Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract Reasoning in Neural Networks

Bachelor Thesis

Ege Karaismailoğlu

September 14, 2020

Supervisor: Prof. Dr. Joachim Buhmann

Advisors: Ivan Ovinnikov, Eugene Bykovets

Department of Computer Science, ETH Zürich

Abstract

We investigate the task of generating solutions to a subclass of visual psychometric test questions under some simplifying conditions. We model the generation task as a two-step process of understanding the rules that govern the visual inputs, and using this understanding to generate an answer to the posed problem. In the first step, our proposed models attain over 91% accuracy. In the second step, our models achieve over 25% generative accuracy under a lightly relaxed accuracy measure. Our work offers insights into how capable neural networks are at reasoning about their inputs.

Contents

Chapter 1

Introduction

In the past decade, the greatest highlight of machine learning has been the rise of deep learning. With the advent of big data and the increases in computational power, deep learning algorithms have achieved superhuman performance on a myriad of tasks, such as image classification [1], playing the game of Go [2] and certain natural language understanding tasks [3].

At first, these successes may give a false sense of what such algorithms are actually doing. We may be inclined to think that these methods "understand" natural images, language, and even the game of Go in the way that humans do. However, it is well understood that deep learning has its flaws. Sample inefficiency, out-of-distribution generalization and adversarial examples are some of the major obstacles which collectively draw a thick line between the current state of deep supervised learning and our understanding of human learning.

One example that illustrates some of these deficits is where a deep neural network is trained to learn the identity function on real scalars, where its training set consists of a dense sampling of a fixed interval, say $[-5, 5]$. Such a network will fail to extrapolate to values outside the learned interval [4].

Given these seemingly contradictory successes and failures attributed to such learning methods, one question naturally arises: How intelligent is deep learning?

The answer to this question is not clear, and has been a topic of debate. We next give two attempts from the research community to gain insights into this question.

In his paper [5], Francois Chollet identifies the ability to generalize as a key indicator of intelligence. Correspondingly, the Abstract Reasoning Corpus (ARC) benchmark proposed in that paper attempts to measure how well systems generalize to new kinds of tasks. The corpus consists of 1000 hand-

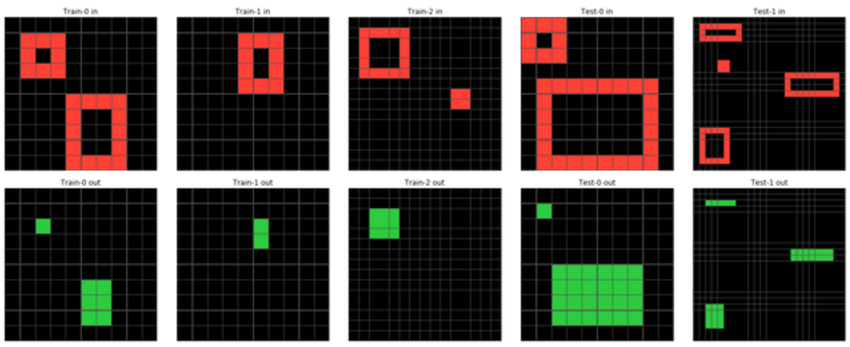

Figure 1.1: An example task from the ARC benchmark. The task consists of three training samples and two test samples. By looking at the input-output pairs, the test-taker is expected to understand that the output is computed by painting the insides of the red rectangles to green.

crafted tasks, such as the one depicted in figure 1.1. In every task, the output depends on the input in a new, unique way, and the test-taker is expected to recover this relationship from between 3 and 8 training samples, depending on the task. For each task, the test-taker is asked to generate the outputs that best complete the test inputs. Chollet notes that the task is currently not approachable by deep learning techniques due to the low number of samples per task and the unique data distribution for each type of task.

Another way of approaching this question is by investigating the abstract reasoning abilities of neural networks. For humans, such abilities can be accurately measured by IQ tests, and the results are valid measures of analytical intelligence, even if they don't reflect other abilities such as social intelligence [6]. One visual IQ test called *Raven's Progressive Matrices (RPM)*, introduced by the psychologist John Raven [7], is strongly indicative of abstract verbal, spatial and mathematical reasoning abilities of the subject [8].

An RPM consists of a 3×3 problem matrix. Each cell of the matrix contains some combination of shapes and lines where all features are clearly distinguishable. The objects in different cells are related to each other via a set of underlying rules which govern the problem matrix. These rules are not made explicit to the subject. The bottom-right cell of the matrix is intentionally left blank. The subject is given a set of candidate images and tasked with selecting the image which best completes the problem matrix. See figure 1.2 for an example problem matrix.

Previous research has utilized RPM as a benchmark to measure the abstract reasoning abilities of deep neural networks [10, 11, 12, 13]. With similar motivations as in Chollet's work, Barrett et al. created a novel dataset called *Procedurally Generated Matrices (PGM)*. They place their focus on the generalization ability of the trained models to new, unseen PGMs from different distributions [10]. On the other hand, Zhang et al. focus more on the setting

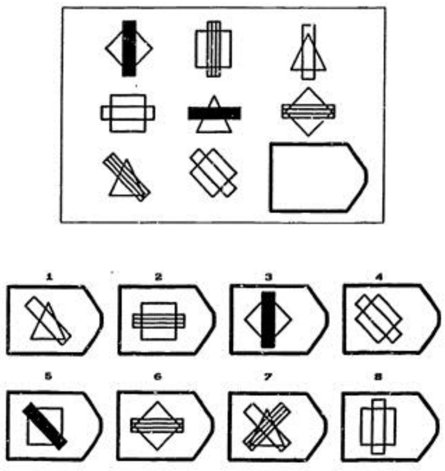

Figure 1.2: A typical RPM problem matrix [9].

where the training and test data come from the same distribution [11]. They too created a new dataset, called RAVEN: A Dataset for Relational and Analogical Visual rEasoNing, on which they test their models. While Barrett et al. and Zhang et al. attempt to tackle the problem in its original formulation, i.e. the discriminative setting, Hoshen and Werman pursue the problem of *generating* the last panel of the problem, albeit for much simpler problems than those pursued by the first two works [13]. Finally, Steenbrugge et al. and Steenkiste et al. independently show that learning representations of RPMs in the latent space using disentangled VAEs helps improve the generalization performance [14, 15]. Note that all of the aforementioned work considers the case where the inputs are on the pixel-level, that is, the models take as input an image depicting the problem matrix.

Excluding the work of Hoshen and Werman [13], all other works attempt to solve the classification task, where the goal is to recognize the candidate image that best completes the problem matrix, out of 8 choices. In this work, we consider the harder problem of generating the 9^{th} panel and predicting the rules which govern the matrix, given the first 8 cells. To this end, we use the matrix rules as additional supervision in our training setup. While it is conceivable that a model can exploit certain superficial statistics in the discriminative setting, understanding the relations between the grid cells is a prerequisite to generating the 9^{th} panel, unless one uses a very dense sampling of the problem matrix distribution.

To solve the generative task, we design a 2-step model pipeline. In the first step, the model receives as input the incomplete problem matrix and infers the rules that govern it. In the second step, the model combines the inferred rules with its initial input to generate the 9^{th} panel.

Feeding problem matrices to models in raw pixel form brings the additional challenge of perceiving the relevant shapes and lines. However, a model's perception capabilities are not relevant in understanding their ability to *reason* about their inputs. Due to this, our models operate on object-level representations of problem matrices rather than pixel-level images. As both PGM and RAVEN provide problem matrices only at the pixel level, we generate a new dataset, *Regenerated PGM (RPGM)* which essentially replicates the PGM dataset, aside from a few simplifications.

Our GNN-architecture recognizes the presence or absence of a rule in any given problem matrix with a mean accuracy of 91.2%. Given an incomplete problem matrix and the rules which govern it, our simple fully connected network can generate the 9$^{\text{th}}$ panel with a mean accuracy of 18.3%. While the architecture proposed here is not sufficient to reliably generate the last panel, it still demonstrates the ability of neural networks to reason to a certain extent, given that we use modest amounts of training data, especially for the generation task.

The rest of this manuscript is organized as follows: In chapter 2, we motivate the problem that the thesis is attempting to tackle. Chapter 3 contains a concise description of graph neural networks, which form the basis of our rule inference models. Chapter 4 details the data generation process: we use the pipeline introduced there to generate datasets which we subsequently use to train our models. Chapter 5 introduces and motivates the generic model template and its various instantiations. In chapter 6, we share our experimental methodology and discuss the results. Chapter 7 discusses the lessons learned, ways to improve upon the approach presented in this manuscript, and future research directions for making learning more efficient and generalizable. The final chapter provides a brief summary of the project.

