MLP Coursework 4: Solving Raven's Progressive Matrices with UTIQ

G45: s1449692, s1410016, s1443569

Abstract

We introduce UTIQ - an end-to-end differentiable architecture with a modular design for solving Raven's Progressive Matrices (RPMs). UTIQ consists of an autoencoder, a reasoning agent and an optional classifier. We explore our architecture's properties when applied to the problem of solving Sandia Matrices – a simplified RPM dataset better suited for current machine-learning architectures. We do an extensive study of our architecture's performance, with over 150 thousand different hyperparameter settings, and explore the difficulty of learning different types of problems. We find that simple feedforward networks with relatively few trainable parameters yield best performance both for our autoencoder and reasoning agent modules. We introduce a new classifier module to both increase interpretability and take advantage of previously unused information for learning, but that proves to perform worse than using no classifier. Our results are an accuracy of 57% on a held-out test set - a relative improvement of 18% over our previous result, which is still 20% shy of the goal we set for ourselves. In our conclusions, we present a discussion of the possible reason for this performance.

1. Introduction

As originally pointed out in the interim report, Intelligence Quotient (IQ) tests have been used in standardized testing for decades and their results are often considered to be a proxy for general human intelligence. When one is to try and compare artificial and human intelligence, a framework for quantifying the intelligence of humans, such as IQ tests, is a reasonable place to look first. In order to orient the reader, we again present the example problem from the interim report in 1.

In order to avoid future confusion, we now define Transformed IQ (TrIQ) as a measure of performance for our models. Using the IQ metric that is usually applied to humans is questionable, and can be seen as controversial in this situation, as the general opinion is that humans and machines cannot be compared easily. What is more, unlike our system, humans don't learn how to do IQ tests from repeatedly doing IQ tests. Thus TrIQ is to serve as a general measure of how well an artificially intelligent agent performs on tasks that would measure IQ in humans. In this report, for simplicity, we will assume that TrIQ scales linearly with the number of problems solved, independent of what the problems actually are.

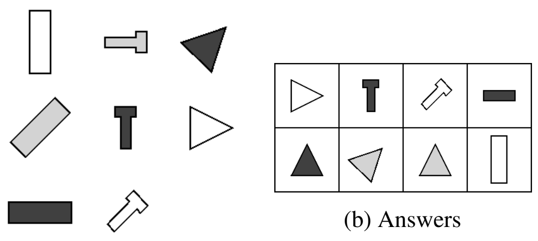
We restate the objectives from our previous report and point to relevant sections. Our main goal for this report was to create an end-to-end differentiable architecture – this was achieved in Section 2.9 and the results are presented in Section 3.2. Our stretch goal was to achieve an accuracy over 78.7%, in order to become state-of-the-art (SotA) – this goal was not met and Section 3.4 gives insight into why. An interesting goal we mentioned in the previous report was that we plan on exploring the consequences of using no encoding structure in our architecture – Section 2.6 describes the details, and the experimental results can be found in Section 3.1. Finally, we introduced a new objective for ourselves, namely, assessing the feasibility of the classifier module, which we introduce in Section 2.8.

2. Methodology

2.1. Dataset

In Section 3 of out interim report we describe the dataset we use. We have made no qualitative changes to it, but have made a quantitative one. In our interim report we described a 90-10 data split (90% of the data is used for training and 10% is held out for testing), which we have changed. We test a lot of models with large hyperparameter spaces in this report, so we run into the major risk of overfitting our test set. This is why we decided to train on 80% of the data and use 10% of it as a validation set. The number of entries for each question type is given in Table 1. Please refer to Section 3 of the interim report for examples of each question type.

Splitting the dataset this way means that we have less data to train on. This means that in order to judge the performance of new models fairly, we should update our baseline



(a) Question

(b) Answers

Figure 1. A sample question - answer pair. The correct answer label for this example is (6) - the gray triangle standing upward.

Type	1-rel	2-rel	3-rel	Logical	**Total**
Total	96	184	500	60	**840**
Train	76	148	400	48	**672**
Validation	10	18	50	6	**84**
Test	10	18	50	6	**84**

Table 1. Number of examples for each question type

results to reflect this fact. The old baseline was obtained by training on 90% of the data and testing on the test set. The new one is obtained by training on 80% of the data and reporting the accuracy on the newly separated validation set. These results, along with those for a theoretical agent that randomly guesses the correct answers, are given in Table 2.

Type	1-rel	2-rel	3-rel	Logic	**Overall**
Old	40%	56%	54%	50%	**52%**
New	40%	56%	50%	50%	**50%**
UR	12.5%	12.5%	12.5%	12.5%	**12.5%**

Table 2. Performance on our baseline system on the old test set (Old) after being trained on 90% of the data and on our new validation set after being trained on 80% of the data (New). This is compared to the expected performance of a theoretical uniform random agent (UR) that makes random guesses for the right answer. Changes in performance between the old and new baseline are highlighted in red.

