains of interest [3].

A key property of any interpretable neural network is *modularity*, which is conceptualised in [4] as the identification of distinct regions in the network performing computations independently, and feeding only the inputs and outputs of these computations to each other. One can examine a network for multiple forms of modularity (see Section 2.1), but in this work we focus on *functional modularity* [5] - that is, we define a module in a neural network as a group of neurons and parameters that implement some specific functionality relevant to the task that the whole network is trained upon. Whilst previous work has studied the existence of functional modules in trained neural networks [5], considerations of how functional modules *arise* during the training process itself appear to have been relatively neglected. At the outset of training, neural network weights may simply be viewed as a large collection of random floating point numbers, yet over the course of many iterations of stochastic gradient descent (SGD) intelligent behaviour arises from these systems. How does SGD mould the network parameters during training? It is this question we look to explore in this work.

In order to take preliminary steps in studying how functional modules arise in neural networks, we study the development of functional modules in a special class of neural networks - those trained on abstract visual reasoning (AVR) tasks (see Figure 2). Abstract visual reasoning problems, which are introduced in further depth in section 2.2.1, were chosen to conduct a study of functional modularity development due to their inherently compositional nature. Solving an AVR task requires multiple sub-tasks to be performed largely independently of one another; it therefore seems that a priori, we have strong reason for neural networks trained on AVR tasks to develop functional modules. In particular, we study functional modules that develop in the state-of-the-art AVR model, the Scattering Compositional Learner (SCL) [6], chosen due to it's strong performance on AVR tasks and high parameter efficiency.

To study functional modules at various points during training of SCL, we use a method introduced in [5]. This method identifies the weights in a network that are responsible for implementing some sub-task of the main task that the network was trained on, by training a binary weight mask on the original weights. In doing so, this binary weight masking algorithm (BWM) removes weights that are independent of the network's ability to perform this sub-task through zero-ablation, whilst preserving the weights that are required for strong performance. We apply the BWM at three stages during SCL's training, specifically when SCL is attaining 50%, 70% and 90% accuracy on it's training dataset, allowing us to study functional module development over an expected 'accuracy window' that covers 40% of SCL's training.

Results of the masking process suggest that distinct functional modules are indeed found by the BWM algorithm, although over a smaller range of SCL accuracies than initially intended. We see that some of these functional modules are able to reattain a significant fraction of the unmasked model's performance on a given sub-task, whilst only containing 10% of the weights of SCL. We also observe that the properties of these modules follow expected trends

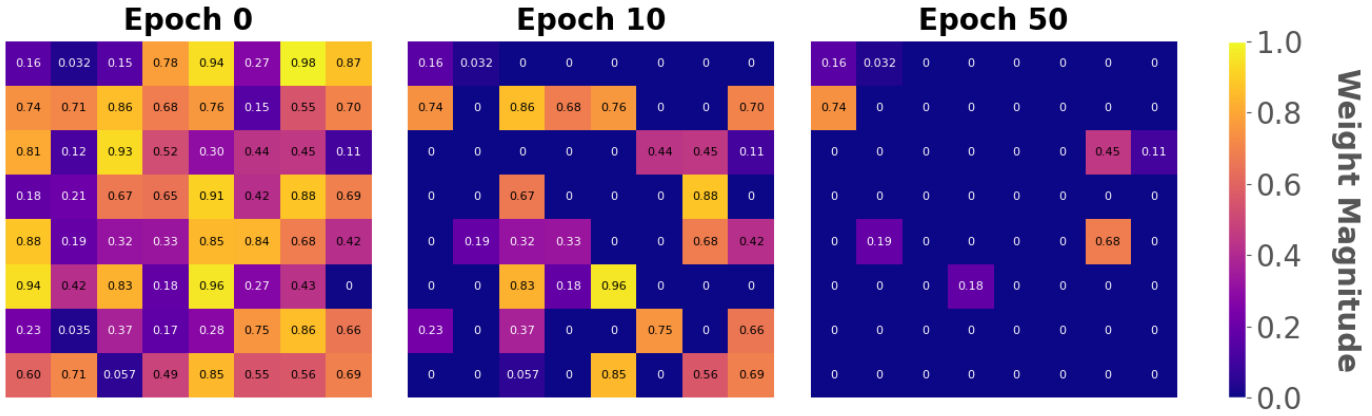

Fig. 1: An illustration of the binary weight masking tool that identifies functional modules, first introduced in [5]. A binary weight mask is applied to the weights of a trained model, and then optimised with respect to a weighted combination of mask sparsity and performance on a given task. This process leads the mask to remove (zero-ablate) the weights that are *not required* for the task, leaving only the parameters that are critical for it's execution (the functional module). Section 3.1 discusses the algorithm in further detail. In this work, we track functional module *development* for shape processing by optimising the binary masks on AVR problems which have a constant shape (see Figures 4(a),4(b)) at *various stages* of model training. **The epoch numbers in the figures above are for the optimisation of the binary mask, not the weights of the network. Model weights are fixed during the binary mask optimisation.**

during training, such as the tendency for these modules to become increasingly *sparse* and *spatially localised* over training. Furthermore, we show that most of the functional modules we discover share a significant fraction of weights with one another - a result that is to be expected given the nature of the sub-tasks that we identify modules for.

This paper is structured as follows: In Section 2, we summarise recent work in the field of neural network modularity and briefly overview the field of abstract visual reasoning in neural networks. This is followed by Section 3, in which we present the methods utilised to study functional module development in SCL throughout it's training; section 3.1 gives a detailed discussion of the binary weight masking algorithm (BWM), followed by section 3.2 and 3.3 which respectively detail the AVR datasets and the SCL model. Results of our analysis of the functional modules are given in Section 4, after which a discussion of the limitations of this work and avenues for further work are given in Sections 5 and 6. Supporting code for this project can be found at https://github.com/IyngkarranKumar/AVR-functional-modularity.

2 Related Work

2.1 Modularity and Interpretability of Neural networks

Modularity in ANNs can broadly be split into two forms, the first of which is *learnt* modularity, which is concerned with the development of modular structures in the trainable parameters of the network during the process of training. Such modularity may arise if neurons are connected to one another but must compete for access to information [1]. One may also consider more *explicitly-built* modularity, in which the various parts of the ANN architecture are designed with specific functionality in mind, in contrast to training a large monolithic network.

This work focuses on learnt modularity, where we use a method first introduced in [5] to probe for groups of parameters that execute specific sub-tasks, which combine to perform the main task that the network was trained on (functional modularity). This is achieved by training a binary weight mask on the pretrained weights of a neural network, optimising this weight mask with respect to a specially designed loss function. The loss function in question consists of a performance metric and a regularisation term that controls the number of weights included in the weight mask (see section 3 for a detailed discussion of this process). With this tool, [5] finds subsets of weights in convolutional neural networks (CNNs) trained for image classification that are responsible for identifying particular classes.

Another line of work that analyses learnt modularity is that of [7] which uses a spectral clustering algorithm to quantitatively measure the degree of clustering in the parameters of trained neural networks. In [8], this spectral clustering method is used to analyse individual clusters of neurons, and quantify the degree to which these clusters have specific functionality.

In addition to this, the results of [9], [10] show that vision models trained on natural image datasets develop clear specialisation at multiple scales; [9] reveals that individual neurons are able to learn commonly occurring features in the training dataset, whereas [11] shows that multiple layers can learn specialised functionality, in a process referred to by the authors as *branch specialisation*. [12] conducts a similar study instead for transformer-based language models, showing that individual neurons in the feed-forward layers of these networks learn to identify syntactic and semantic patterns that frequently appear in the text corpora that the model was trained on.

Modularity in neural networks may also be *explicity-built* when selecting the architecture of the model, with sensible choices of architecture enhancing both the interpretability and systematic generalisation capability of the model in question [13], [14], [15]. [13] aims to improve upon the lack

of interpretable structures in monolithic deep learning models by training gated modular neural networks (GMNNs) and determining the extent to which the gate learns a useful problem decomposition and the individual modules learn simpler auxilliary tasks. Their results show that whilst GMNN modules do learn useful subtasks, existing methods of training these architectures do not *guarantee* an interpretable problem decomposition. [16], [17] follows a similar paradigm of explicity building in modularity by training a self-assembling neural modular network that uses a recurrent neural network (RNN) to dynamically infer a series of pre-determined neural modules; these are then composed to answer a question from a Multi-hop QA dataset [18]. Once again, they find that this explicity built modularity improves upon the baseline performance of models that do not build in explicit submodules, whilst also aiding the interpretability of the computation taking place.

2.2 Computational models on AVR tasks

We now review the structure of AVR tasks, as well as previous work in developing benchmarks and computational models to solve AVR problems. We refer the reader to [19] for a comprehensive summary of existing benchmarks and a taxonomy of current AVR models.

2.2.1 Background: AVR problem structure

It is strongly recommended that the reader consult the associated figures that display visual examples of AVR problems, particularly Figure 2.

In an AVR task, a list of images $\mathcal{O} = \{o_i\}_{i=1}^{8}$ is given in the form of a 3×3 matrix with a missing entry (usually in the bottom right corner). This 3×3 matrix is referred to as the *context matrix*. An answer set $\mathcal{A} = \{a\}_{i=1}^{8}$ is also provided, from which a solver must select an image with the aim of choosing $a \in \mathcal{A}$ that best completes the context matrix. During the creation of the matrices, various formally defined transformation rules are applied along the matrix rows and columns, meaning that just *one* of the $a \in A$ correctly completes the matrix, which we denote as a^*; the rest violate at least one of the transformation rules. Thus, a successful solver is one that is able to determine a^* from the answer set with a high degree of consistency.