Chapter 2

Motivation

At first, using object-level descriptions instead of raw pixel inout may seem counter-intuitive. The astute reader will notice that there may easily exist a deterministic program that solves the generative task with a 100% accuracy, and this is indeed the case. However, this thesis is not about solving a problem for which conventional methods fail. Rather, it is about gaining insights into the capabilities of neural networks.

2.1 Abstract Reasoning & Intelligence

To motivate our choice of the learning task, we first argue why the existing results are not enough to convince us that neural networks are capable of abstract reasoning.

First, note that in the discriminative setting where models are trained to identify the correct grid cell out of a set of candidates, the rules are not used as additional supervision when training models. The success of models in the discriminative setting is thought to imply that the models also understand the rules which govern the problem matrices. However, it is well understood that neural networks *take the shortest path* to minimizing their loss functions, which is often through exploiting superficial statistics. The amount of statistical regularities to exploit is potentially even more significant in this domain, as datasets are generated procedurally. Therefore, the aforementioned implication does not necessarily hold. This argument is supported by the following results reported in [10]:

1. The model's confidence in its predicted meta-targets (the meta-target is a summary statistic derived from the rules of the problem matrix) and its accuracy are positively correlated.
2. Adding a regularizer term to the loss function which penalizes false rule prediction results in better generalization.

On the other hand, the ability to generate the 9th grid cell is much harder to explain just by statistical regularities: There are roughly 10^{33} possible cells that the model can generate, and it is infeasible to densely sample neither this space nor the input space. This is why we focus on the generative task.

Since the generative task is likely too complex to tackle directly, we instead focus on a simplified version. We assume that the matrix rules are available to us at training time. Our hope is that solving this relaxed version of the generative problem will give us insights about how to approach the harder task of learning to generate the 9th grid cell without any access to problem matrix rules. In chapter 7, we discuss some of these insights.

2.2 Relation to ARC

There are many similarities between generating the 9th grid cell and solving an ARC task. To a certain extent, one can draw an analogy between the first two rows of the problem matrix and the training pairs of an ARC task (For example, the first three columns in figure 1.1 are the training pairs). Then, generating the 9th grid cell is analogous to generating the outputs for the test inputs of an ARC task. Indeed, the third row of the matrix is governed by the same rules as the first two, just as the test pairs of an ARC task is governed by the same logic as the training pairs.

From this lens, the generative RPGM task can be interpreted as a special kind of ARC task. The ARC task in the general case is considerably more complex, as even the size of the output grid needs to be determined by the subject and there is less structure overall compared to a problem matrix, where shapes and lines may only exist in fixed locations.

As such, fully solving the generative task (i.e. without requiring problem matrix rules to be available during training) could in turn give us insights about how a neural network based approach to solve the ARC benchmark might look.

Chapter 3

Background

3.1 Graph Neural Networks

A graph neural network (GNN) is a neural network that operates on inputs in the form of graphs. Data in the form of graphs can arise in many disciplines, such as social sciences and biology. A GNN takes as input a graph, typically in the form of an adjacency matrix A, and a feature matrix X where the i^{th} row contains the features of the i^{th} node. The output of the GNN depends on the application. Possible outputs include per-node classification, per-node regression, but one can also seek some global classification regarding the entire input graph.

We can formulate the operation carried out by the $(j+1)^{\text{th}}$ hidden layer of a GNN generically as follows:

$$H_{j+1} = \sigma(f_\theta(H_j, A)))$$

where $H_{j,i}$ denotes the hidden representation of the i^{th} node in the j^{th} layer, θ represents the learnable parameters and σ denotes a non-linearity (we use the ReLU activation function).

Different kinds of GNN layers differ in the way that they define f. The vast majority of proposed layers contain some sort of aggregation operation: The new representation of the i^{th} node usually depends on the weighted average of its neighbors in some way.

In this work, we use two different GNN layers. First, a particularly simple layer which helps us establish a baseline, and then a more complex, recently proposed layer.

3.1.1 Simple GNN Layer

We first give the formula which describes this layer, and then elaborate on its meaning.

$$H_{j+1} = \sigma(D^{-1}AXW) \tag{3.1}$$

Here, D denotes the degree matrix of the input graph, i.e. $D_{i,i}$ is the number of neighbors of node i, and W are the learnable parameters. The term $(D^{-1}AX)_i$ corresponds to the average of the features of the neighbors of node i. Consequently, the expression $D^{-1}AXW$ corresponds to updating each node's representation as the non-weighted average of all its neighbors, and then applying a learnable linear transformation to the resulting feature matrix. One can arbitrarily set the length of the new node representations by selecting W's dimensions appropriately.

3.1.2 Graph Attentional Layer

We next review the graph attentional layer proposed in [16]. As above, we first give the formula which describes it, and then give some intuition on the performed operation.

$$\boldsymbol{x'^k}_i = \sigma(\alpha^k_{i,i}\boldsymbol{W}\boldsymbol{x_i} + \sum_{j \in \mathcal{N}(i)} \alpha^k_{i,j}\boldsymbol{W}\boldsymbol{x_j}) \tag{3.2}$$

$$\alpha^k_{i,j} = \frac{\exp(\text{LeakyReLU}(\boldsymbol{a}_k^\mathsf{T}[\boldsymbol{W}\boldsymbol{x_i} \| \boldsymbol{W}\boldsymbol{x_j}]))}{\sum_{k \in \mathcal{N}(i) \cup \{i\}} \exp(\text{LeakyReLU}(\boldsymbol{a}_k^\mathsf{T}[\boldsymbol{W}\boldsymbol{x_i} \| \boldsymbol{W}\boldsymbol{x_k}]))} \tag{3.3}$$

Equation 3.2 shows us the fundamental difference between the graph attention layer and the simple GNN layer defined above. The graph attention layer allows each node i to attach weights $\alpha_{i,j}$ to all of its neighbors j. The superscript k denotes that $\boldsymbol{x'^k}$ is the output of the k^{th} attention head. Each attention head potentially learns a different weighting $\boldsymbol{\alpha}^k$, allowing the model to simultaneously pay attention to multiple ways in which node features can be combined.

Chapter 4

The Data Generation Pipeline

Recall that we intend our models to take as input high-level object descriptions of problem matrices, rather than pixel-level renderings. Since the PGM dataset is published in an image format, we resort to regenerate this dataset by closely following the recipe provided in [10]. The regenerated dataset is referred to as Regenerated PGM (RPGM). This chapter outlines the generation process of RPGM which we use in subsequent stages to train our models.

We first explain the characteristics of the generated datasets, and then go through the generation process of an individual problem matrix. Finally, we detail the file format of problem matrices.

4.1 Differences Between PGM and RPGM

In generating RPGM, we have decided to simplify the PGM dataset in three different ways. This section explains each of these simplifications.

1. In the original PGM dataset, the authors generated multiple datasets where each dataset is governed by a ″regime″ [10]. For instance, a dataset D_A may only contain small to medium sized shapes, while another dataset D_B contains larger shapes. These datasets are then used to assess a given model's generalization capabilities by training it on dataset D_A and testing it on dataset D_B. For our purposes, it suffices to generate the dataset governed by the ″neutral regime″ which does not restrict its problem matrices in any way.
2. In addition to the above, the authors of [10] also distinguish between the *distracting* and *non-distracting* settings for the datasets. In the distracting setting, the shape and line attributes that are not governed by any *rule* (as we will shortly define) are drawn randomly from the set of values which they can take. In the non-distracting setting, these at-

tributes are held constant across the entire problem matrix. Intuitively, the distracting setting is harder since one needs to pay close attention to each object attribute and check if the values can be explained by the existence of a rule. In contrast, the non-distracting setting is simpler because it is easy to detect if a quantity is held at a constant, and thus one needs to focus on fewer details to solve the classification task.

   Sampling attribute values in a way that does not give rise to spurious rules (as in the distracting setting) would incur a significant overhead in the implementation effort. Furthermore, we consider the learning task for our models to be much harder than classification, which was the task pursued in [10]. Thus, we have decided to generate our datasets only in the non-distracting setting.

3. The final simplification is regarding the orientation of the rules in a problem matrix. In the PGM dataset, a rule could relate either the cells in the same row or the cells in the same column. We constrain the problem matrices of RPGM such that a rule can only relate the cells in in the same row. This simplification complies with the original specification for RPMs given by [17]. The same simplification was also followed by [11] when creating the RAVEN dataset.