2.2. Randomized hyperparameter search

As our system has a big number of hyperparameters, it is not computationally feasible for us to explore the whole of the hyperparameter space using grid search. This is why we opted for using randomized search. Randomized search requires we specify ranges for each of our hyperparameters and repeatedly sample uniformly at random from the n-dimensional hyperrectangle defined by these ranges. This approach is found to be more efficient than grid search for large hyperparameter spaces (Bergstra & Benigo, 2012).

In our implementation of randomized search, we read in a set of values for each parameter and sample from the set that is the result of taking the Cartesian product of these sets. A sample `.json` file, with some of the values we consider for each hyperparameter is given in Appendix A.

In Section 3, when we perform experiments, we mention the hyperparameters that produced each model we consider. Here we list the general training hyperparameters and the ranges we sampled values from:

- Learning rate – $[10^{-2}; 10^{2}]$
- Momentum – $[0; 10]$ – we use SGD with Nesterov momentum (Ilya Sustkever & Hinton) in our experiments
- L2 regularization coefficient – $[0; 10]$
- Batch size – $[1; 64]$

2.3. Early stopping

After performing several initial experiments, we found that our models often suffer from overfitting. One approach we took to prevent this is to perform early stopping – during training we keep track of the best parameters for validation accuracy, the best validation accuracy and the epoch they were measured. After each epoch, we check how many epochs have passed since we have seen an improvement over the best value stored. If we haven't seen an improvement for a certain the number of epochs, which is a tunable threshold we call *patience*, we terminate training, returning the parameters of the model that achieved the best validation accuracy.

2.4. Batch normalization

We experienced some divergent behaviour (in the form of NaN losses) and one of the contributors were the large activation values of the reasoning agent and classifier. We thus note the inclusion of the option of adding batch normalization (Ioffe & Szegedy, 2015) to the architecture's reasoning agent and classifier. With the batch normalization, these issues of divergence were severely reduced. We used the default PyTorch parameters for the BatchNorm1d layer - epsilon of 10^{-5} and momentum of 10^{-1}.

2.5. Training

We performed randomized search over the hyperparameter space, as described in Section 2.2. Models were trained for up to 100 epochs with early stopping (Described in Section 2.3) with a patience of 10 epochs. Validation accuracy was recorded after each batch and the early stopping criterion was checked after each epoch. The model with the best accuracy on the validation set was returned for each training session. We performed additional training for models that never met the early stopping criterion but that yielded no improvement.

2.6. Autoencoders

The autoencoders are responsible for producing encodings of the input that are then used by the reasoning agent. They are also used to reconstruct the reasoning agent's prediction of the encoding of the answer.

We briefly restate the interim report's design choice of using autoencoders. Our architecture makes use of autoencoders (or more concretely, an encoder and decoder pair, which are trained as an autoencoder, but are applied at separately throughout the architecture), to achieve two purposes. Firstly, autoencoders abstract away the size of the image and allow the rest of the architecture to deal with a fixed size representation. Secondly, our conjecture is that when we restrict the encoding size, the autoencoder is forced to learn useful features of the input image – this is backed by the results in 3.2. However, in order to truly challenge this design choice, we additionally implemented an "identity" autoencoder, which just flattens the input pixels into

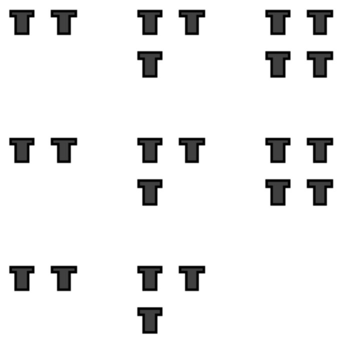


Figure 2. A sample question exhibiting temporal dependencies.

a one dimensional array. The results of this experiment can be found in Section 3.1. We used the autoencoders we previously defined – feedforward and convolutional autoencoders, as well as Principle Component Analysis (PCA).

2.7. Recurrent reasoning agent

The reasoning agent is the component in our architecture, responsible for taking the embeddings (a.k.a. encodings) of the question images produced by the autoencoder and outputting an embedding of the correct answer. We train our reasoning agent so as to minimize the mean squared error between their outputs and the embedding of the correct answer image, as produced by the architecture's autoencoder.