AVR networks were chosen for this study due to their *inherently compositional nature*, requiring multiple independent sub-tasks to be solved in order to find a^*. Let us make this explicit through consideration of an AVR problem in further depth (Figure 3). The core element of an AVR problem instance is the **entity** - this is a 2 dimensional geometric shape that has four attributes which can vary across panels: type, size, colour and orientation. Possible values for the attributes are given in Table 1. Entities can then be composed into more complex configurations known as **layouts**, which come in three types: centre-layouts, which group entities individually, 2×2 layouts, which groups four entities into a 2×2 grid, or the 3×3 layout, which groups 9 entities into a 3×3 grid. Examples of each type of layout are given in Figures 10(a), 2, and 4(a) respectively. Layouts have two attributes that can change across panels; layout *Number* and *Position*. Briefly, the Number attribute determines how many entities are *actually present* in the layout; to illustrate, a 2×2 layout may only be populated with 3 entities, with the other slot left empty, as is the case in some panels of Figure 2. The Position attribute then determines which positions in the layout are filled with entities. Further description of these attributes is given in Table 1. AVR problems can therefore be seen as the task of determining how entity and layout attributes transform from panel to panel, and using these 'transformation rules' to select the correct answer panel from the answer set. There are four transformation rules that determine how entity and layout attributes transform from one panel to the next:

1) **Constant rule** - The attribute of the entity/layout remains constant. The *shape type* of Figure 10(a) shows an instance of this rule . This is the simplest rule to spot and extrapolate to the missing panel.
2) **Progression rule** - The attribute of the entity/layout monotonically increases/decreases. An instance of this rule is shown in Figure 2 for the *shape type* - the number of sides of the shape type decreases by one as we move along the row (pentagon, square, triangle).
3) **Arithmetic rule** - The attribute of the entity/layout in the third panel can be derived by summation/subtraction of the attributes in the first and second panel. An instance of this rule is shown in Figure 4(a) for the layout *Number* attribute - the number of entities in the third panel is give by the difference between the number of entities in the first panel and second panel.
4) **Distribute Three rule** - 3 distinct values of the attribute for the entity/layout are sampled from the permitted values (Table 1), and then permuted in the three panels of each row. An instance of this rule is given for the *layout number* attribute in Figure 2 - each row has a panel with either $1, 3$ or 4 entities, and there is no clear progression/arithmetic relationship between panel number and number of entities.

We refer the reader to [20] for a description of the (I-)RAVEN problem structure in even greater depth.

In order to correctly solve an AVR problem instance, we therefore see that a solver must develop the ability to do two things:

1) Identify the entities, layouts, and their attributes in a particular panel. This ability corresponds to *perception* in humans. It is worth noting that it is difficult to consciously monitor acts of perception - our brains identify shapes, colours and sizes automatically.
2) Determine the *rules* that the attributes of the entities and objects in a particular problem instance follow. This ability corresponds to *reasoning* in humans, and is requires conscious effort to do.

2.2.2 AVR benchmarks

At present, four datasets are usually used to train and benchmark AVR models; these are PGM [22], RAVEN [21], I-RAVEN [23] and RAVEN-FAIR [24]. PGM is the largest of these datasets, containing approximately 10^7 problem

	Type	Permitted values
Shape type	Entity attribute	circle, triangle, square, pentagon, hexagon
Colour	Entity attribute	10 grey-scale values uniformly distributed in $[0, 255]$
Size	Entity attribute	6 scaling factors uniformly distributed in [0.4, 0.9]
Orientation	Entity attribute	8 rotation angles uniformly distributed in $[0, 2\pi]$
Number	Layout attribute	A single integer value sampled from $[0, l]$ where l is the maximum number of entities in the layout
Position	Layout attribute	k positions from the allowed entity positions of the layout, where k is the Number attribute.

TABLE 1: Further details of I-RAVEN problem structure.

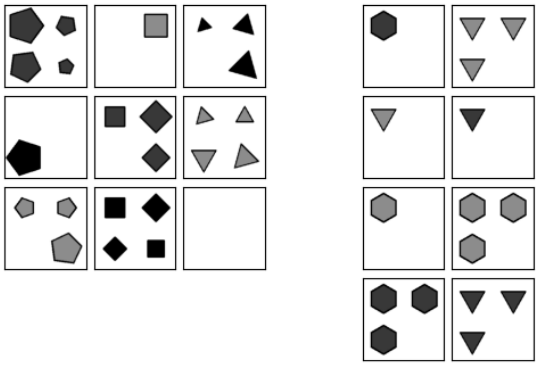

Fig. 2: An example AVR instance. We see that each panel has up to four **entities**, which vary in shape type, colour, size and orientation. These entities are arranged in 2×2 grid - one of the 3 **layouts**. Furthermore, notice that the 2×2 grid in each panel does not have the same number of entities, nor are they in the same position (These are the number and position attributes of layouts). Finally, consider the **transformation rules** that the panels obey row-wise. Here, we need to only determine the transformation rules for the **layout Number** and the entity **shape type** attributes. The layout number attribute obeys the Distribute Three rule, and the entity shape type obeys the Progression rule. The correct answer is therefore the panel with a single triangle (first panel in the second row of the answer set).

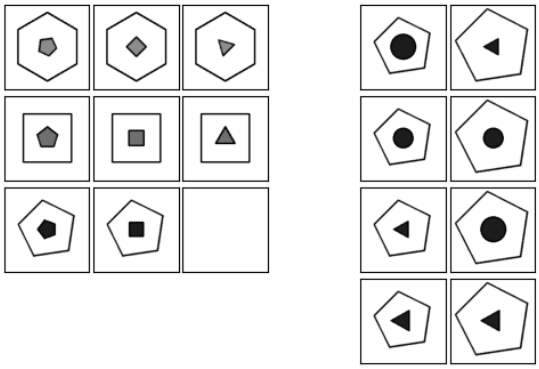

Fig. 3: An example AVR problem instance. The context matrix is on the left, with the answer set on the right. Models trained on the original RAVEN dataset [21] were able to achieve superhuman performance without using the context matrix, indicating a clear flaw in the RAVEN dataset.

instances spread over 8 regimes of varying difficulty. The dataset was originally introduced to test out-of-distribution generalisation of AVR models, motivated by the fact that the training and test suites of preceding benchmarks were found to be generated from distributions that were too similar [19]. [21] introduced RAVEN in 2019 as a benchmark with more complex problem instances than PGM - the average problem instance in RAVEN requires a solver to correctly identify 6 rules on average between various attributes of the matrix, roughly 3 times the number as in PGM. However, following work revealed that the method of generation for the answer set of RAVEN problem instances was flawed, with some models [25] able to achieve superhuman performance on the benchmark *without using the context matrix*, thus finding a 'shortcut' in the answer set. Figure 3 shows the seriousness of the flaw. This led to the introduction of I-RAVEN and RAVEN-FAIR, which both use a method of answer set generation that prevent AVR models finding shortcuts toward the correct answer, thus avoiding the problem of context-blind models achieving superhuman performance.

2.2.3 AVR models

With regards to computational models on AVR tasks, we focus on deep learning based, end-to-end models, rather than symbolic methods, given the recent success of the deep learning paradigm. In the rest of this paper, we take AVR models to refer to deep learning based models only. AVR models achieve a wide range of performance on AVR tasks, with relatively unspecialised architectures such as models with a CNN encoder followed by LSTM/MLP layers attaining performance accuracies significantly below human performance on the same benchmark [19], whereas state-of-the-art models with specially designed architectures, such as the Scattering Compositional Learner (SCL) [6] and Multi-scale Relation Network (MRNet) [24] consistently outperform human attempts on by a noticeable margin. As of the most recent review of the AVR field [19], the highest scoring model on PGM is MRNet with 98.0% accuracy, and the state-of-the-art on I-RAVEN is a modified SCL [26], achieving 96%. For reference, humans achieve 84% accuracy on I-RAVEN (no human baseline is provided for PGM).

Whilst the modularity and interpretability tools introduced

in Section 2.1 allow one to probe the internal mechanisms of a trained neural network, to the best of our knowledge, the study of the *development* of these internal mechanisms - functional modules in particular - remains relatively neglected. Having a deeper understanding of the way in which stochastic gradient descent moulds the trainable parameters of a neural network would give developers the ability to identify and prevent the development of undesirable components in a neural network before they arise. On a more fundamental level however, such insights would provide a greater understanding of how intelligent behaviour arises in neural systems. We reiterate that abstract visual reasoning problems appear to be an ideal ground to take preliminary steps towards the goal of understanding functional modularity development, given their highly compositional nature and the wealth of previous benchmarks and models to study.

3 METHODS

The aim of this study is to identify groups of parameters in an AVR model that are responsible for specific functionalities (functional modules), and analyse how they develop over the course of training a model. At a high-level, our approach to this is as follows: We use a method introduced in [5], which we refer to as the binary weight masking algorithm, or BWM, that allows one to determine the group of weights in a neural network that are responsible for executing some sub-task. For a range of different sub-tasks, we then apply the BWM algorithm at various stages during the training of SCL, and through comparing differences in these sets of weights we can study how the functional modules develop.

The following section presents our method in further depth. We first cover the binary weight masking algorithm in section 3.1, followed by a discussion of the sub-tasks we find functional modules for in section 3.2. In section 3.3 we describe the main model used in this study - the Scattering Compositional Learner [6] and details of its training. We finish in section 3.4 by outlining a key preprocessing step that we believe all functional modules returned by the BWM must undergo to improve the validity of any further analysis on them.

3.1 Binary weight masking algorithm

To illustrate how the BWM works, consider a neural network M trained on some task T, with trained weights $W_0 = \{w_i\}_{i=1...N}$, where w_i is the ith trained weight and the neural network has N learnable parameters in total. In many cases, it is possible to decompose T into a number of subtasks, $\{t_i\}_{i=1...n}$, where the t_i may be executed either in series or parallel with one another. For example, to solve the main task of finding the missing matrix entry (task T) in the AVR problem instance in Figure 3, numerous subtasks must be executed, such as determining how the outer and inner shape transform across rows, as well as how the inner shapes change colour. These are the sub-tasks t_i that we refer to. In networks with ideal modular structure, there should exist a set of weights W_{t_i} that are a subset of W_0 and are responsible for successfully executing task t_i; **the function of the BWM algorithm is to find this set of weights** W_{t_i}. Before proceeding to how the BWM achieves this, it is worth exploring what is meant for W_{t_i} to be *responsible* for executing task t_i. A particular weight $w_j \in W_0$ cannot simply be said to either contribute or not contribute to a task t_i - the degree to which w_j contributes to the execution of task t_i lies along a continuous spectrum, with some weights being *critically important* to the successful execution of t_i, some being moderately important, and some weights being independent of t_i (zero-ablating them has no effect on the ability of M to perform t_i). To overcome this definitional issue, we can introduce the set of weights $W_{t_i}^f$, with $f \in [0, 1]$, and define it as follows: $W_{t_i}^f$ is the set of weights such that, if all weights **except** $W_{t_i}^f$ are zero-ablated, the performance of the neural network on task t_i reduces by a fraction of $1 - f$ relative to the accuracy that the network was initially achieving on task t_i. To illustrate this, say that a network M was achieving an accuracy of A on task t_i; zero-ablating all weights in M expect the set $W_{t_i}^{0.95}$ would then lead to a new accuracy of $0.95A$ on task t_i. In other words, $W_{t_i}^{0.95}$ is the set of weights that allows the network M to re-attain 95% of it's performance on task t_i. It is this definition we use (or close variations) when referring to a functional module in the following sections; note also that we set $f = 0.95$, meaning that the superscript is usually dropped from $W_{t_i}^f$.