We next explain how an individual problem matrix is generated.

4.2 Problem Matrix Generation

At a high level, the problem matrix generation is a 2-step process:

1. Sampling the set of rules that will be present in the problem matrix.
2. Sampling attribute values for the problem matrix which adhere to the set of rules sampled in step 1, and that are otherwise constant.

4.2.1 Sampling the set of rules

We first define what is meant by a rule and give some examples to convey its meaning. We then define the set of all rules and the set of all valid combinations of rules. For the detailed definitions of all rules, we refer the reader to [10].

A rule is formally defined as a triplet (o, a, r) where $o \in Objects$ is the type of object, $a \in Attributes$ is an attribute of object o and $r \in Relations$ is a relation. The "rule" is that the relation r should be present between the attributes a of all the objects in the problem matrix. The sets, *Objects*, *Attributes* and *Relations* are defined as follows:

$$Objects = \{shape, line\}$$

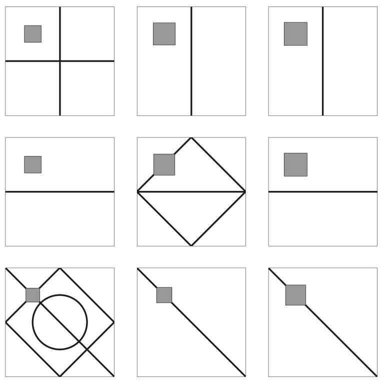

Figure 4.1: An example problem matrix. This matrix contains the rules $(shape, size, progression)$ and $(line, type, and)$.

$$Attributes = \{size, color, type, number, position\}$$

$$Relations = \{progression, xor, or, and, consistent\ union\}$$

To get an intuition about the meaning of a rules, consider the rule $(shape, size, progression)$. If this rule is present in a problem matrix, the shapes that are contained within the matrix will be growing in size as we go from the leftmost column towards the rightmost column. In other words, the shapes will be getting *progressively* bigger.

Similarly, consider the rule $(line, type, and)$. The presence of this rule in a problem matrix indicates that the following statement will hold for each row of the matrix: The types of lines that appear in the third grid cell are exactly those which also appeared in both the first and the second grid cells.

Figure 4.1 illustrates the instantiation of a problem matrix in which these two rules are present. Note that besides the size of the shapes and the types of the lines present, every other attribute is constant. In particular, all shapes are squares, they are the same color, they are in the upper left portion of each grid cell and there is one shape per grid cell. A similar observation can also be made for the lines in the background. The fact that all these attributes are constant is a direct consequence of generating problem matrices in the non-distracting setting.

Next, we define the set of all rules as the union of the following Cartesian products:

$$S_1 = \{shape\} \times \{size, color, type\} \times Relations$$

$$S_2 = \{shape\} \times \{number\} \times \{progression, consistent\ union\}$$

$$S_3 = \{shape\} \times \{position\} \times \{xor, or, and\}$$
$$S_4 = \{line\} \times \{color\} \times Relations$$
$$S_5 = \{line\} \times \{type\} \times \{xor, or, and, consistent\ union\}$$

This definition strictly follows [10]. The reason for the absence of the remaining triples is intuitive. For example, $(line, type, progression)$ is absent because an order over different types of lines such as diagonal, horizontal, etc. is not naturally defined. Similarly, $(shape, number, or)$ is missing because the number of shapes in a grid cell is always unique, and thus there is no notion of sets of numbers of shapes.

Lastly, we define the set of all valid combinations of rules via a collection of constrains that are imposed over each valid combination.

Any valid combination of rules...

1. may not contain two rules r_1, r_2 such that $r_1 = (o, a, r)$ and $r_2 = (o, a, r')$. Intuitively, this constraint states that it is not allowed to impose two different relations over the same attribute of the same type of object. This is trivially impossible.
2. may not simultaneously contain a relation on the number of shapes and a relation on the position of shapes. Such a combination is not instantiable, as can be seen by attempting a combine a progression relation on the number of shapes with any set relation on the position of shapes. The constraints imposed by these rules work against each other.
3. may not simultaneously contain two rules r_1, r_2 such that $r_1 = (line, position, r)$ and $r_2 = (line, color, r')$ where $r' \in \{xor, or, and\}$. Similar to the previous constraint, these rules often work against each other when one attempts to instantiate them, so we don't allow them to occur simultaneously.
4. may not simultaneously contain two rules r_1, r_2 such that $r_1 = (shape, position, r)$ and $r_2 = (shape, a, r')$ where $a \in \{size, color, type\}$ and $r' \in \{xor, or, and\}$. Similar to the previous constraint, these rules often work against each other when one attempts to instantiate them, so we don't allow them to occur simultaneously.

In theory, a valid combination can have as many as 6 rules. However, the authors of [10] have constrained the PGM dataset to contain problem matrices that only have up to 4 rules. Accordingly, our rule sampling procedure operates as follows:

1. sample $n \xleftarrow{u.a.r.} \{1, 2, 3, 4\}$.
2. sample n rules u.a.r. such that their combination is valid.

Attribute Values			
Sizes (scaling factors)	Colors (grayscale intensities)	Shape types	Line types
1	1	circle	diagonal down
2	2	triange	diagonal up
3	3	square	vertical
4	4	pentagon	horizontal
5	5	hexagon	diamond
6	6	octagon	circle
7	7	star	
8	8		
9	9		
10	10		

Table 4.1: All possible attributes for each shape and line.

4.2.2 Sampling attribute values

Next, we outline the procedure of sampling a problem matrix which obeys a given combination of rules. Sampling of a problem matrix is simply a question of determining the attributes of the shapes and lines that will be present in each grid cell. For each shape that will be present in the matrix, we sample its size, color, type (e.g. a circle, triange, etc.) and position. Similarly, we sample the color and type (vertical, left diagonal, etc.) of each line that will be present in the matrix.

Table 4.1 lists the set of possible values for each attribute. All values are taken directly from [10]. Note that for sizes and colors, the partitioning of the scaling factor space and the grayscale intensity space is already too fine-grained for the human eye. To us, a shape with $size = 7$ and $color = 4$ is indistinguishable from another shape with $size = 6$ and $color = 5$.

As stated above, the sampling must be done in a way such that all of the given rules are obeyed, and no other rule is obeyed. Note that given a valid combination of rules, the problem matrix which corresponds to this combination is never unique. Figure 4.2 illustrates the variety of matrices that can arise.

The sampling of attribute values works in the following way. Shape and line attributes are treated and sampled separately and combined at the end of the procedure. We outline the sampling of shape attributes below, the procedure for line attributes is completely analogous.

For shape attributes, we first sample values for the sizes, colors and the types of shapes that can appear in the problem matrix. If there is a rule concerning a given attribute, the sampling for that attribute obeys that rule. Otherwise,

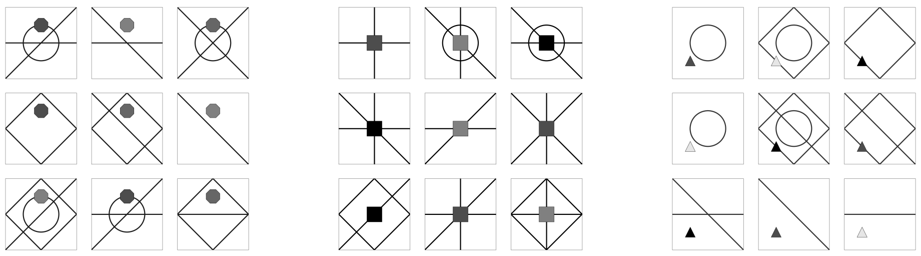

Figure 4.2: All three problem matrices obey $(shape, color, consistent\ union)$ and $(line, type, xor)$. The first rule states that the same three colors appear on the shapes of each row. The second rule states that for every row, the lines in the third grid cell are those that appeared either in the first cell or the second cell, but not both.

one value is sampled for that attribute, and each shape that appears in the the problem matrix is assigned this value.

For example, assume that $(shape, type, or)$ is in the rule combination. In this case, we sample sets of shape types for each one of the 9 grid cells in the matrix such that the sets for the cells in the third column always correspond to the union of the respective sets in the first and the second column. We ensure that the sets in the first and second column are neither disjunct nor identical, so that one can distinguish between $\oplus$, $\cup$ and $\cap$.

On the other hand, assume that no rule regarding the sizes of shapes is present. In this case, we sample one size from the set of all sizes and assign it to all shapes that appear in the problem matrix.