As outlined in our plan in the interim report, we implemented a reasoning agent using a Recurrent Neural Network (RNN). We previously hypothesized this can be beneficial as the questions can be viewed as a time series – for instance, the problem in Figure 2 is a simple counting task, in which there is a clear temporal dependence. However, this intuition failed us, as will be apparent in Section 3.2. In our experiments, we used 1-layer Vanilla RNN and Long Short-Term Memory (LSTM) networks with *tanh* nonlinearity - the poor performance they exhibit deterred us from pursing recurrent architectures with more layers. Additionally, we originally planned on using the SkipLSTM architecture proposed in the interim report, but the results in Section 3.2 suggest that LSTMs do not adequately capture the types of relations between images we want to leverage. We note this is in agreement with the constructive feedback we received on our previous report.

2.8. Classifier

The classifier is a new part of the architecture that we wanted to explore. For the interim report, we used an Mean-Squared Error (MSE) loss to estimate the distance between our reasoning agent's prediction and the correct answer's encoding using our autoencoder. While this was useful for a baseline, there were some improvements that we thought we could make. The biggest issue with that implementation was that we only used the correct answer to learn. There is a missed opportunity there, since we have a lot more negative examples – for every correct answer, we have an additional 7 wrong answers which we did not use. Therefore, we constructed what we call the **Pairwise** classifier.

This classifier works with the encodings of each possible answer (both right and wrong) produced by the autoencoder. Given a prediction from the reasoning agent, it runs each pair (prediction, answer) through the same network (which we implemented as feedforward for this report), producing a single number. It is important to note that the weights, associated with processing the 8 correct answers' encodings are reused. Thus we get 8 numbers, which we treat as unnormalized log-probabilities for a cross-entropy loss with the one-hot encoded version of the number corresponding to the spot of the correct answer. This is done in order to deal with the fact that the answers are randomly shuffled in each question. That is, there is no useful information in the position of the correct answer – and so, concatenating all answer encodings wouldn't work. Another advantage of this classifier is that it is more interpretable – it produces a probability distribution over the possible answers, which we can examine and gain an insight into the model's confidence in its predictions. Unfortunately, as seen in 3.3, the classifier did not improve, and actually decreased the validation accuracy.

2.9. UTIQ

Our general architecture consists of an encoder, reasoning agent and (an optional) classifier, each of which is described above. We wanted to do experiments on each of those independently, which is why we created the UTIQ architecture. During initialization, UTIQ allows us to mix and match any three components (or even forego the classifier). It then allows us to pre-train or freeze parts of the architecture at our convenience. Finally, our training procedure captures the model's setting, allowing it to calculate only the appropriate losses, which we describe below. This allows us to fully automate experimentation through the randomized hyperparameter search described in Subsection 2.2.

The losses awe use in training are as follows

1. Reconstruction loss - MSE loss between input image and autoencoder reconstruction. Used during pretraining or if the autoencoder is not frozen.
2. Latent prediction loss - MSE between the reasoning agent's latent prediciton and the correct answer's latent encoding. Used if no classifier is present.
3. Categorical loss - Cross-entropy loss between the logits output by the classifier and a one-hot encoded version of the number corresponding to the spot of the correct answer.

These losses are accumulated as appropriate and used to train the network. We considered adding a hyperparameter that weights each of the terms in the loss, but were unable to do it, due to time constraints.

3. Experiments

In order to make an attempt to reach both our objectives, it was sufficient for us to utilize UTIQ to run random experiments in the hyperspace of every possible combination of modules and hyperparameters for each module, as described in Section 2.2. However, we felt we stood a better chance of finding good hyperparameters for each module by first keeping everything but that module the same as the baseline. Once that was done, and we had approximations of the types of hyperparameters that worked for each module, we proceeded to run extensive experiments on any combination of modules we found reasonable. We next present the results of these experiments.

3.1. Autoencoder

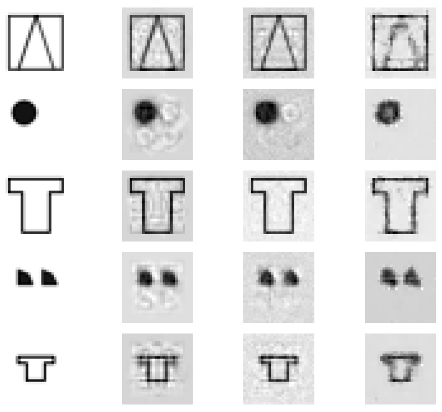


Figure 3. Reconstructions of images with each type of autoencoder. Leftmost is ground truth (or identity autoencoder), second column is PCA, third column is feedforward, last column is convolutional. The convolutional autoencoder visibly performs worse than the others.

This set of experiments were crucial for our objective of making our architecture end-to-end differentiable. We needed to swap the PCA autoencoder with a differentiable one. We also wanted to perform experiments with the identity autoencoder (defined in Section 2.6), in order to confirm our assumption that we need an autoencoder.