The BWM takes two inputs, the first of which is a trained model M, that we assume has developed a group of weights W_{t_i} responsible for executing task t_i. The second is what we refer to as a *task dataset* - this is a dataset containing problems only for task t_i, of which examples are provided in section 3.2. The BWM algorithm then applies a binary mask to the weights of M, and evaluates the quality of the binary mask over the task dataset, with respect to a joint loss function $\mathcal{L} = \mathcal{L}_{t_i,p} + \alpha\mathcal{L}_{reg}$. $\mathcal{L}_{t_i,p}$ measures the *performance* of the masked neural network on the task dataset (a cross entropy loss is used in this study), and $\alpha\mathcal{L}_{reg}$ is a regularisation term used to control the number of weights included in the mask B, with α a user-defined hyperparameter. The exact form that the regularisation term takes to achieve this is discussed shortly. The binary mask is then iteratively updated to find the mask that optimises $\mathcal{L}$. This corresponds to the BWM removing the weights in M that are not critical for the performance of t_i (by masking them with 0), with the number of weights removed being determined by α. Note that the reason why a *binary* mask is chosen in [5] is so that the application of the mask to network M does not *change the functionality* that M has learned; we are assuming that the initial training process of M has already led to the development of specialised modules for task t_i, and our goal is to *find* the weights that compose these modules, not to change their function.

To update a binary mask (which has a discrete parameter space), via gradient descent, which requires a continuous parameter space, the BWM algorithm trains a logit mask on the weights of the network M, then maps these trained logits to binary values. That is, for the jth weight in W_0, we first associate a logit $l_j \in \mathbb{R}$. Then l_j is *stochastically* mapped to $\{0, 1\}$ with probability p_j^0 and p_j^1 respectively - this is preferred over a deterministic mapping from $l_j \to \{0, 1\}$ as it

allows the optimisation process to better explore the search space of logit masks and reduces the chance of convergence to sub-optimal local minima in the loss landscape. This stochasticity is introduced by sampling from the Gumbel-Softmax distribution [27], [28] for each logit l_j:

$$s_j = \sigma(l_j - \log(\log U_1 / \log U_2)/\tau) \tag{1}$$

where $\sigma(x)$ is the sigmoid function and τ is the temperature, with U_1, U_2 sampled from a uniform distribution, $U_1, U_2 \sim U(0, 1)$. Then, the sample is deterministically mapped to a binary value as follows:

$$b_j = [1_{s_j > 0.5} - s_i]_{stop} + s_i \tag{2}$$

Equation 2 is the straight-through estimator [29], a method used to bypass the issue of gradient flow through threshold functions by setting $db_j/ds_j = 1$. Finally, the original weight w_j is multiplied by the binary value b_j to achieve the binary masking. The loss of the masked network is calculated, **and it is the logits $\mathbf{l_j}$ that are updated via gradient descent.**

The process above entails that the logit l_j can be interpreted as the log probability that a particular weight $w_j \in W_0$ will be included in the mask. The explicit form of the regularisation term $\mathcal{L}_{reg}$ now becomes clear - it is simply a sum over all logits $\mathcal{L}_{reg} = \sum_i^N l_i$ where N is the total number of weights in the model M. Minimising this loss term ensures that the BWM retains as few weights as possible that are required to achieve high accuracy on the task dataset. The logits are initialised with value 2.5, ensuring that there is a high probability that the initialised binary values are 1 - a desirable condition to ensure that the optimisation of the binary mask does not quickly get stuck in a local minima in which $\mathcal{L}_{reg} \approx 0$.

3.1.1 *Choosing the regularisation hyperparameter*

The results of the BWM is critically dependent on the regularisation hyperparameter α. Too large an α will lead to the masking process being overly restrictive with the respect to the number of weights it allows into a functional module, which leads to the masked model being unable to reattain the performance that the unmasked model achieved on the task dataset. Conversely, too small an α means that there will be insufficient optimisation pressure to make the mask sparse, and thus the functional module found for task t_i may contain weights that are not critical for the execution of the task. Due to the importance of α, in our experiments we performed a hyperparameter search over the following five $\log(\alpha)$ values[1]:

$$\log(\alpha) = [-10.0, -6.0, -5.66, -5.33, -5.0] \tag{3}$$

To justify the choice of these values for α, we note that the SCL model used in this study has $\sim 10^5$ parameters (the SCL architecture is detailed further in section 3.3). Therefore, setting $\alpha \approx 1$, will lead to $\mathcal{L}_{reg} \approx 10^5$, given definition of $\mathcal{L}_{reg}$ as the sum of logits in the logit mask. The performance loss term $\mathcal{L}_{p,t_i}$ is of order 10^1, meaning that the two loss terms are of roughly equal magnitude for $\log(\alpha)$ in the range $[-6, -5]$. Setting $\log(\alpha) = -10$ serves as a control hyperparameter - this corresponds to a regularisation term that is 10^{-5} the magnitude of the performance term in the masking loss function, meaning that there is very little optimisation pressure on the BWM to remove weights from the mask. In theory, the functional modules found with $\log(\alpha) = -10$ should achieve performance similar to the unmasked model on a given task dataset. This was not always the case however, as shown in section 4.1 and discussed in section 5. The value of α chosen to proceed with the module analysis, denoted as α^*, is then the largest α that achieves a performance on the task dataset of at least 95% of that achieved when masking with the control hyperparameter $\log(\alpha) = -10$.

1. We often refer to $\log(\alpha)$ rather than α itself, as $\log(\alpha)$ has a simpler numeric form

3.2 Task datasets

We now discuss the tasks that we find functional modules for over the course of training SCL.

Recall the various sub-tasks that are required to solve an AVR problem instance introduced in section 2.2.1, such as perception of entities and their various attributes, as well as the determination of the rules that the entity and layout attributes are transformed by across each row of an AVR problem instance. In this work we focus only on studying functional modules and their development for the perception of a specific attribute of the entities, specifically, the perception of the **entity shape types**. Referring to Table 1, this means that we look only for functional modules responsible for perception of whether an entity is a circle, triangle, square, pentagon, or hexagon; using the notation above, we have five sub-tasks $\{t_i\}_{i=1,2,...5}$. It is important to note that there likely exist functional modules that are responsible for identifying other attributes of entities and layouts (such as entity size, and colour, and layout number and position), and functional modules that are responsible for processing the rules that these attributes are transformed by. Analyses of these modules however, is left for future work.

In order to use the BWM algorithm to identify the weights responsible for some task t_i, we must generate datasets that test the pretrained model's ability on t_i only - these are the task datasets referred to previously. This was succesfully done with only small modifications to the original generation process of I-RAVEN problem instances, the process of which is discussed in [21]. For each sub-task (circle, triangle, square, pentagon and hexagon perception), we generate $7,000$ AVR problem instances that each only contain an entity of a fixed shape type. It is important to note that the task datasets are $1/10$th the size of the original I-RAVEN dataset; this results from the fact that we *constrain* attributes of the entities in the task dataset, meaning that there are fewer possible ways to generate I-RAVEN problem instances. Figures 4 showcases example problems from the square and circle task datasets respectively, with further examples given in Figure 9 of the appendix.

3.3 The Scattering Compositional Learner (SCL)

Following creation of the task dataset for task t_i, the second input to the binary weight masking algorithm is a trained AVR model, from which weights can be identified

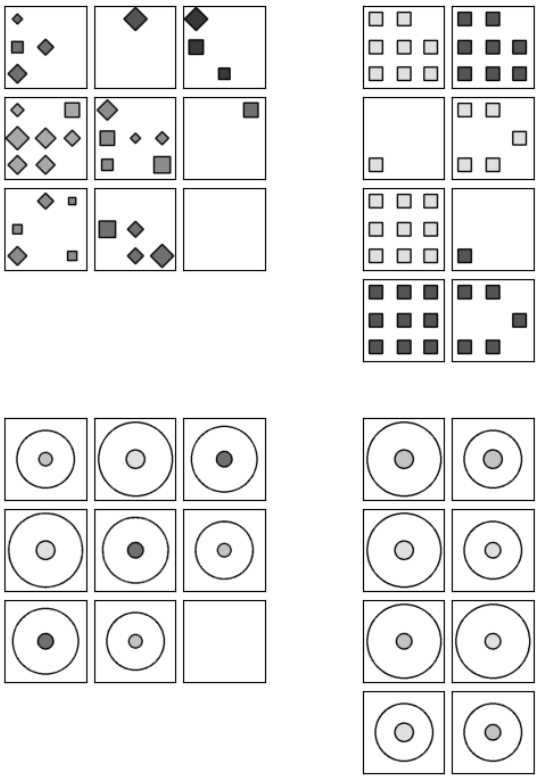

Fig. 4: Problem instances from the squares and circles task datasets. All the entities in the squares dataset have shape type *square*. Thus, when we apply the BWM algorithm, we hope to identify the weights that AVR models develop in order to identify and process squares. The same holds for the circle, and other shape task datasets.

that implement the functionality of t_i. In this paper, we train weight masks on a single AVR model; the Scattering Compositional Learner (SCL) [6] at various stages during it's learning process.

SCL was chosen as the model to analyse in this study for two reasons, the first of which being that it achieves state-of-the-art (SOTA) accuracy on multiple AVR benchmarks as of 2022. On I-RAVEN, SCL achieves accuracies of 95.0%, with the next highest performing models, MRNet and SRAN [23], [24], achieving 86.6% and 73.7% percent respectively. For reference, humans score 84% accuracy on this dataset. In addition to this, comparing the number of parameters of SCL to that of the other strongest performing models shows that SCL has significantly higher parameter efficiency - the number of trainable parameters in SCL is approximately 1.4×10^5, whilst MRNet and SRAN require 1.9×10^7, and 4.4×10^7 parameters to achieve the performance above. Previous work has suggested that a combination of high accuracy and relatively low parameter count suggests the development of specialised functional modules [5], meaning SCL is particularly suited for our study. An important point to note, and a key limitation of our study, is that training weight masks on a single model architecture is insufficient to make strong conclusions about how functional modules develop across *all* AVR models due to the strong influence of explicity built modularity on the learnt modules that develop. Nevertheless, we hope an analysis similar to that in this paper can be undertaken for a wider range of models in future work, which we discuss in further depth in section 5.