After the set of values for the sizes, colors and types that a shape can take are determined, we next determine the amount of shapes for each grid, and the position of each shape within a grid. The values are chosen in the same way as explained above. Additionally, we ensure that there are enough shapes in a grid cell such that all the different size, color and type values for that grid cell can be represented.

Finally, we assign to each shape a size, color and type from the respective grid sets. We make sure that each attribute value from a grid set is assigned to some shape within that grid, so that the corresponding rules can actually be recognized.

4.3 File Format of a Problem Matrix

After all attributes have been determined, we store the generated problem matrix with a one-hot encoding scheme, and the rules it obeys with an indicator vector.

4.3.1 Encoding of a problem matrix

The encoding of the problem matrix consists of the concatenation of the representations of each grid cell, where each grid cell is represented by 336 0's and 1's:

- The first 9×11 elements encode the sizes of the 9 shapes in one-hot encoding,
- The following 9×11 elements encode the colors of the 9 shapes in one-hot encoding,
- The following 9×8 elements encode the types of the 9 shapes in one-hot encoding,
- The remaining 6×11 elements encode the colors of the 6 lines in one-hot encoding.

Notice that we always use 1 more element than the corresponding value set's cardinality to encode an attribute. This is because we also take into account the case where there is no shape/line in the position that we are encoding.

4.3.2 Encoding of the rules

As stated above, we encode the rules present in a matrix with a vector v that holds 29 indicator variables.

$$v_i = \begin{cases} 1 & \text{if rule } i \text{ is present} \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

Chapter 5

The Model

As we briefly mentioned in the introductory chapter, we would like to generate the 9th panel and infer the present rules, given the first 8 cells of the problem matrix. We first give a high level overview of our design and motivate it. Afterwards, we dive into the details.

As you may have already noticed from attempting to understand the examples in figures 1.2 and 4.1, there exists a principled approach to solving a problem matrix. Even in the discriminative setting where one has to identify the right candidate panel, one should first observe the given incomplete matrix and identify emerging structures. Only after understanding which rules are governing the problem matrix can one start assessing how well each candidate cell fits in with the rest of the grid cells. Our design attempts to model this approach.

Figure 5.1 illustrates our design. We train two distinct neural networks: The *Classifier* Network and the *Generator* network. The Classifier learns to predict which rules are present in a given incomplete problem matrix. The Generator learns to generate the 9th panel, given the first 8 panels and a rule vector which specifies the rules that govern the matrix.

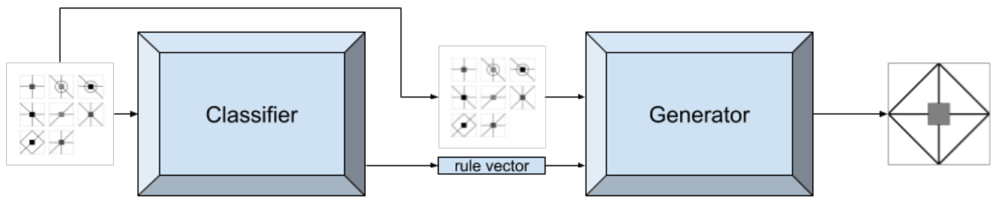

Figure 5.1: The generative pipeline.

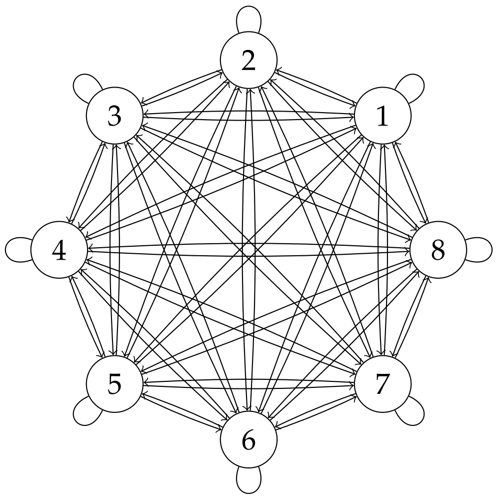

Figure 5.2: The incomplete problem matrix is cast into a fully connected graph with self loops.

5.1 The Classifier Network

5.1.1 Model architecture

The Classifier $\mathcal{C}$ takes as input a binary vector of size $8 \times 336 = 2688$. This vector contains the object-level description of the first 8 grid cells of the incomplete problem matrix, where the encoding of each cell is essentially a concatenation of one-hot encodings. More details on the encoding can be found in section 4.3.1. $\mathcal{C}$ outputs 29 values in $[0, 1]$, where the i^{th} value represents the confidence of $\mathcal{C}$ that the i^{th} rule is present in the problem matrix.

We next describe the general design of the Classifier $\mathcal{C}$. This design is shared by all different models that we developed. Individual models differ in their choice of the particular type of GNN layer that they use.

Our design consists of a multi-layer GNN, followed by a fully-connected network (FCN). To use GNNs, we cast the incomplete problem matrix into a fully connected graph with self loops where each grid cell corresponds to a node. Figure 5.2 illustrates this setup. We initialize the nodes with the object-level descriptions of the corresponding grid cells, concatenated with a position tag which encodes the location of the grid cell within the problem matrix.

The GNN takes the fully-connected graph as input and operates on it for a certain number of layers, depending on the specific model. After the last layer of the GNN, all learned representations are averaged (or concatenated, depending on the model), and the resulting vector is fed into an FCN which uses ReLU activations between its linear layers, and outputs a

29-dimensional vector. Each element of the output vector is passed through the sigmoid function to get the final confidence values for every rule.

5.1.2 Justifications for the model design

Why graph neural networks?

We first share the following insights:

1. When trying to understand if a rule is present in a given problem matrix, a sensible approach is to check the conditions of that rule for each row. That is, we apply the same abstract mathematical operation to each row.

2. When studying a problem matrix, we tend to compare 2 grid cells with each other at a time. This comparison operation depends on which grid cells we are comparing, but only to a certain extent.

These observations suggest that a possible solution applies the same function, or a family of closely related functions repeatedly over regions of the incomplete problem matrix. This in turn suggests that weight sharing may be significantly more efficient than attaching weights to every single pairing of problem matrix attributes.

Assume that we decided to use a pure-FCN approach to solve this task. Even if we reduced the representation of our input from 2688 dimensions to 400 dimensions in our first layer, we would incur over 800k learnable parameters. Just the first layer would need a gigantic training set to properly train.

Thus, GNNs seem to be a good choice for our problem. They enable weight sharing, thus reducing the complexity of the model while retaining its capacity. Furthermore, their operation can be understood as focusing on pairs of grid cells at a time which matches the nature of our task.

Why a fully connected graph?

At first, a grid structure may seem more appropriate than connecting all nodes to each other. However, we note that the graph lays down the infrastructure for message passing, and there is no good reason not to compare cells that are not adjacent to each other. In fact, there is no good reason to prohibit direct comparison between any two grid cells. On the contrary, imposing any sort of structure to the graph may introduce a non-inductive bias. Due to these reasons, we use a fully connected graph and let the model learn from data which pairs of grid cells it should place its attention on.

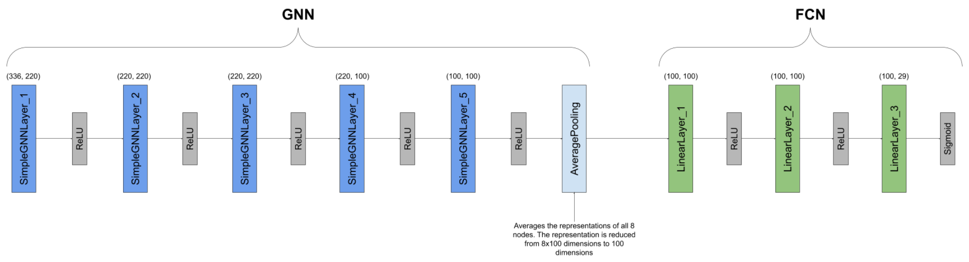

Figure 5.3: Our baseline model.

5.1.3 Baseline

Figure 5.3 illustrates our baseline architecture. The architecture is simply an instantiation of the general design described in section 5.1.1. The baseline model has 265229 trainable parameters.

The baseline suffers from two important design flaws:

1. Since the simple GNN layer proceeds by computing the *unweighted* average of the neighborhood (including itself) each node, each grid cell contributes equally to representations of each grid cell. For instance, a cell in the last row influences the representation of the upper left cell as much as the upper left cell's prior representation itself. This can cause the model to have a non-inductive bias.