The best results of over 30 000 architectures with different hyperparameters are found in Table 3. The best agent (dubbed Agent 139) achieved a validation accuracy of 66.66%. This is the best differentiable agent we were able to create, but not the best overall, as seen in Section 3.2.

We see that feedforward autoencoders seems to perform the best. We attribute this to the fact that the number of data points we have is small and the pictures we train on mostly consist of first and second order features (like edges and corners) – things that the identity autoencoder cannot capture adequately in its latent space (of pixels). On the other hand, ConvNets bring value only when dealing with higher-order features (low order features in ConvNets only have access to their local spatial group). This is evident in Figure 3, where it can be seen that the ConvNets do not

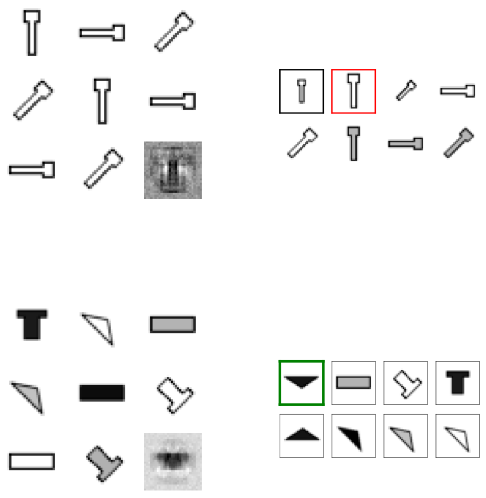


Figure 4. Agent 139's range of answers **Top**: Problem where the agent has the biggest uncertainty **Bottom**: Problem with best reconstruction. In the top figure, the image outlined in red is the correct answer and the one outlined in black is the agent's prediction.

capture edges adequately. It is worth noting that the feedforward reconstructions preserve edges better than PCA. This suggested that a pre-trained frozen feedforward autoencoder could produce better results than using PCA, but we weren't able to justify this intuition, as the results from Section 3.2 show.

As we can see from the **Top** example in Figure 4, even our best differentiable model can sometimes make arbitrary predictions that seem unintuitive. We convinced ourselves that the **Top** problem in the picture is one that had no similar ones in the training set and thus the architecture was not able to rely on previous experience to generalize to this problem. The bottom problem, however, is one we found impressive - the architecture is able to predict the correct shape, size, orientation and color of the answer to the problem, which would take a human a couple of seconds to solve. The full 84 results of our validation set are available in Appendix B.

3.2. Reasoning Agent

These sets of experiments were initially aimed at achieving our goal of beating the SotA results on the Sandia matrices. Our conjecture was that the biggest gain in performance was to be had by improving the reasoning agent so we wanted to isolate it from the influence of other trainable parameters. Additionally, as previously mentioned, doing this allowed us to get an idea of the regions in the hyperparameter space for the reasoning agent that produce adequate results. Since we failed in improving SotA, we will focus on some interesting properties of the models we produced.

For this set of experiments we fixed the reasoning agent to be the only differentiable component in our architecture. We used a PCA autoencoder to produce embeddings and did

Enc. size	AE hidden size	RA layers	RA hidden size	Batch size	Learning rate	Momentum	Val. acc.
250	200	1	300	64	0.30	0.5	66.66%
100	100	2	150	8	0.05	0.3	65.47%
150	200	1	200	8	0.05	0.0	64.28%
200	30	1	150	64	0.30	0.3	63.09%
200	100	1	200	64	0.30	0.1	63.09%

Table 3. The results and hyperparameters for our best autoencoders. All autoencoders present in the table are feedforward, with one hidden layer - the feedforward consistently outperformed the identity and convolutional autoencoders, which at best achieved 57.31% and 59.45% respectively.

Enc. size	Num. layers	Hidden dim.	Activation	Batch size	Learning rate	Momentum	Val. acc.
50	1	100	relu	64	0.05	0.1	**70.23%**
100	1	250	relu	32	0.10	0.2	69.04%
50	1	100	relu	64	0.10	0.5	69.04%
250	1	150	relu	4	0.05	0.5	67.85%
100	1	300	relu	32	0.10	0.1	67.85%

Table 4. The results and hyperparameters for our best feedforward reasoning agent combined with a PCA autoencoder and no classifier.

not use a classifier. We use the same logic as in our baseline in Coursework 3 – we chose the answer with embedding closest to the embedding produced by the reasoning agent. The distance metric used was MSE. We experimented with feedforward and RNN setups for our reasoning agent. The network hyperparameters we considered and the ranges from which we sampled values for each are listed below:

- Number of hidden layers – between 1 and 5 – only for feedforward networks
- Hidden layer size – between 20 and 500 – For both RNN and feedforward
- Activation function – ReLU or Sigmoid – for feedforward only
- The recurrent network type – Vanilla RNN or LSTM

We ran over 80 000 experiments - sampled from a state space of 5 million possibilities. The best results for feedforward and RNN are tabulated and presented in Table 4 and Table 5 respectively. As initially conjectured, we saw a big leap in performance from just changing this single module - the biggest performance increase across all three modules. We note that both types of networks show an improvement – the best feedforward agent has a 40% relative performance gain over our baseline on the validation set, and the best RNN agent – 16% relative gain. We suspect that the better performance of the feedforward agents might be due to the fact that the amount of training data we have does not warrant the use of more expressive models such as RNNs. In the next few paragraphs, we try to justify that assumption, as well as present other useful conclusions we drew from this set of experiments.