To briefly summarise the model architecture, SCL can be decomposed into three main components: object networks, attribute networks and relationship networks, through which solving of the AVR problem takes place sequentially. The ith object network is responsible for extracting representations of the ith object in a panel of an AVR problem instance (entities, layouts), with the jth attribute network then determining the value of the jth attribute (this could be entity colour or type, layout number or something else). The kth relationship network then has the role of working out whether the kth rule holds between the jth attributes of the ith object identified in the panel. Note that $j = 1, 2, ..6$ (6 attributes in Table 1) and $k = 1, ..4$ (4 rules). The architecture of SCL is introduced in great detail in [6].

We trained SCL for 100 epochs on an I-RAVEN dataset containing $70,000$ problem instances, using a batch size of 32 and a learning rate of of 1×10^{-3}, whilst clipping gradients at 0.5. The final model attains an accuracy of 91% on I-RAVEN, just 4% short of the performance achieve in [6]. The key step that allows us to study the development of functional modules in SCL is to save the model weights at various stages during training, specifically, when SCL is attaining $50\%, 70\%$ and 90% accuracy on the I-RAVEN dataset[2]. These points occur after $2, 13$ and 90 epochs respectively, and for notational ease, we refer to the SCL model at these points during training as SCL_{50}, SCL_{70} and SCL_{90}. The BWM is then applied for all 5 task datasets across each of these 3 versions of SCL, yielding 15 weight masks in total.

3.4 Weight mask preprocessing

To conclude this section, we discuss a key processing step in our pipeline that takes place after the BWM finds masks for a particular task-model pair[3], but before any analysis of the masks takes place. To motivate this preprocessing step, we must consider with greater care which weights the BWM algorithm finds during the masking process. Recall that for a particular sub-task t_i and model M, the BWM returns the set of weights that M needs to execute task t_i. Consider an example in which the BWM is applied on some version of SCL and the squares task dataset, returning the set of weights $W_{squares}$. In order to obtain high accuracies on the squares dataset, we claim that $W_{squares}$ can be decomposed into two non-overlapping subsets, that correspond to two different functionalities. One of these subsets are the weights required for the perception of squares in an AVR problem instance - this is the set we are interested in. However, another set of weights will be present in $W_{squares}$ that are responsible for *very basic processing tasks* in an AVR problem instance, such as identifying that there are 8 context and 8 answer panels, or that the bounding box for each of the panels is black. The presence of weights that execute these

2. Random initialisation of the model weights means that SCL begins with 12.5% accuracy, corresponding to random guessing between the 8 answer panels.

3. This refers to a particular combination of a task dataset and model, of which there are 15 in this study. For example, squares-SCL_{90} is a task-model pair, as is pentagons-SCL_{50}.

simple processing tasks are due to the end-to-end nature of deep learning models - recall that the inputs to these models are simply a grid of pixels. We can express this argument more precisely, by writing the weights found by the BWM algorithm, $W_{squares}$, as the union of two sets of weights:

$$W_{squares} = W^{true}_{squares} \cup W_{shared} \tag{4}$$

where $W^{true}_{squares}$ is the set of weights responsible for perception of squares *only*, and W_{shared} is the set of weights that are responsible for the basic processing tasks. Note that both sets of weights are required for SCL to attain high performance on the squares dataset, hence why they are both incorporated into $W_{squares}$.

We refer to the set of weights that execute the basic processing tasks as W_{shared} as this set of weights should be shared across *all* masks (that is, W_{shared} will be present in the mask found for the circles dataset, the triangles dataset etc). Given that we are interested only in the weights that are responsible for the execution of task t_i - $W^{true}_{squares}$ in our example - we therefore want to remove W_{shared} from the masks before our analysis begins. The fact that W_{shared} is present in *any* weight mask makes this process relatively trivial. To remove W_{shared}, we first train a sixth mask[4] on another task dataset, specifically, a task dataset that contains only entities that are white in colour (which we refer to as the 'lights' dataset - see Figure 10(b)). We emphasize that this choice of task dataset is arbitrary, as by definition W_{shared} is present in *all masks*. We then find and remove the weights that are common across all six of these masks, which leaves us with just the true weights required for the execution of the tasks shape perception tasks. One may question the need for training this sixth mask, and ask why instead we did not simply remove the weights shared across all 5 shape masks, $W_{circles}, W_{triangles} \ldots, W_{hexagons}$. This, however, would have removed any weights that are required for the perception of general features of shapes such as corners or edges, which should be kept in the 'true' set of weights.

Following the removal of shared weights in this preprocessing step, the masks found by the BWM are ready for analysis. This is the focus of the next section.

4 Results

We now present the results of applying the BWM algorithm on versions of SCL captured at the $50\%, 70\%$ and 90% points of it's training stage. We trained the binary weight masks on all three versions of SCL for 100 epochs, with a logit initialisation value of 2.5. The task datasets used for training contained $7,000$ problem instances split into batches of size 32; in addition, 10% of the problems in task datasets were withheld to form a validation dataset.

We begin with section 4.1, in which training curves and summary tables of various key metrics obtained during the masking process are presented. The training curves and tables verify that the masking process is mostly working as expected, however there are some peculiarities in the results that we explore further. Following this, in section 4.2, we analyse two key global properties of the discovered weights masks, and their development over SCL training. In section 4.3, we then consider the degree of weight sharing between the shape functional modules, before concluding this section in section 4.4 with an analysis of how functional module weights are distributed throughout the layers of SCL.

4.1 Verification of the BWM algorithm

To illustrate the validity of the mask training process, we begin by plotting smoothed training curves obtained during the masking process for the squares dataset on SCL_{90} in Figure 5. [5]. Learning curves are given for all five values of the regularisation parameter $\log(\alpha)$. Recall that our aim is to find the weights in SCL_{90} that are responsible for perception of squares. Figure 5(a) shows the performance of the masked model on the squares dataset itself, with Figure 5(b) showing the performance on the original I-RAVEN dataset, which of course contain *no constraints* on the shape type of the entities in the AVR problems. Figure 5(c) is simply the final few thousand steps of curve shown in Figure 5(a). Comparing Figure 5(a) and Figure 5(b), we see that the accuracy of the masked model on the task dataset noticeably exceeds the accuracy of the masked model on the original I-RAVEN dataset, for all values of α. This is in line with our expectation - if the masking algorithm is identifying the weights responsible for the squares perception sub-task, whilst removing any weights not required for this sub-task, we expect it to maintain relatively high accuracy on the squares dataset and for performance to suffer across all other tasks. Furthermore, observe the mask performances across the *differing* values of α. Clearly, when $\log(\alpha)$ is equal to -10, the mask achieves it's greatest accuracy on the squares subtask ($\sim 78\%$). Conversely, when $\log(\alpha)$ is set to -5, we see a noticeable drop in accuracy in comparison to when $\log(\alpha)$ is -10, with the $\log(\alpha) = -5$ mask attaining only $\sim 68\%$ on the squares dataset. This further verifies the masking process, because we expect noticeably higher accuracies for low values of α in which BWM has less incentive to remove weights. Similar training curves were obtained across all task-model pairs, and are displayed in section A of the appendix.

The results for *all* task-model pairs are given in Tables 2, 3 and 4. In each, five key data points are given. The first is the accuracy that the model attains on the task dataset *with no masking applied* - this is denoted as A_0. The second is the accuracy that the masked model obtains with the control hyperparameter $\log(\alpha)$ set to -10, which we give as A_{-10}. In theory, A_{-10} should be roughly equal to A_0, but this is not always the case, especially for SCL_{90} and SCL_{70}. Next, we give $\log(\alpha^*)$, which is the value of α used in subsequent analyses of the masks; we remind the reader that α^* is defined as the greatest value of α that achieves a masked accuracy within approximately 5% of A_{-10}. The reason for this definition is perhaps clearer now - ideally, we would have defined α^* as the greatest α that attained accuracies within 5% of A_0, but given that A_0 and A_{-10} differed by more than 5% for SCL_{70} and SCL_{90}, we opted for the this alternate definition. Finally, the accuracy of the model masked with hyperparameter α^* on **both** the task dataset

4. in addition to the 5 masks found for a particular SCL version

5. The choice of the squares-SCL_{90} pair is arbitrary, with similar training curves obtained for all other task model pairs.

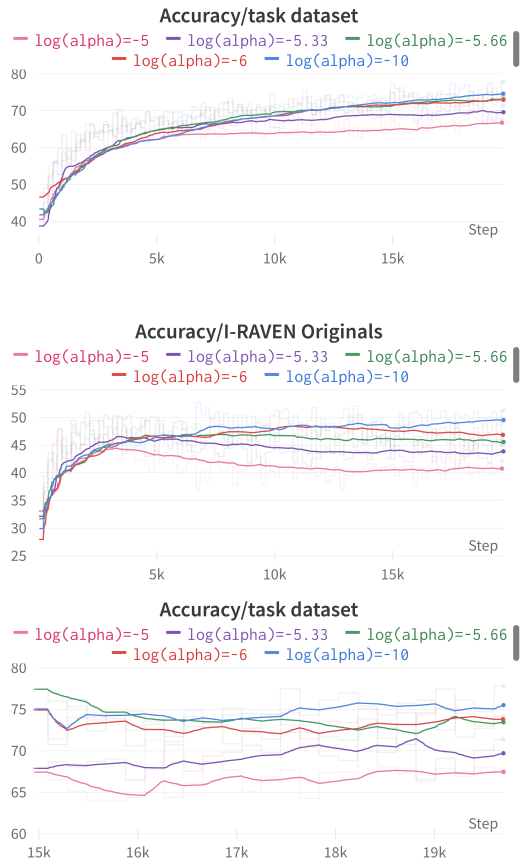

Fig. 5: Learning curves for SCL_{90} (note that the y-axis range on each plot differs).

and the original I-RAVEN dataset are provided, denoted as A_{α^*} and $A^O_{\alpha^*}$ respectively.