2. More fatally, because the graph is fully connected, all cells are updated to be the mean of all cells before the network performs any subsequent computation. This means that our baseline is invariant to the order of the cells, once it is fully trained. This can make it harder for the model to distinguish the rules which depend on the order of cells, such as progression.

Despite its shortcomings, we will see that the baseline is already quite powerful. We next present a family of models which overcome these flaws.

5.1.4 The n-headed GATNet

Recall from section 3.1.2 that graph attention layers employ an attention mechanism which allows the network to learn for every node a set of weights for each neighboring node, including itself. This ability solves both issues of the baseline:

1. The unweighted sum becomes a weighted sum where the weights are unique for each node. Because the weights are learned, the model's bias is now obtained from the training set, which means it is consistent.

2. The uniqueness of the weights breaks the order invariance.

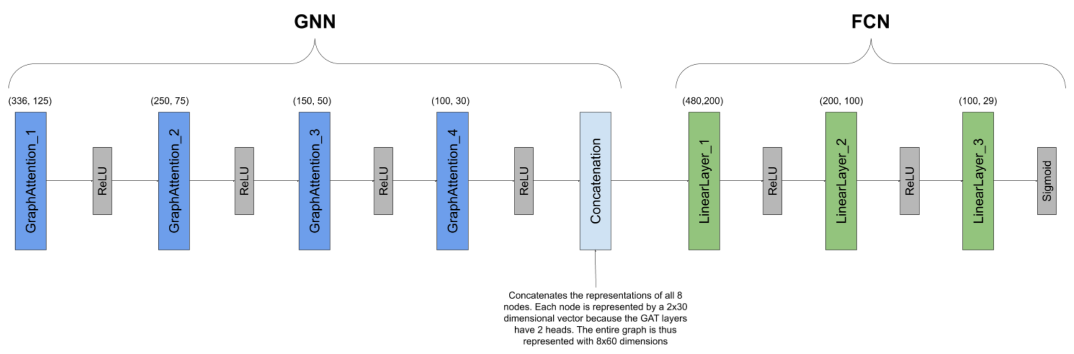

Figure 5.4: The 2-headed GATNet.

Moreover, using n heads will cause GATNet to learn n sets of weights, allowing it to interpret its input in n different ways in each hidden layer. Combining these different interpretations may help the model to become good at identifying rules on different relations, as opposed to being particularly good at one class of rules, e.g. rules with the progression relation, and performing poorly in recognizing other rules.

Compared to the baseline, GATNet is modified in the following ways:

- The GNN is composed of 4 layers instead of 5.
- Instead of simple GNN layers, we use GAT layers in the GNN.
- We replace the average pooling at the end of the GNN with a concatenation.

The resulting model contains roughly 273500 trainable parameters. The exact number depends on the number of attention heads we use. Figure 5.4 illustrates the detailed design.

5.2 The Generator Network

In the beginning of this chapter, we explained that the Generator will take as input the incomplete problem matrix concatenated with the rules that govern it, and produce the 9th panel.

However, early experiments showed that using all 8 grid cells of the input introduces too many learnable parameters due to the sheer size of the input (2688 dimensions). Using GNNs is not straightforward in this case either. Since we are also passing the rule vector to the model, it is not clear if one should append the rule vector to the representations of each grid cell, or perhaps design the model in such a way that it takes graphs as input, but can additionally condition on a special vector.

On the other hand, we feel that the generative task does not necessarily have a complex solution. Since the rules are given, the model only needs to learn the meaning of each rule as demonstrated by the training samples. The rules themselves are simple mathematical relations, such as progression and set operations, which are easily learnable. As it will turn out, the hardness of the generative task indeed lies in combining multiple rules, rather than learning them.

Taking this discussion into consideration, we make the following simplification to our dataset in order to see the forest through the trees: We disallow the *consistent union* relation from occuring in the dataset which we use to train our Generators. This simplification allows the correct 9th panel to be recovered just from the two grid cells in the last row and the rules which govern them. Indeed, none of the remaining rules relate different rows of the problem matrix with each other. They only define constraints that have to independently hold for each row. Thus, we can now get away with passing to our Generator only 2 grid cells instead of 8, which greatly reduces the size of the input.

Before proceeding, we explain one final necessary modification to our dataset. Recall that when encoding a certain position within a grid cell of a problem matrix, say the center of the grid, we encode the attributes (size, color and type) of the shape that will occupy that position *separately*. If there will be no shape in the said position, all three attributes take the same special value (0). Otherwise, none of them take the value 0, as 0 indicates that there is not a shape at that position. This is an implicit constraint satisfied by all problem matrices in our datasets.

However, our model may not necessarily obey this constraint. If we don't additionally ask our model to predict for each position if there is a shape at that position, and only ask it to predict attribute values for the shape, the model may then generate a panel encoding which on the one hand encodes that the shape in the bottom left corner should be a certain shade of gray (by selecting the class that corresponds to that shade) but on the other hand encodes that there shouldn't be a shape in the bottom left corner at all (e.g. when encoding the type of the shape). In this case, it is not clear if the model would prefer there to be shape or not.

To solve this issue, we append the object-level representations of all grid cells with a 9 dimensional binary vector. The i^{th} element of this vector is an indicator variable which encodes if a shape exists in the i^{th} position of the grid cell. We also ask the model to generate this vector as part of its output, and use it to guide the panel generation process. We name this modified representation as the *augmented representation*.

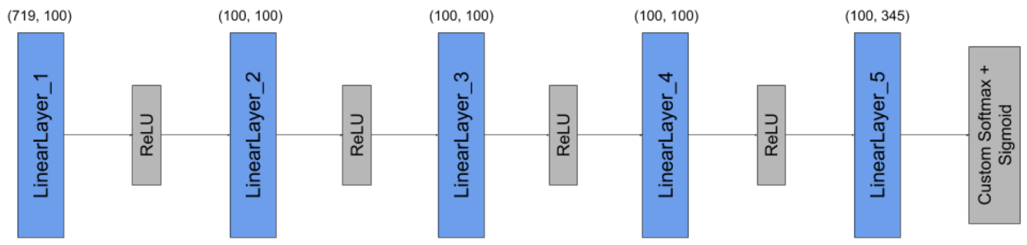

Figure 5.5: Our simple generator design.

5.2.1 Model architecture

The Generator $\mathcal{G}$ takes as input a binary vector of size $2 \times 345 + 29 = 719$. This vector contains the augmented representations of the last 2 grid cells of the incomplete problem matrix, appended by the rule vector which specifies the rules that govern the problem matrix which the 2 grid cells belong to.

$\mathcal{G}$ outputs 345 values. We apply the softmax and sigmoid functions over $\mathcal{G}$'s output in such a way that it conforms to the grid cell encoding format introduced in section 4.3.1.

As for the actual architecture itself, we use a very simple fully connected network (FCN) in accordance with our intuition on the hardness of the problem. Figure 5.5 illustrates our design. In total, the model has 137145 trainable parameters.

Chapter 6

Experiments & Results

6.1 Experimental Setup

We implement our data generation pipeline in NumPy. All of our models are implemented using NumPy [18] and PyTorch [19]. We use PyTorch-Geometric [20] to implement GNNs. All plots are generated using Matplotlib [21]. Our code is publicly available under `https://github.com/ekarais/bachelor-thesis`.

6.2 Inferring the Rules

For rule inference, we generate a dataset with $256k$ problem matrices. We use an $80\% - 10\% - 10\%$ training−validation−test split. On average, each matrix is governed by 2.5 rules. This number is close to twice as high as the average of the original PGM dataset (1.37, [11]). Since our labels are binary vectors with 29 dimensions, this leads to significant class imbalance. To prevent the model from simply learning to always predict the 0-vector, we penalize false negatives 10.6 times more than false positives.

We use a batch size of 2560. This number is a compromise between the advantages of using a smaller batch size [22] and the faster training times provided by using a larger batch size. We use the ADAM optimizer [23] and an initial learning rate of 10^{-3}. The learning rate is empirically determined. We use binary cross entropy as our loss function. Although we don't explicitly regularize our models, we do employ a learning rate scheduler which monitors the model's loss on the validation set. If the validation loss does not decrease for 10 consecutive epochs, we update the learning rate to be 75% of its old value.