The effect of embedding size on the validation loss Figure 5 shows validation accuracy plotted against validation loss for our feedforward reasoning agent experiments. Each data point represents the results for a given setting of the hyperparameters. The size of the points is proportional to the number of trainable parameters in the network and their color is given by the encoding size.

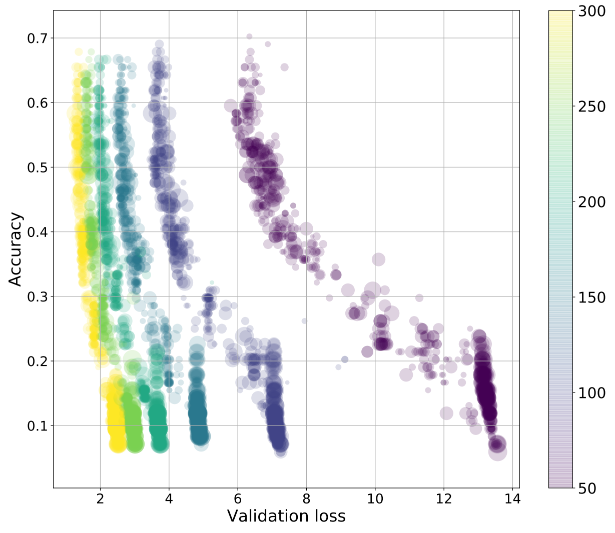


Figure 5. Validation accuracy plotted against validation loss on the x-axis for all of the feedforward reasoning agent trials. Color indicates the embedding vector size and the size of the points on the plot is proportional to the number of hyperparameters in the network.

A very discernible trend on this figure is for the validation loss to be smaller for bigger encoding sizes – this is to be expected, as more information is preserved when encoding the images and the total loss is made up in part from the autoencoder reconstruction loss.

We can also see that there is a negative correlation between validation loss and accuracy. This makes sense – if we were able to reconstruct the correct answer better, that means that the latent prediction was closer to the actual embedding of the correct answer, which means it is more likely for our classifier to make a correct prediction. What in interesting

Enc. size	network_type	Hidden dim.	Batch size	Learning rate	Momentum	Val. acc.
50	vanilla	400	16	0.10	0.1	58.33%
100	vanilla	300	8	0.10	0.5	58.33%
50	vanilla	400	16	0.05	0.4	57.14%
50	vanilla	300	16	0.10	0.1	55.95%
150	vanilla	300	4	0.10	0.5	55.95%

Table 5. The results and hyperparameters for our best RNN reasoning agent combined with a PCA autoencoder and a classifier that chooses the minimum-Euclidean distance answer.

is that the correlation seems to be stronger for smaller encoding sizes – one possible explanation for this is that for smaller embedding sizes, the only way for our reasoning agent to decrease the validation loss is to learn something intrinsic about the problem. This intuition agrees with the view of machine learning working so well due to the series of information bottlenecks present in most architectures (Tishby & Zaslavsky, 2015).

On the number of hyperparameters and encoding size Figure 6 shows a heatmap of validation accuracy for different numbers of hyperparameters and different embedding sizes produced from the results of our experiments with the feedforward reasoning agents. The accuracy in each rectangle is averaged over all experiments with the given values for the two hyperparameters.

There is a noticeable trend for networks with less parameters combined with smaller encoding sizes to achieve higher accuracy (hence the lighter colors in the top-left corner of the heatmap). We attribute this to the small number of training examples we have in our training set – the use of deeper, wider network is usually only warranted in the presence of large amounts of data, which is not true in our case. We took these results into account for our other experiments and chose smaller networks and embedding sizes for our reasoning agent.