	A_0	A_{-10}	$\log(\alpha^*)$	A_{α^*}	$A^O_{\alpha^*}$
Circles	89.0	80.0	-5.66	76.0	44.4
Triangles	84.0	71.0	-5.33	71.0	48.0
Squares	92.0	78.0	-6.0	74.0	47.0
Pentagons	90.0	78.0	-6.0	75.0	48.0
Hexagons	92.0	77.0	-5.33	75.0	44.0

TABLE 2: Masking results for SCL_{90}

	A_0	A_{-10}	$\log(\alpha^*)$	A_{α^*}	$A^O_{\alpha^*}$
Circles	81.0	77.0	-5.33	73.0	44.0
Triangles	76.0	74.0	-5.66	70.0	43.0
Squares	80.0	77.0	-5.66	75.0	47.0
Pentagons	87.0	78.0	-5.66	75.0	48.0
Hexagons	85.0	77.0	-5.66	75.0	46.0

TABLE 3: Masking results for SCL_{70}

	A_0	A_{-10}	$\log(\alpha^*)$	A_{α^*}	$A^O_{\alpha^*}$
Circles	60.0	65.0	-5.33	62.0	35.0
Triangles	56.0	62.0	-5.33	59.0	33.0
Squares	63.0	65.0	-5.66	63.0	39.0
Pentagons	68.0	66.0	-5.66	63.0	38.0
Hexagons	63.0	65.0	-5.66	62.0	38.0

TABLE 4: Masking results for SCL_{50}

The tables show the validity of the masking process for the other 14 task-model pairs. We see that the masked models achieve lower accuracies on SCL_{50} (Table 4) than on the higher accuracy models, by comparing A_{α^*} columns across all three versions of SCL. In addition to this, the accuracies of the masked model on the task datasets (A_{α^*}) clearly exceed the accuracies of the masked model on the original I-RAVEN dataset, $A^O_{\alpha^*}$, as was the case for the learning curves.

However, some of the values in the tables above require further explanation. First consider the accuracies in the A_0 column. For SCL_{70} and SCL_{50}, we see that the performance of the unmasked model on the task datasets exceeds that of the purported accuracy of the model. For example SCL_{70} achieves an accuracy of 87.0% on the pentagon sub-task, with SCL_{50} attaining 68.0%; we initially expected that these the two models would achieve roughly 70% and 50% respectively. Upon further inspection, this result is not particularly surprising and is due to the variation in accuracy of the pre-masked model across all sub-tasks. A particular model is likely to perform stronger on some of sub-tasks than others, yet it's final accuracy will be the mean accuracy it attains across all of these sub-tasks. To check this, we evaluated SCL_{70} on the lights dataset (Figure 10(b)) and found it's accuracy to be just 63%. If SCL_{70} achieves similar accuracies on other sub-tasks, it is clear to see how this will lead to an average accuracy of 70%.
A more surprising result can be seen when comparing the columns of A_0 and A_{-10} for SCL_{90} and SCL_{70}. Recall that A_{-10} is the accuracy that the masking process achieves when setting the logarithm of the regularisation hyperparameter $\log(\alpha)$ to -10. Given that this hyperparameter choice leads to a regularisation loss term 10^{-5} times smaller than the size of the performance loss term, we should expect A_{-10} to be roughly equal to A_0. That is, in the case of using the control hyperparameter, we should expect the masking process to reattain the accuracy of the unmasked model. This is clearly not the case in Tables 2 and 3, with A_{-10} over 10 percent less than A_0 in some cases (e.g. squares, for SCL_{90}). A possible reason for this is an insufficient number of training epochs for the masking process, which seems supported by the fact that the training curve in Figure 5(a) has not fully converged after 100 epochs. To test this, we re-ran the masking process for 200 epochs on the squares task dataset for SCL_{90} (using the control hyperparameter), but an additional 100 epochs training only increased the mask performance by 4%. It thus appears as if the rate of convergence for the masking process significantly slows for higher task dataset accuracies. One potential explanation for this is the relatively small size of the task datasets used during the masking process - compared to the $70,000$ problem instances that SCL was trained upon, the task datasets had only $7,000$ problem instances. This is explored in further depth in section 5.

What are the implications of the divergence of A_{-10} and A_0 on our study of functional modules development? It appears as if they limit our study quite significantly. The tables show that SCL_{50} achieves average accuracies of $\sim$

60% on the shape task datasets[6], whereas SCL_{70} and SCL_{90} average roughly 75%. As a result, we are only tracking functional module development over a change in accuracy of approximately 15%, rather than the 40% that was initially expected from the study of SCL_{50} through to SCL_{90}, and we ask the reader to bear this in mind when considering the results of the following section. Methods to tackle this limitation are detailed in section 5.

4.2 Sparsity and Localisation of modules

To begin an analysis of how the shape functional modules develop over the course of model training, we consider two key global properties of the modules; their *sparsity* and *dispersity*. For a mask W_{t_i}, we define it's sparsity, s_i as:

$$s_i = 1 - \frac{n(W_{t_i})}{n(W_{model})} \tag{5}$$

where we have introduced W_{model} as the set of weights of the whole model, and $n(W_{t_i}), n(W_{model})$ as the number of weights in the functional module W_{t_i} and the whole network respectively. This sparsity metric captures the degree to which the model distributes the execution of sub-task t_i amongst weights in the network. A sparsity of approximately 1 indicates that practically all weights in a network are required for succesful execution of t_i, whereas a sparsity much less than 1 means that a very small number of weights are needed for t_i.

We are also interested in the representing the degree to which the module weights are spatially localised in the network. Our motivation here is to find a metric that is correlated with how strongly SCL concentrates weights for a task t_i in particular regions of the network. To illustrate, first recall that the SCL model used in this study has 44 layers. Then, if all the weights required to execute some task t_j lie in just 4 of these layers, whereas the weights required to execute another task t_k lie in 8 layers, we'd like d_k to be twice the value of d_j, to capture the fact that the module for task t_k is twice as distributed across the model as that for task t_j.
We now define a simple metric that has this property. Consider the weights of a functional module W_{t_i} that are are in the jth layer of the network - we denote this as $W^j_{t_i}$, where clearly $W^j_{t_i} \subseteq W_{t_i}$[7]. For a particular weight $w \in W^j_{t_i}$, we let $l(w)$ represent the position of the weight in the layer. This is best conceptualised by viewing all the weights in a particular layer as a tensor, i.e. if w is the weight connecting the *first* neuron in layer $L-1$ of some multi-layer perceptron (MLP) to the *second* neuron in layer L, $l(w)$ can be represented as the column vector $(1, 2)^T$. With this in mind, we can then represent the (normed) spatial variance of weights in a layer j as:

$$\widehat{\sigma^2_j} = \frac{1}{nD_j} \sum_w \frac{(l(w) - \bar{l}_j)^2}{\bar{l}_j^2} \tag{6}$$

where we sum over all $w \in W^j_{t_i}$. $\bar{l}_j$ is the average position of the weights in layer j[8] and D_j is the dimension of the tensor - these serve as normalisation constants. $\widehat{\sigma^2_j}$ gives the degree of spreading of weights in the layer j. If all the $w \in W^j_{t_i}$ are concentrated in a certain region of layer j, $\widehat{\sigma^2_j}$ will be relatively small, with the converse holding if the weights are well distributed throughout the layer. Then, to quantify the degree of weight spreading across the whole model, we simply average over all N layers of SCL to arrive at our metric d_i:

$$d_i = \frac{1}{N} \sum_j^N \widehat{\sigma^2_j} \tag{7}$$

This is the metric that we refer to as *dispersity*. . We emphasise that the dispersity as defined here is only a simple proxy to capture the dispersion of module weights across SCL, and there is significant room to refine it. Avenues for improvement are briefly explored in 5.
We plot sparsity and dispersity for all shape modules across the three model accuracies in Figure 6.

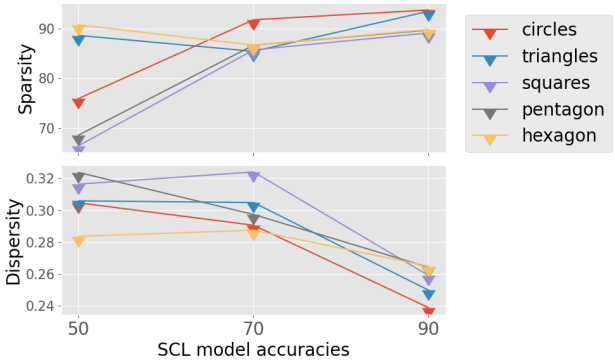

Fig. 6: Line plots for sparsity and dispersity metrics. **Note the different scales on the y-axes**, although we emphasise that dispersity values have no clear interpretation.

It is interesting to see that the modules found by the weight masking algorithm contain at most 30% of the weights of the whole model (a sparsity of 70%), and progress to a sparsity of approximately 90% for SCL_{70} and SCL_{90}. This corroborates the theory that neural networks do develop specialisations in their weights for particular functionalities, as SCL_{70} and SCL_{90} are able to discard 90% of their trained weights and still attain high performance accuracies on the task datasets. From the sparsity figure, we also see that the triangle and hexagon modules maintain a sparsity of roughly 90% over all three versions of SCL, whereas the sparsity of the circle, square and pentagon modules increase noticeably from SCL_{50} to SCL_{70}. This could potentially be explained by the fact that the triangle and hexagon sub-modules fundamentally require fewer weights to execute their respective sub-tasks than the other shape modules, though we did not carry out further experiments to probe this.
The dispersity plot indicates that over the course of training, functional modules for all shapes become increasingly spatially concentrated in the model layers by a factor of about 25%. Again, this is in line with our expectations. As functional modules become increasingly sparse, it is also likely that the weights that comprise the functional modules

6. Using regularisation hyperparameter α^*
7. The equality holds if all of the weights for task t_i are in layer j.
8. $\bar{l}_j = \mathbb{E}[l(w)]$ for $w \in W^j_{t_i}$

will group together in particular areas of the network, as certain layers, and regions of these layers become increasingly specialised. This may be related to the phenomenon of branch specialisation discovered in [11]. As is the case for the rest of the functional module analyses below, one could re-run our experiments for functional modules found with other task datasets to verify these trends.