Recall our generative pipeline in figure 5.1. We assumed that the Generator G receives the incomplete problem matrix and the *correct* set of rules as input

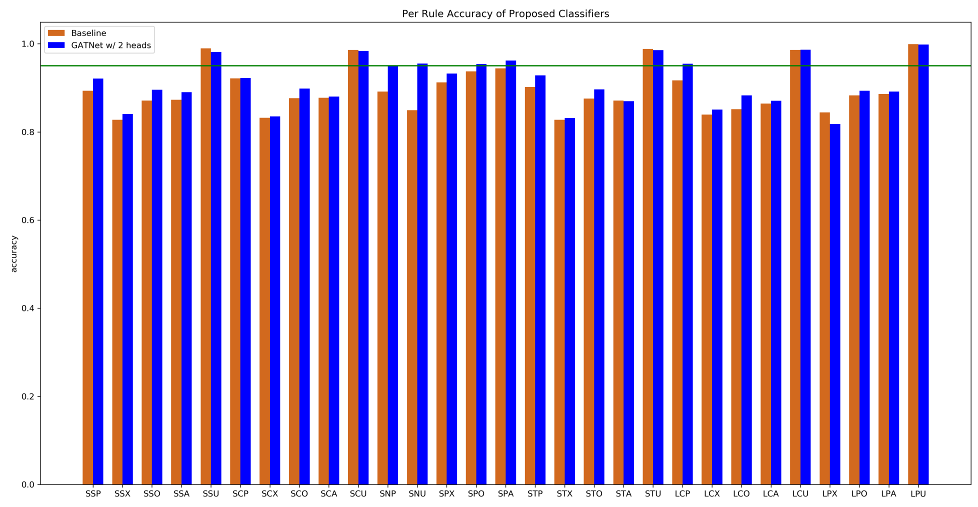

Figure 6.1: The average accuracies of our baseline model and the 2-headed GATNet, for each rule. Rules are encoded by the initials of their components. For instance, SPO represents $(shape, position, or)$. The green horizontal line corresponds to the 95% mark.

when generating the 9th grid cell. Since the rule vector will be produced by the Classifier C, we would like to know how often C predicts the *entire rule vector* correctly, given an incomplete problem matrix. We refer to this quantity as the *hard accuracy*. Note that hard accuracy is upper bounded by the model's average accuracy in classifying a rule correctly.

In the following, we report results from running inference on the test set, unless stated otherwise.

6.2.1 Baseline versus GATNet

We first look at how our baseline and the 2-headed GATNet perform after being trained for 300 epochs. Figure 6.1 depicts their accuracies for every rule, on the held-out test set. Note that both models contain roughly the same amount of learnable parameters.

Observe that the 2-headed GATNet consistently outperforms the baseline, with very few exceptions. Moreover, the baseline only attains an accuracy over 95% for rules that contain the *consistent union* relation. Recall from figure 4.2 the meaning of this relation. The *consistent union* relation over an attribute dictates that there are only 3 different values that these attributes can take, and each of these vales must be present at exactly one grid cell per row. Since we only generate problem matrices in the non-distractive setting, the consistent union relation is actually invariant to the order of grid cells. The invariance does not hold for any other relation.

Now recall that the baseline model is also invariant to the order of grid cells in the problem matrix. This fact explains why the baseline model is as good as the 2-headed GATNet at classifying consistent union rules, but performs worse at any other rule. Breaking the order-invariance seems to help the 2-headed GATNet better identify all rules.

The accuracy difference between the two models is most significant for rules on the number of shapes. This could be caused by the way the number of shapes per grid cell is encoded in our representation. Unlike sizes, colors, types and positions which are explicitly encoded, the model would have to count the number of one-hot encoding vectors whose first element is zero in order to compute the number of shapes. This could require the model to combine the grid cells of its input in a different way. Since the baseline does not have multiple heads, it may be directed towards the representation that is not helpful in classifying rules on numbers of shapes, but is helpful for rules on other attributes.

Finally, we observe that both models struggle the most in identifying *xor* relations. We know that the *xor* function is harder than *or* and *and* functions because it is not linearly separable. We suspect that this inherent complexity may be causing the models to perform worst in *xor* relations.

6.2.2 Number of heads

Next, we investigate the tradeoff between the number of heads and the number of learnable parameters per head, or equivalently, the size of the hidden representations. We instantiate three GATNets with 1, 2 and 3 heads. We adjust the sizes of the GNN layers such that all three models have roughly $275k$ learnable parameters. For example, the layers of the GATNet with 2 heads produce outputs that have half the dimensions as the GATNet with 1 head. The outputs of the 2 heads are then concatenated to obtain a final output that is equal in size to the output from the GATNet with 1 head.

Figure 6.2 illustrates the results. We observe that the model with 1 head significantly falls behind compared to the models with multiple heads. This result indicates that the ability to combine the grid cells of the input in multiple ways is indeed beneficial. On the other hand, the model with 3 heads performs only marginally worse than the model with 2 heads.

Table 6.1 depicts the average accuracies of these models. We see that on average, the model variant with 3 heads is actually slightly better at correctly classifying a given rule, compared to the variant with 2 heads. We thus arrive to the conclusion that there is little difference between 2 or 3 heads.

In our experiments, the GATNet with 2 heads achieved the best hard accuracy (18.2%). Although the models easily achieve average accuracies over 90%, their performance is still not good enough to reliably predict the entire

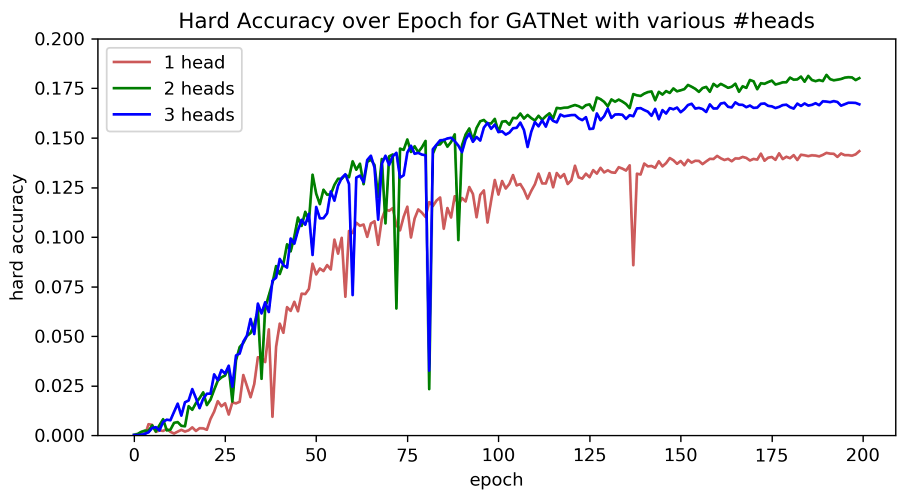

Figure 6.2: The hard accuracy curves of GATNet with 1, 2 and 3 heads on the test set.

number of heads	1	2	3
average accuracy	0.899	0.907	**0.915**

Table 6.1: Average rule classification accuracy of GATNet with 1, 2 and 3 heads after 200 epochs of training.

rule vector correctly. In section 7, we discuss possible ways of boosting our model's accuracies even further.

6.3 Generating the 9th panel

For the generative task, we create two datasets D_1 and D_2, each with $25.6k$ problem matrices. As in the previous setup, we use an $80\% - 10\% - 10\%$ training−validation−test split. Every problem matrix in D_1 contains exactly 1 rule. On the other hand, each matrix in D_2 is governed (on average) by 2.5 rules. As explained in the previous chapter, none of the rules contain the *consistent union* relation.

As an artifact of the dataset generation process, the most common value for each attribute is the value which indicates the absence of the corresponding object. This introduces an artificial class imbalance on the attribute values which the model is trained to discriminate. To combat this artifact, we collect attribute statistics over the 9th panels in our dataset and weigh all attribute values accordingly.

We use a batch size of 64. Compared to the rule inference task, we can afford to use this few both due to the lower number of learnable parameters of our model and the size difference between the datasets, which is an order of magnitude. We use the ADAM optimizer [23] and an initial learning rate of 10^{-2}. The learning rate is empirically determined. Our loss function is an unweighted sum over cross entropy loss functions applied over each attribute value that the model generates. Each cross entropy function is weighted according to the statistics of the corresponding attribute. For the last 9 elements of the model's output which indicate the presence or absence of shapes in the corresponding positions, we apply binary cross entropy. As in the rule inference setup, we employ a learning rate scheduler which works in the same way.

When computing the loss function, we compare the output of the model to the encoding which was generated by the data generation algorithm. The astute reader will notice that it is not immediately clear why this comparison is reasonable. Indeed, if a rule containing the *progression* relation is present in the problem matrix, there may be multiple correct 9th panels. For instance, if the rule is $(shape, type, progression)$ and the types of shapes in the first and second grid cell in the last row are *circle* and *triangle* respectively, the model may select any type with a higher number of corners.