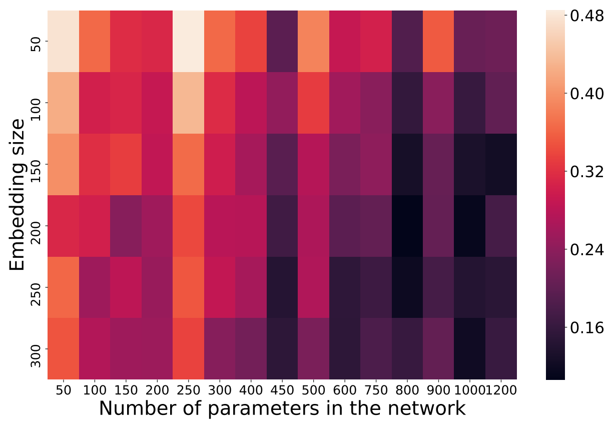


Figure 6. A heatmap of mean validation accuracy for feedforward agents of different sizes and different sizes of the embedding produced by PCA. Notice that smaller networks and smaller embedding sizes tend to result in better accuracy.

On the effect of L2 regularization We tried values for the L2 regularization coefficient ranging from 0 to 1. Only 3 of our top 100 feedforward models and only 5 of our top 100 RNN models had a regularization coefficient more than 0.05. This led us to the decision of not using values for this hyperparameter higher than 0.1 in subsequent experiments. We believe the reason why L2 regularization proves ineffective in this scenario is that we have a relatively small number of parameters in our networks, as noted above.

3.3. Classifier

This set of experiments were done to assess inclusion of the **Pairwise** classifier we defined in Section 2.8. We trained both models that use PCA to produce encodings and ones that use trainable autoencoders (both Feed-forward and convolutional). Following the conclusions in Section 3.2, for the reasoning agent, we only experimented with feedforward networks and we reduced the range of hyperparameter values we explore.

Our best model out of 40 000 architectures achieves an accuracy of 54.76% on the validation set, as seen in (Figure 8, Architecture **(B)**), which was on par with the baseline we initially established. The addition of the classifier allowed us to better assess the model's confidence in its answers – we present two examples in Figure 7. However, our results show that the benefit we gain from using the classifier does not outweigh the loss we incur from worse reconstructions. Notice that even the confident prediction at the **Bottom** of Figure 7 is not associated to a good reconstruction.

We believe that this is not indicative of a general faulty architectural choice, but rather due to the relative small amount of data we work with. Because of the restricted amount of data, we are not able to train very deep and complicated models. However, the classifier may need to have a deeper structure in order to properly learn a complex distance metric. We do, however, conclude that the results we received in this set of experiments are inferior to those received when we trained with no classifier. Our suggested way of remedying this is found in Section 4.1.

3.4. Test set results

Table 7 summarizes the results of our best-performing models, which we chose to evaluate on the test set. Agent 139 is the differentiable model using a feedforward autoencoder and a feedforward reasoning agent, which is described in Section 3.1 and achieves 67% accuracy on the validation set. The model referred to as *PCA best* in the table below is the best model from the experiments described in Section 3.2,

Enc. size	RA Hidden sizes	Activations	Classifier sizes	Batch size	Learning rate	Momentum	Val. acc.
200	200	relu	80, 50	8	0.05	0.5	54.76%
200	150	relu	80, 50	8	0.05	0.5	54.76%
200	200	relu	50, 30	8	0.10	0.1	54.76%
150	200	relu	50, 30	64	0.05	0.5	54.76%
150	150	relu	80, 50	32	0.05	0.5	53.57%

Table 6. The results and hyperparameters for our best models using the **Pairwise** classifier. All five networks use a PCA autoencoder and a feedforward one-layer and reasoning agent. The best validation accuracy achieved with a feedforward autoencoder was 50%.

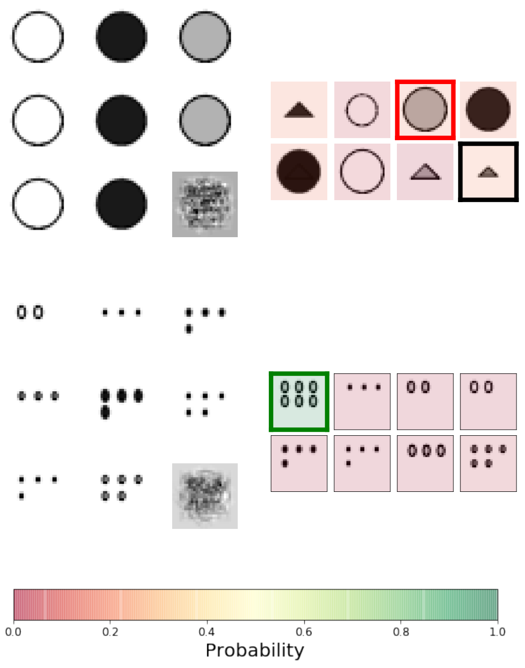


Figure 7. Examples of classifier prediction (represented by border) and confidence (represented by colour). Notice that the latent reconstructions no longer resemble the actual answer, as the latent input is fed into the classifier. **Top**: The classifier is really uncertain about its prediction (black border) and gets the answer (red border) wrong. **Bottom**: Our results from the interim report suggested that the models learn to count easily. These results confirm that, with the classifier putting a lot of probability mass on the correct answer (green border).