4.3 Weight Sharing between Modules

In the next stage of our analysis, we concern ourselves with the degree of *sharing* across the functional modules. Here, we look to study whether SCL learns to group certain functionalities together in modules. To illustrate, one might expect that the triangle and hexagon modules share a relatively large fraction of weights, given that there appear to be similar functions involved in triangle perception and hexagon perception; a hexagon can simply be constructed from six equilateral triangles. We quantify the degree of sharing between two modules, W_{t_j} and W_{t_k} through the metric S_{jk}, where:

$$S_{jk} = \frac{n(W_{t_j} \cap W_{t_k})}{\min(n(W_{t_j}), n(W_{t_k}))} \tag{8}$$

Once again $n(W_{t_i})$ is the number of weights in the functional module identified by the BWM for task t_i. Here, $n(W_{t_j} \cap W_{t_k})$ is the number of weights that are shared by W_{t_j} and W_{t_k}. Notice that $S_{jk} = 1$ if the weights of one module are a subset of the weights of another, and 0 if no weights are shared. S_{jk} is plotted for all shape functional modules in Figure 7, once again for all three versions of SCL.

The first noticeable result from the sharing matrices is that the shape functional modules found by the BWM algorithm share over 50% of weights for all pairings, apart from a single pair in the SCL_{70} figure (7(b)). This is especially intriguing for SCL_{90}, the reason for this being that the modules found for SCL_{90} contain only 10% of the weights of the whole network (this is shown in the sparsity plot of Figure 6). Thus, given that we see sharing fractions of up to 70% for SCL_{90}, it appears as if SCL learns to group significant fractions of weights between modules that execute similar tasks.

We also notice that each SCL version appears to have noticeably distinct degrees of sharing across the shape modules. The average degree of sharing for SCL_{50} is 57.3%, and noticeably lower for SCL_{70}, at 53.6%. SCL90 has a higher average sharing degree (63.4%), with some modules sharing up to 72.5% of their weights with another (see the hexagon-pentagon entry). However, the noticeable difference in weight sharing fractions seen for SCL_{90} and SCL_{70} modules is at odds with our expectations. In section 4.1, we found that the masking process achieves similar accuracies for both of these models (see A_{α^*} in the results tables). In addition, the spatial distribution plots to be introduced in section 4.4 and Figure 8 indicate that the functional modules found for SCL_{70} and SCL_{90} are similarly distributed through the layers of the model. Yet, there are clear differences in the sharing fractions between two task datasets, with the hexagon-pentagon pair perhaps being the clearest illustration of this. For SCL_{70}, it has a sharing

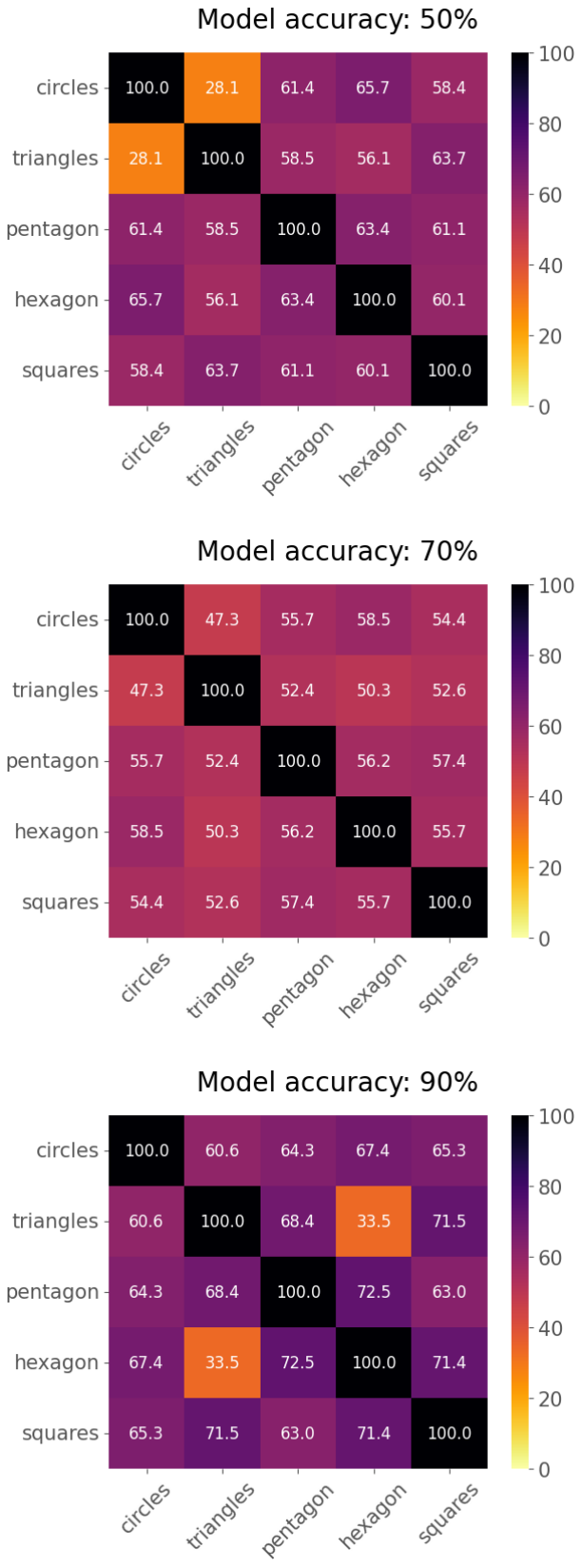

Fig. 7: Weight sharing matrices for all three SCL model accuracies. The average sharing degrees across the SCL models are $57.3\%, 53.6\%$ and 63.4% respectively. These matrices are symmetric, by definition.

fraction of 63.4%, whereas it is 77.5% for SCL_{90}. At present, we are unsure of the origin of this peculiar result, and would be interested in seeing further work that looks to resolve this.

4.4 Spatial Distribution of Modules

Our final line of analysis is motivated by an interest in how the functional modules are distributed over the individual layers of SCL. This is in contrast to the dispersity metric of section 6, which aims to study the distribution of module weights over the layers of SCL as a *scalar* quantity. For a functional module W_{t_i}, we can use our previous notation to define $n_j = n(W^j_{t_i}$ as the number of weights of the module that are in layer j of the model. The fraction $f_j = n_j/n(W_{t_i})$ is then the fraction of weights in W_{t_i} that are in layer j, which we plot the logarithm of in Figure 8. To obtain Figure 8, we also average f_j over all shape sub-modules. We can thus interpret the module being studied here as a functional module that is responsible for general shape perception. Our goal is to examine the distribution of module weights over SCL's layers during the course of training.

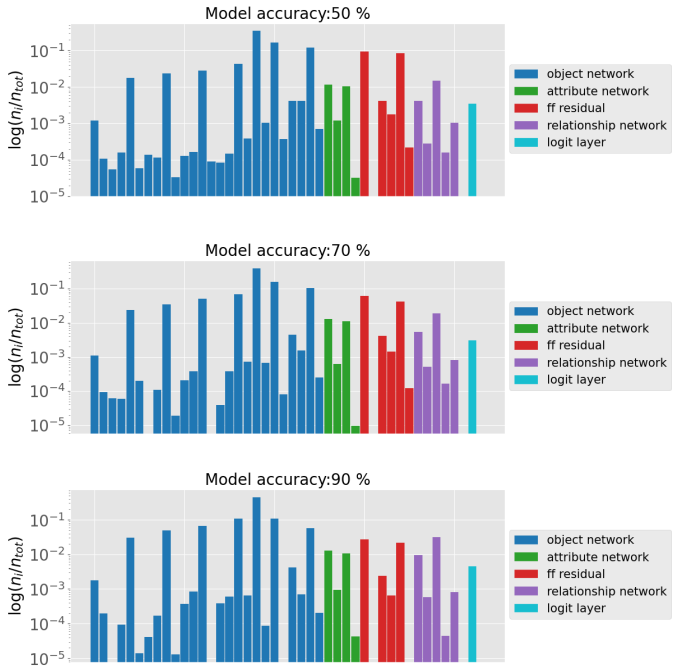

Fig. 8: Spatial distribution plots for all three SCL accuracies, where we plot the logarithm of f_j averaged across shape datasets. Referring to the y-axis label, n_{tot} corresponds to $n(W(t_i))$. ff residual in the legend corresponds to a feed forward residual layer.

Comparison of the three figures shows there does not appear to be any large redistribution of weights across the three versions of SCL. To illustrate, observe the two highest bars in Figure 8(a), which contain over 20% of the weights in this general shape functional module - these two layers also contribute significantly in Figures 8(b) and 8(c), with only a weight small redistribution. Coupled with the results of the sparsity plot in Figure 6, this suggests an interesting dynamic taking place. In progressing from SCL_{50} to SCL_{90}, the model training process cuts away weights from the shape functional modules (thereby increasing sparsity), but does so in a way to preserve the spatial distribution of the weights.

Although there appears to be no large scale redistribution of weights in the plots above, we do see a slightly increasing concentration of weights in layers as one progresses from SCL_{50} to SCL_{90}. In the SCL_{50} plot, there are just two layers with *no* weights in the general shape module (the 2nd feed forward residual layer, and the first logit layer). This however, is more common in the SCL_{90} plot, with 5 layers featuring no weights. This rough trend of reduced weight dispersion throughout the network is also captured in Figure 6 of section 4.2.

5 Evaluation

In the penultimate section of this paper we discuss limitations of this work as well as avenues for further work to address these.

We initially set out to study functional module dynamics as training progressed from 50% to 90% accuracy, or over an 'accuracy window' of 40%. However, we later found that SCL_{90} and SCL_{70} were unable to re-attain their pre-masked accuracies, even when using the control regularisation hyperparameter (Tables 2, 3), and that SCL_{50} was attaining accuracies of roughly $60-65\%$ on the shape task datasets. Together, these two factors meant that the true size of the accuracy window we studied functional modularity over was roughly 15%, instead of the 40% initially intended. This is the most severe limitation of our study as we capture only a relatively small time frame during SCL's development, which could explain the similarities we observe in the spatial distributions of the functional modules that we see in Figure 8. Thus first issue for future work to resolve is to ensure that the masking process is able to re-attain the pre-masked accuracies of SCL_{70} and SCL_{90} over a reasonable number of training iterations. Given that an additional one hundred epochs of training on SCL_{90} improved the masking accuracies by only 4%, we believe that expanding the size of the task datasets would be most effective here. This could potentially be done through incorporating problem instances from other AVR datasets such as PGM [22], though these would have to be modified to fit the constraints of the task dataset in question. Increasing the number of allowed attribute values may be another way to increase task dataset size. However, a simpler approach to increasing the size of the accuracy window over which functional module development is studied is to apply the BWM algorithm on versions of SCL with even lower accuracies. For example, a similar analysis could also be carried out for an SCL_{30}, or for a control version of SCL that has yet to undergo any training (this would be an $\text{SCL}_{12.5}$).