In the above example, the data generation selects the type of the shape uniformly at random from all shapes that have at least 4 corners. Computing the loss in the way we described will thus bias the model towards choosing the mean value out of all eligible values for that attribute. While it is not necessarily helpful, this bias is not harmful either, and we accept it for the lack of a better way of computing the loss function.

Next, we describe the simple process of of computing the 9th panel from the logits that the model outputs. The attributes of the lines in the background are computed in a straightforward way. For each line, the model generates a vector with 11 elements where each element corresponds the confidence score of assigning the corresponding color to the line. The first element corresponds to the confidence that the line should not exist. Let us call this vector the attribute vector for the line colors. We simply take the index of the maximum of the attribute vector. For shapes, we first check the last 9 logits that the model outputs. As explained before, the i^{th} logit corresponds to the confidence of the model that there should be a shape in the i^{th} position of the panel. If the logit is smaller than 0.5, we determine the shape, color and type values of the i^{th} shape as 0: the special value which indicates that these shapes do not exist. Otherwise, we observe the attribute vectors for the shape, color and type of that object, and take the index of the maximum of the corresponding attribute vectors, excluding the first element.

Finally, we distinguish between two types of accuracy.

- The *regular accuracy* measures how often the model generates a panel which correctly completes its input.

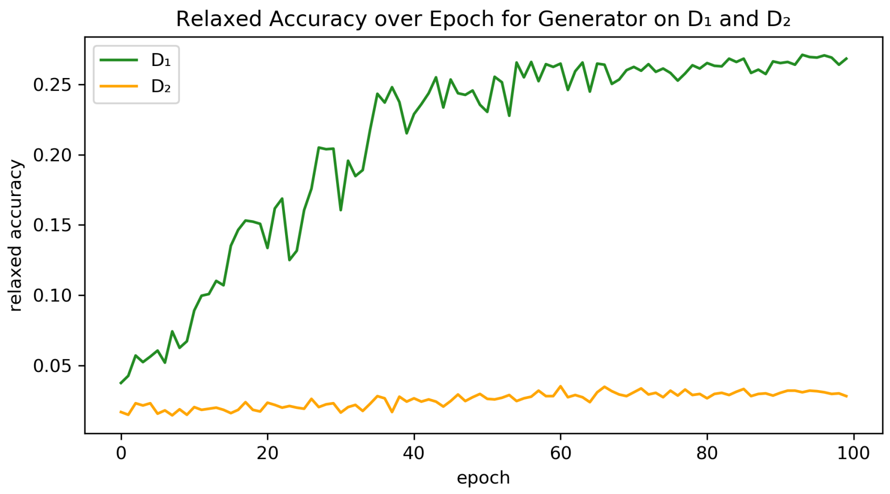

Figure 6.3: The relaxed accuracy curves of the Generator on the test sets of D_1 and D_2.

- The *relaxed accuracy* measures how often the model generates a panel which obeys the rules that are *present in the matrix*.

The difference between these two measures is subtle but important. Recall that we generate problem matrices in the *non-distractive* setting. This means that attributes which are not governed by a rule are held to a single constant value across the entire problem matrix. The regular accuracy deems an output correct if and only if the attributes which are governed by a rule obey that rule *and* all other attributes are held to a constant across the problem matrix. On the other hand, the relaxed accuracy measure only checks if attributes that are governed by a rule are chosen such that the rule is obeyed.

In our pipeline, we check the correctness of the generated panels with respect to both measures *on the fly* by feeding the model's batch output to an auxiliary function.

6.3.1 Experiments: understanding the root of the complexity

We train the Generator described in section 5.2.1 on the datasets D_1 and D_2 for 100 epochs. Figures 6.3 and 6.4 show the relaxed accuracy and loss curves obtained over both datasets respectively.

Figure 6.3 shows that when trained on D_1, the Generator attains a relaxed accuracy of about 26.8%. While not perfect, this result already demonstrates the ability of the Generator to understand and apply individual rules. On the other hand, the model's relaxed accuracy on D_2 is far worse, only about 2.8%. This discrepancy between relaxed accuracies shows that the model's internal data representation is likely not suitable for the generative task.

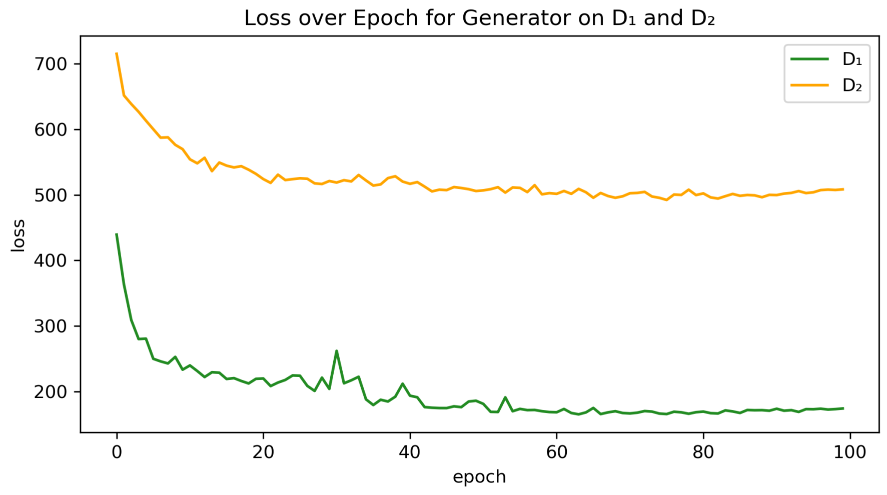

Figure 6.4: The loss curves of the Generator on the test sets of D_1 and D_2.

While the model performs reasonably well at generating panels which obey one rule, it struggles to generate panels that obey multiple rules, although these rules may govern attributes that are independent of each other.

We believe that the model is not learning the meaning of rules in a flexible way. In particular, the model seems to fail to make the insight that problem matrices which share some rules are similar in nature. Since D_2 contains problem matrices with multiple rules, it is presumable that D_2 also contains *more information* about the generative task, since the model can observe more rules. However, because the aforementioned insight is not explicitly embedded into the model's architecture, it is rather overwhelmed by this richer dataset. The loss curves displayed in figure 6.4 confirm this phenomenon.

We next discuss an interesting observation. Notice again that the model's relaxed accuracy on D_1 is 26.8%. In expectation, a quarter of all problem matrices in D_2 contain one rule. Now assume that we ran inference with the model trained with D_1 on D_2. Even if the model did not get any sample with more than one rule right, it's relaxed accuracy would still be around 6.7%, that is, more than twice as much as the relaxed accuracy we achieve by training the model on D_2 itself! Thus, training the model with data from the simpler task seems to be a viable and interesting way of approaching the generative task.

Finally, we report the regular accuracies attained by the model on both datasets. For D_1, the model was only able to generate 2 panels which correctly complete the problem matrix, out of the 2560 samples in the test set. For D_2, the model was not able to generate a single panel that correctly completes the problem matrix. We next discuss why this is the case.