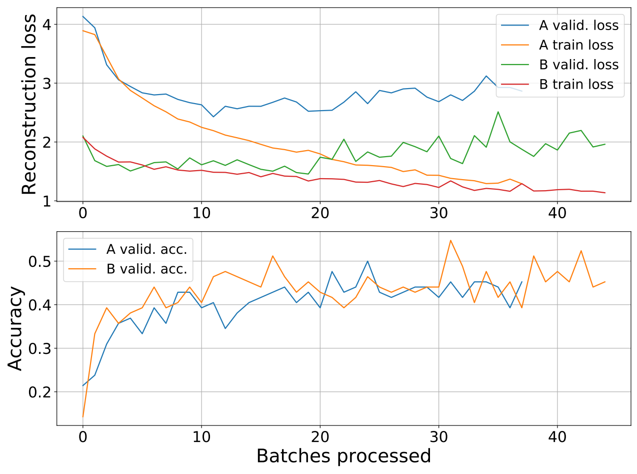


Figure 8. Learning curves of the best architectures that employ the pairwise classifier. Architecture **(A)** employs a feedforward autoencoder, while **(B)** uses PCA. Both are noisy due to the small dataset size, but there is a clear indication that learning comes to a halt early and thus further epochs would not improve validation performance.

Type	1-rel	2-rel	3-rel	Logic	**Overall**
Agent 139	50%	56%	54%	50%	**54%**
PCA best	40%	61%	58%	67%	**57%**
Baseline	40%	56%	40%	50%	**48%**
UR	12.5%	12.5%	12.5%	12.5%	**12.5%**

Table 7. Test set results for Agent 139 (best performing end-to-end differentiable model), the best system with PCA encoding (best overall system), and for our baseline system.

which achieves a validation accuracy of 70%. These are compared to our baseline, which achieves 50% validation accuracy and a theoretical agent making random guesses.

We can see that even though our best models greatly outperform the baseline on the validation set, the gap in test set performance is smaller. The most likely reason for this is that our validation set was too small and for this reason we have overfitted it – we evaluated a great number of models, which increased the probability of choosing hyperparameters that do well on the specific validation set. In future situations similar to this one, we will employ K-way cross-validation and suffer the computational complexity increase that is associated to it.

3.5. Comparison to human performance

Finally, to put these results into a grander context, we performed an in-house experiment with humans solving the problems in our test set. The mean accuracy for both sets was 89% with a standard deviation 2%. This is significantly higher than the performance of our models and even though we have made great improvements over random guesses, our model is still trailing far behind humans. To put this into perspective, assuming an average IQ of 115 among our human test subjects (which is suggested to be the requirement for adequate performance at university level), our best agent has a TrIQ of only 72. We hope our proposals for improvement in Section 4.1 are enough to increase this number to over 100.

4. Related work

Our work could be applied to the larger domain of image prediction. This is a popular topic in the literature – for example, video frame prediction as seen in (Babaeizadeh et al., 2017) deals in part with positional predictions of abstract objects. However, we do believe to be one of the first to tackle the problem of predicting answers to non-verbal IQ tests. As stated in our interim report, our project closely relates to work done by (Mekik et al.) and (Hoshen & Werman, 2017), which we revisit briefly here. We were unable to find any new work directly related to this problem published between this report and the interim one.

(Mekik et al.) use neural networks in combination with a symbolic agent to address the same task and dataset. However their approach is not fully differentiable - they use neural networks to solve only a small sub-problem of the task – they apply a ConvNet to encode information about the images and then proceed to classify them using a rule-based agent. In this report, we tried to match or beat their performance using an end-to-end differentiable model, but we were unsuccessful, as seen in Section 3.4.

(Hoshen & Werman, 2017) also use a neural network approach to tackle a subset of the problem. They take a sequence of two images and try to predict the third, with the shapes presented varying in shape, size, orientation and color. They achieve a test-set accuracy of 90% on this simplified problem. As this is an ever further simplification of RPMs than the one we explore in this report, we consider their results a stepping stone towards solving the problem posed by (Mekik et al.).