Another limitation of this study is the fact that we have studied only a single AVR model, namely the Scattering Compositional Learner (SCL). This limits our ability to make conclusions about the nature of functional modularity development in AVR networks as we expect that the *explicity-built* modularity of SCL influences it's *learnt* modularity to a significant degree. For example, it seems highly likely that the presence of the object networks in SCL

will affect how any modular structures responsible for object perception arise during training; a network without these object networks may develop perception modules that vary with respect to some important module parameter, such as sparsity or dispersity. Thus, following work should look to carry out an analysis like this one on other successful AVR models, such as MRNet [24]. Networks like MRNet were not considered in this study due to their parameter size, with MRNet in particular having 10^2 the number of parameters as SCL. However, if the BWM is succesfully applied to these models, one could then seek to identify any similarities in functional module development across all AVR models which may give deeper insight into how deep learning systems learn to perform abstract visual reasoning. Finally, there is significant scope to extend our analysis framework of the functional modules. For example, whilst the dispersity metric d_i that we define captures a rough notion of the degree to which module weights are dispersed through a network, there are many ways for it to be enhanced. To illustrate, consider the following example of two functional modules; one with all of it's weights in the first four layers of SCL, and another with it's weights in 6th, 11th, 25th and 39th layers respectively (these layer numbers are arbitrary). All else equal, our dispersity metric would assign equal dispersity values to both of these modules, although it is clear that the second module is far more dispersed throughout the network. Methods to incorporate some measure of relative distance between layers that house the weights of the functional module into d_i would thus be valuable. Furthermore, this metric is not well suited to networks with a number of branches executing computations in parallel[9]. Enchanced versions of d_i should account for this property of neural networks also. We believe that there are many other potential paths to derive insight from the functional modules found by the BWM, of which we have only scratched the surface with our analysis. One interesting line of work may be to determine how the network builds pipelines to solve a complex task. For example, in order to solve the AVR problem of Figure 4(b), a network must first perceive the existence of circles and their various attributes (call this task t_1). Following this, it must then work out the corresponding transformation rules, after which it must select the correct answer from the answer set - let these be task t_2 and t_3 respectively. Assuming we can find functional modules for task t_1, t_2 and t_3, can we then study the pipeline that the network builds to tackle the whole problem? In other words, how are these individual function modules be *linked together* to execute the main task that the network is trained upon?

6 Conclusion

In this work, we presented a framework for studying the development of functional modules in neural networks over the course of training. We analysed the functional modules that arise in a particular class of neural networks - abstract visual reasoning models - due to the highly compositional nature of these tasks. We focused on a state-of-the-art AVR model, SCL, to conduct our studies, given it's impressive performance and high parameter efficiency; we then found functional modules in SCL by applying a binary weight masking algorithm (BWM) that finds the weights required by the network to implement particular sub-tasks. The sub-tasks in question were that of shape perception, although we emphasize that our framework allows a vast range of sub-tasks to be studied.

Our results show that the BWM is able to locate specialised weights for the execution of shape perception in SCL, finding modules that comprise only 10% of the weights in SCL. We see that the sparsity of these functional modules increases over training, as well as the degree to which the weights composing the modules are spatially localised. Moreso, a study of the the degree of weight sharing between these functional modules reveals that SCL learns to share a significant fraction of weights between shape functional modules.

The limitations of our work provide clear direction for future studies. A clear first step would be to enlarge the size of the task datasets, as we hypothesise that small task datasets lead to the BWM being unable to converge to the pre-masked model accuracies in the later stages of mask training. This in turn would increase the size of the accuracy window over which to study functional module development, which was limited to only 15% in this work. Following this, we would be interested in seeing the framework applied first to other AVR models, and then to a broader class of neural networks such as Large Language Models (LLMs) [30] that have displayed impressive general capabilities in recent years. In the long term, we hope that such work is able to shine light on the inner workings of deep learning, and disperse the air of mystery that currently surrounds their internal mechanisms.

Acknowledgments - Thank you to Dr. Shauna Concannon for guidance on this project throughout the year, and for interesting and thought-provoking discussions each week.

9. Briefly, this is because in arriving at d_i, we envision a neural network solely as a sequence of weight tensors. This is mostly true for SCL, in which data follows an object network, then attribute network, then relationship network pipeline sequentially. Of course, the same cannot be said for other AVR models.

References

[1] T. Räuker, A. Ho, S. Casper, and D. Hadfield-Menell, "Toward transparent ai: A survey on interpreting the inner structures of deep neural networks," 2023.

[2] Y. Zhang, P. Tiňo, A. Leonardis, and K. Tang, "A survey on neural network interpretability," *CoRR*, vol. abs/2012.14261, 2020. [Online]. Available: https://arxiv.org/abs/2012.14261

[3] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–503, 2016. [Online]. Available: http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html

[4] M. Amer, "Phd thesis: Modularity in artificial neural networks," 2021.

[5] R. Csordás, S. van Steenkiste, and J. Schmidhuber, "Are neural nets modular? inspecting functional modularity through differentiable weight masks," 2020. [Online]. Available: https://arxiv.org/abs/2010.02066

[6] Y. Wu, H. Dong, R. Grosse, and J. Ba, "The scattering compositional learner: Discovering objects, attributes, relationships in analogical reasoning," 2020.

[7] D. Filan, S. Casper, S. Hod, C. Wild, A. Critch, and S. Russell, "Clusterability in neural networks," 2021. [Online]. Available: https://arxiv.org/abs/2103.03386
[8] S. Hod, D. Filan, S. Casper, A. Critch, and S. Russell, "Quantifying local specialization in deep neural networks," 2021. [Online]. Available: https://arxiv.org/abs/2110.08058
[9] C. Olah, A. Mordvintsev, and L. Schubert, "Feature visualization," *Distill*, 2017, https://distill.pub/2017/feature-visualization.
[10] C. Olah, N. Cammarata, L. Schubert, G. Goh, M. Petrov, and S. Carter, "Zoom in: An introduction to circuits," *Distill*, 2020, https://distill.pub/2020/circuits/zoom-in.
[11] C. Voss, G. Goh, N. Cammarata, M. Petrov, L. Schubert, and C. Olah, "Branch specialization," *Distill*, 2021, https://distill.pub/2020/circuits/branch-specialization.
[12] M. Geva, R. Schuster, J. Berant, and O. Levy, "Transformer feed-forward layers are key-value memories," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 5484–5495. [Online]. Available: https://aclanthology.org/2021.emnlp-main.446
[13] Y. Krishnamurthy and C. Watkins, "Interpretability in gated modular neural networks," in *eXplainable AI approaches for debugging and diagnosis.*, 2021. [Online]. Available: https://openreview.net/forum?id=VMwAfi-c92r
[14] L. Kirsch, J. Kunze, and D. Barber, "Modular networks: Learning to decompose neural computation," *Advances in neural information processing systems*, vol. 31, 2018.
[15] D. Bahdanau, S. Murty, M. Noukhovitch, T. H. Nguyen, H. de Vries, and A. Courville, "Systematic generalization: What is required and can it be learned?" *arXiv preprint arXiv:1811.12889*, 2018.
[16] R. Hu, J. Andreas, M. Rohrbach, T. Darrell, and K. Saenko, "Learning to reason: End-to-end module networks for visual question answering," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 804–813.
[17] R. Hu, J. Andreas, T. Darrell, and K. Saenko, "Explainable neural computation via stack neural module networks," in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 53–69.
[18] V. Mavi, A. Jangra, and A. Jatowt, "A survey on multi-hop question answering and generation," 2022. [Online]. Available: https://arxiv.org/abs/2204.09140
[19] M. Malki'nski and J. Ma'ndziuk, "Deep learning methods for abstract visual reasoning: A survey on raven's progressive matrices," *ArXiv*, vol. abs/2201.12382, 2022.
[20] C. Zhang, F. Gao, B. Jia, Y. Zhu, and S.-C. Zhu. (2019) Supplementary material for raven: A dataset for relational and analogical visual reasoning. [Online]. Available: https://wellyzhang.github.io/attach/cvpr19zhang_supp.pdf
[21] ——, "Raven: A dataset for relational and analogical visual reasoning," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 5317–5327.
[22] D. G. T. Barrett, F. Hill, A. Santoro, A. S. Morcos, and T. Lillicrap, "Measuring abstract reasoning in neural networks," 2018.
[23] S. Hu, Y. Ma, X. Liu, Y. Wei, and S. Bai, "Stratified rule-aware network for abstract visual reasoning," in *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, vol. 35, no. 2, 2021, pp. 1567–1574.
[24] Y. Benny, N. Pekar, and L. Wolf, "Scale-localized abstract reasoning," 2021.
[25] C. Zhang, B. Jia, F. Gao, Y. Zhu, H. Lu, and S.-C. Zhu, "Learning perceptual inference by contrasting," in *Advances in Neural Information Processing Systems*, 2019.
[26] M. Malkinski and J. Mandziuk, "Multi-label contrastive learning for abstract visual reasoning," *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–13, 2022. [Online]. Available: https://doi.org/10.1109%2Ftnnls.2022.3185949
[27] E. Jang, S. Gu, and B. Poole, "Categorical reparameterization with gumbel-softmax," 2017.
[28] C. J. Maddison, A. Mnih, and Y. W. Teh, "The concrete distribution: A continuous relaxation of discrete random variables," in *International Conference on Learning Representations*, 2017. [Online]. Available: https://openreview.net/forum?id=S1jE5L5gl
[29] Y. Bengio, N. Léonard, and A. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation," 2013.
[30] OpenAI, "Gpt-4 technical report," 2023.