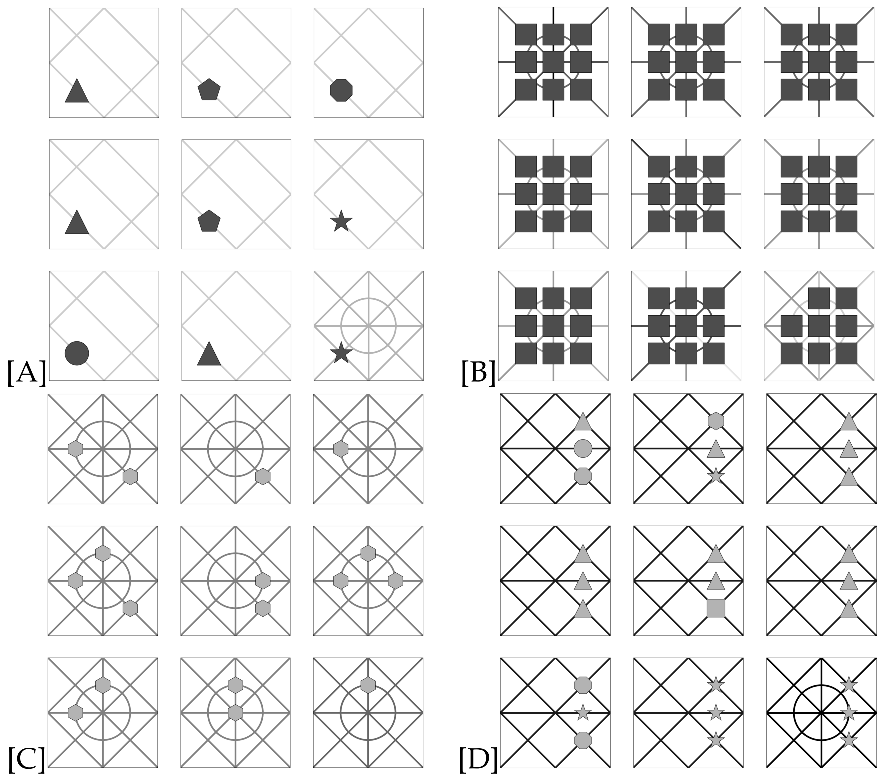

Figure 6.5: Four problem matrices. In each matrix, the last panel is generated by our model. A: The matrix is governed by (shape, type, progression). The model's answer is accepted by the relaxed accuracy measure, since a star has more corners than a triangle. B: The matrix is governed by (line, color, and). The model's answer is accepted by the relaxed accuracy measure, since the colors of the lines that appear in the generated panel are exactly those which appear in both grid cells of the last row. C: The matrix is governed by (shape, position, xor). The model's answer is rejected by the relaxed accuracy measure, since the set of occupied positions is not the xor of the sets of occupied positions of the two grid cells of the last row. D: The matrix is governed by (shape, type, and). The model's answer is accepted by the relaxed accuracy measure, since only stars appear in both grid cells of the last row. Note how in all problem matrices, our model chose to include all possible lines. In all examples except C, this causes the model's answer to be rejected by the regular accuracy measure.

In our experiments, we determine that the model has a great tendency to include all possible lines in the generated panel. Figure 6.5 provides some examples. We don't know why this is the case, but it degrades both the model's relaxed accuracy and its regular accuracy.

This bias is much probably not the only reason for the subpar accuracy. We expect the simplistic architecture to also limit the model's capacity.

Chapter 7

Potential Improvements & Future Directions

In this chapter, we discuss ways to improve our models and a possible strategy to attack the harder problem of generating the 9th panel.

7.1 Improving the Classifier

One could implement the two following suggestions to possibly attain a better classifier:

1. Instead of using one big neural network, one could partition the task of inferring 29 rules in some meaningful way. One can then train a different model for each child classification task, and concatenate all results to obtain the final prediction. For example, one could train one network to recognize rules on the colors of shapes, another network to recognize rules on the types of lines and so on. While this approach could boost performance, it focuses too much on solving the problem. Since the real goal is to understand the *reasoning abilities* of a neural network, we believe that it is much more beneficial to train a single network and thus force it to learn a representation that generalizes to predict all 29 rules correctly.

2. Encoding of the problem matrices in a one-hot scheme results in very sparse and very large representations. It forces our models to have too many learnable parameters just to be able to take the problem matrix as input. We believe that alternative encoding schemes such as embedding the matrices in a low dimensional space could aid in reducing the number of weights of our models and thus make them more data efficient.

7.2 Improving the Generator

1. Clearly, getting rid of the Generator's bias towards including all lines in the background would immediately improve the accuracy. However, we suspect this to be a technical issue, and thus find it uninsightful.
2. We believe that the Generator could also benefit from a proper embedding of the problem matrices for the same reason as explained above.
3. The more interesting issue is the struggle of the model in combining multiple rules. As our experiments indicate, curriculum learning seems to be a promising direction in tackling this issue.
4. One could attempt to model the combinatorial nature of combining rules directly in the model's architecture. Designing the network appropriately could guide it towards this inductive bias. A similar idea has been pursued in [24].

7.3 How to generate without using rules

In the absence of rule vectors to train our models with, there are two main ways to tackle the generative task:

1. Disregard the fact that problem matrices are governed by rules, and try to learn the mapping directly.
2. Take into account the causal relationship between a latent variable (the rule vector) and the observed problem matrices and try to learn the latent variables.

We believe that the second option is worth pursuing as it offers a principled way of approaching the problem and is much closer to the way humans go about solving the task.

To flesh it out some more, the idea here would be to apply clustering over the set of problem matrices. The appropriate loss function to get the clusters corresponding to rule combinations is not clear. However, this loss function should not be hand-crafted. It should rather somehow be learned from data, as the primary goal is still to understand if neural networks, or machine learning algorithms in general, are capable of abstract reasoning.

Chapter 8

Conclusion

This work showed how neural networks can be used to generate the 9th panel to the famous class of Raven's Progressive Matrices style visual psychometric tests, under the assumption that the rules that govern individual matrices are also provided to the model at training time.

We have shown that architectures with a Graph Neural Network component are highly effective at identifying the existence of rules, given an incomplete problem matrix. Our best models have attained 91.2% rule classification accuracy and 18.2% hard accuracy, respectively.

We have seen that even the simplest neural network offer high potential at generating the 9th panel when they are constrained to the class of problem matrices that are governed by one rule. It remains a challenge to combine multiple rules and to hold attributes that aren't governed by any rule to a constant.

At this point in time, it seems that neural networks are not the silver bullet that they may appear to be. Great care needs to be taken in both their architecture and the characteristics of the datasets with which we train them. Algorithmic improvements seem necessary to enable neural networks to autonomously reason.

Bibliography

[1] Samuel Dodge and Lina Karam. A Study and Comparison of Human and Deep Learning Recognition Performance Under Visual Distortions. *arXiv e-prints*, page arXiv:1705.02498, May 2017.

[2] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[3] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *arXiv e-prints*, page arXiv:1910.10683, October 2019.

[4] Andrew Trask, Felix Hill, Scott Reed, Jack Rae, Chris Dyer, and Phil Blunsom. Neural Arithmetic Logic Units. *arXiv e-prints*, page arXiv:1808.00508, August 2018.

[5] François Chollet. On the Measure of Intelligence. *arXiv e-prints*, page arXiv:1911.01547, November 2019.

[6] Wayne Weiten. *Psychology: Themes and Variations*. Cengage Learning, 2017.

[7] J.C. Raven. *1938*. Western Psychological Services, 1938.

[8] R Snow, Patrick Kyllonen, and B Marshalek. *The topography of ability and learning correlations*, pages 47–103. 01 1984.

[9] Strategies for how to solve ravens matrices iq problems. https://www.highiqpro.com/solve-matrices-iq-problems.

[10] David G. T. Barrett, Felix Hill, Adam Santoro, Ari S. Morcos, and Timothy Lillicrap. Measuring abstract reasoning in neural networks. *arXiv e-prints*, page arXiv:1807.04225, July 2018.

[11] Chi Zhang, Feng Gao, Baoxiong Jia, Yixin Zhu, and Song-Chun Zhu. Raven: A dataset for relational and analogical visual reasoning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[12] Chi Zhang, Baoxiong Jia, Feng Gao, Yixin Zhu, Hongjing Lu, and Song-Chun Zhu. Learning Perceptual Inference by Contrasting. *arXiv e-prints*, page arXiv:1912.00086, November 2019.

[13] Dokhyam Hoshen and Michael Werman. IQ of Neural Networks. *arXiv e-prints*, page arXiv:1710.01692, September 2017.

[14] Xander Steenbrugge, Sam Leroux, Tim Verbelen, and Bart Dhoedt. Improving Generalization for Abstract Reasoning Tasks Using Disentangled Feature Representations. *arXiv e-prints*, page arXiv:1811.04784, November 2018.

[15] Sjoerd van Steenkiste, Francesco Locatello, J. Schmidhuber, and Olivier Bachem. Are disentangled representations helpful for abstract visual reasoning? *ArXiv*, abs/1905.12506, 2019.

[16] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. *arXiv e-prints*, page arXiv:1710.10903, October 2017.

[17] Patricia A. Carpenter, Marcel A. Just, and Peter Shell. What one intelligence test measures: A theoretical account of the processing in the Raven Progressive Matrices Test. *Psychological Review*, 97(3), 1990.

[18] Travis E Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett,

editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[20] Matthias Fey and Jan E. Lenssen. Fast graph representation learning withPyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[21] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[22] N. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *ArXiv*, abs/1609.04836, 2017.

[23] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2015.

[24] Adam Santoro, D. Raposo, David G. T. Barrett, Mateusz Malinowski, Razvan Pascanu, P. Battaglia, and T. Lillicrap. A simple neural network module for relational reasoning. In *NIPS*, 2017.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.