APPENDIX

More Task Specific Dataset Examples

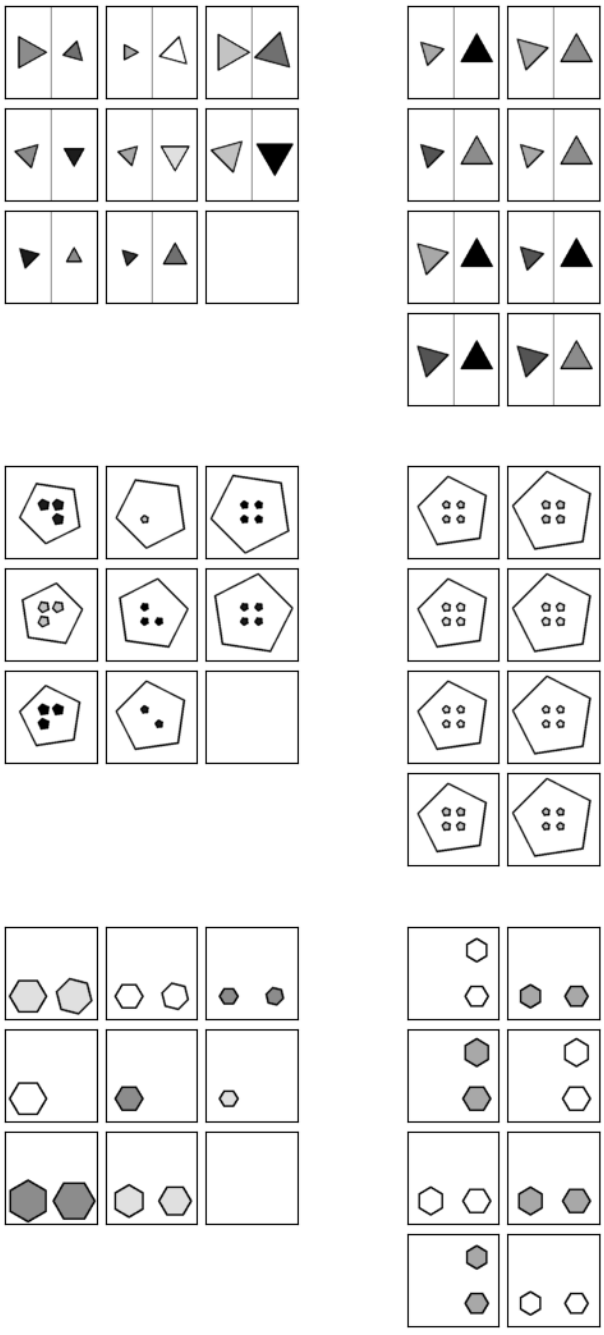

Fig. 9: Examples from the triangles, pentagons and hexagons task datasets.

More AVR problems

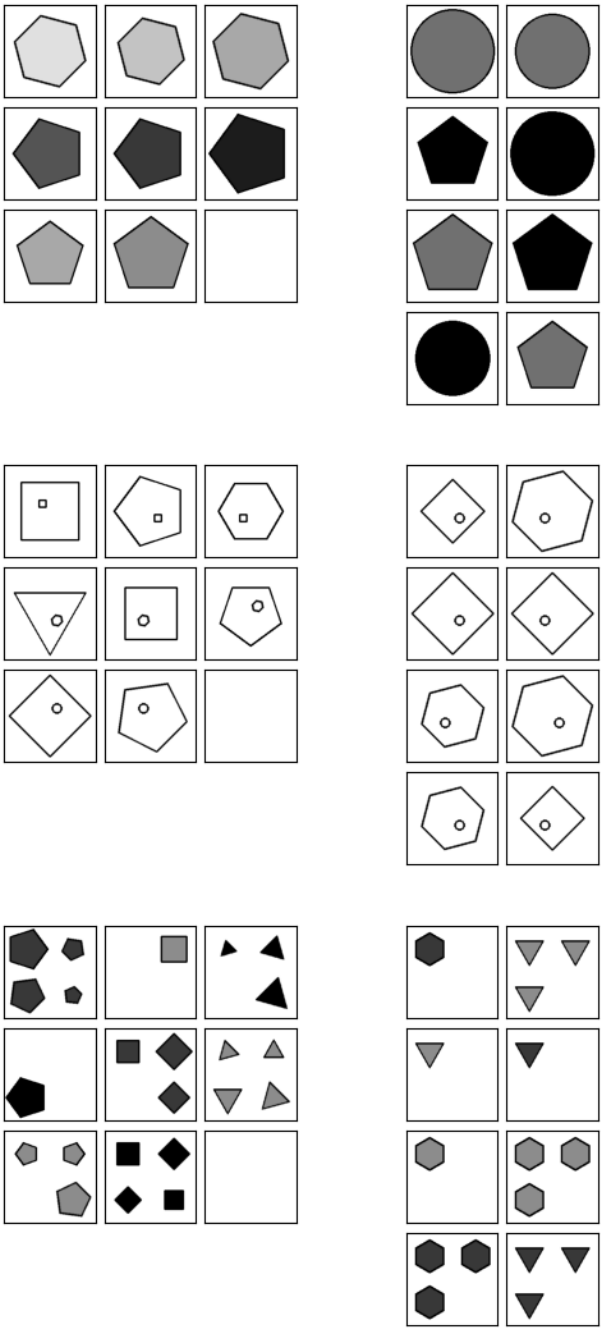

Fig. 10: Further examples of AVR probem instances. Figure 10(a) gives an example of the centre-layout, discussed in section 2.2.1
. It is also taken from a task dataset that contains only AVR problem instances with very large entity sizes (A.K.A the 'big' task dataset). Figure 10(b) is an example from the dataset containing only white shapes, often referred to as the 'lights' datasets. It was this dataset used in the preprocessing step of section 3.4.

Training Curves

Learning curves for selected task-model pairs.

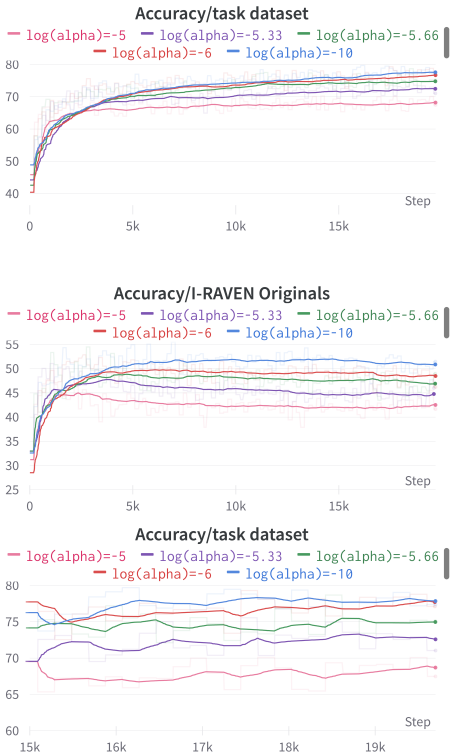

Fig. 11: Learning curves for SCL_{70} on the squares task dataset.

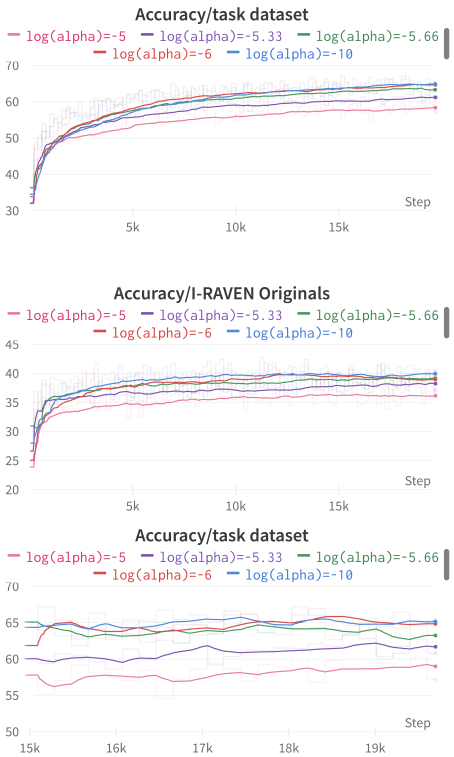

Fig. 12: Learning curves for SCL_{50} on the squares task dataset.

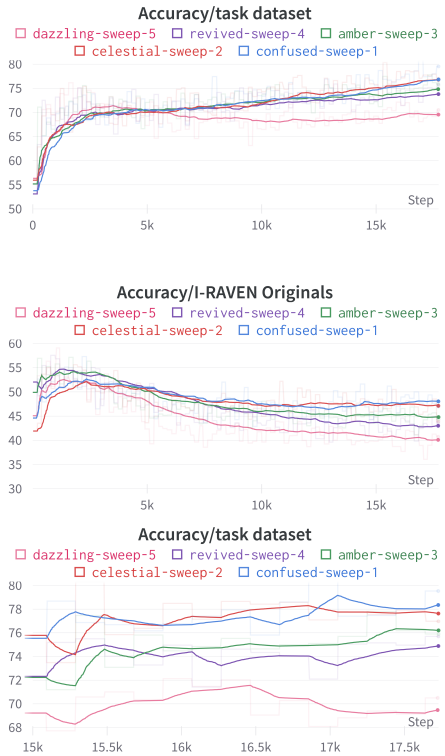

Fig. 13: Learning curves for SCL_{90} on the circles task dataset.

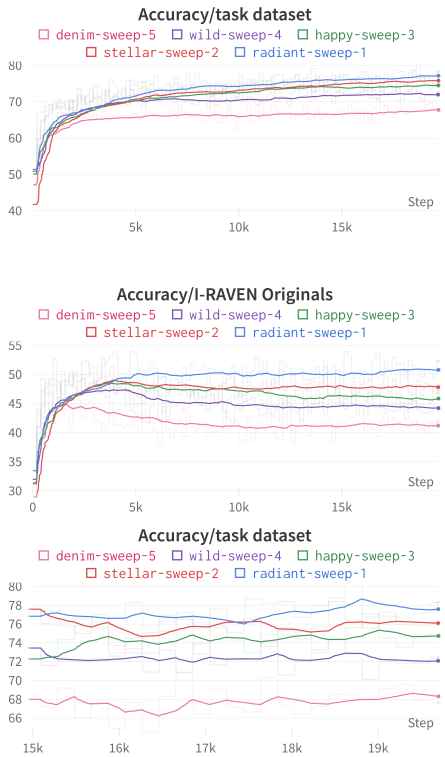

Fig. 14: Learning curves for SCL_{70} on the hexagons task dataset.