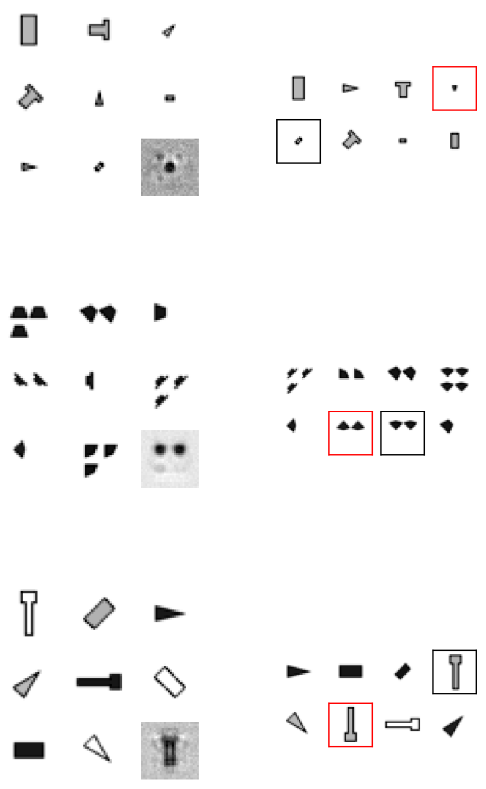


Figure 9. Types of mistakes our agents make. **Top**: Inability to cope with small size answers. **Middle**: Inability to cope with shape. **Bottom**: Inability to cope with rotation.

4.1. Future work

Figure 9 shows the types of mistakes that our agents consistently make. We suspect they mostly come from the network's inability to pick out the correct structure due to the limited size of the images we present as input. Thus, as a next step, we would produce a higher fidelity dataset and train our architecture on it. The bigger image size would also allow us to revisit ConvNets or even explore Capsule Networks (Sabour et al., 2017), which are specifically designed to deal with these types of problems.

There is a very clear divide between the types of problems that make up IQ tests. Therefore, we think a mixture of experts (Jordan & Jacobs, 1993) would be very well suited here. This would allow us to train agents which are optimized for each specific type of task and allocate problems of their respective expertise at test time. We think this approach is very likely to improve performance on this type of problem.

Our proposed classifier abstraction adds many advantages to the UTIQ architecture, as shown in 2.8. We would therefore explore alternatives to the **Pairwise** classifier presented in this report and consider the trade-off between the interpretability those alternatives offer and their potential drawbacks, potentially by introducing additional hyperparameters as suggested in Section 2.9.

5. Conclusions

The primary objective we set in the interim report was to build a fully differentiable model that performs better than a random agent on our test set. We not only met this goal, but improved on our non-differentiable baseline in the interim report. Additionally, we explored introducing more interpretability into our architecture, but failed to do so without a decrease in performance.

The second objective we set for ourselves was to improve over the test set accuracy of 78.7% reported in (Mekik et al.). We failed to meet this goal. There are three factors we believe contributed to this. Firstly, (Mekik et al.) do not mention the use of a validation set – this means their models were able to learn from more data, which might be crucial for performance, given the restricted size of the dataset. Secondly, the fact that they do not use a validation set might have led to them overfitting their test set. Lastly, our search of possible models might have not been extensive enough and we might not have chosen the most appropriate architecture to tackle the problem.

References

Babaeizadeh, Mohammad, Finn, Chelsea, Erhan, Dumitru, Campbell, Roy H., and Levine, Sergey. Stochastic variational video prediction. *CoRR*, abs/1710.11252, 2017. URL http://arxiv.org/abs/1710.11252.

Bergstra, James and Benigo, Yoshua. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 2012. URL http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

Hoshen, Dokhyam and Werman, Michael. IQ of Neural Networks. sep 2017. URL https://arxiv.org/abs/1710.01692.

Ilya Sustkever, James Martens, George Dahl and Hinton, Geoffrey. On the importance of initialization and momentum in deep learning. URL http://www.cs.toronto.edu/~hinton/absps/momentum.pdf.

Ioffe, Sergey and Szegedy, Christian. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, jun 2015. ISSN 1938-7228. URL http://proceedings.mlr.press/v37/ioffe15.html.

Jordan, Michael I. and Jacobs, Robert A. Hierarchical mixtures of experts and the em algorithm. 1993. URL http://www.cs.toronto.edu/~hinton/absps/hme.pdf.

Mekik, Can Serif, Sun, Ron, and Dai, David Yun. Advances in Cognitive Systems X (20XX) 1-6 Deep Learning of Raven's Matrices. URL http://www.cogsys.org/papers/ACS2017/ACS{_}2017{_}paper{_}23{_}Mekik.pdf.

Sabour, Sara, Frosst, Nicholas, and Hinton, Geoffrey E. Dynamic routing between capsules. *CoRR*, abs/1710.09829, 2017. URL http://arxiv.org/abs/1710.09829.

Tishby, Naftali and Zaslavsky, Noga. Deep learning and the information bottleneck principle. *CoRR*, abs/1503.02406, 2015. URL http://arxiv.org/abs/1503.02406.

A. Hyperparameter search settings

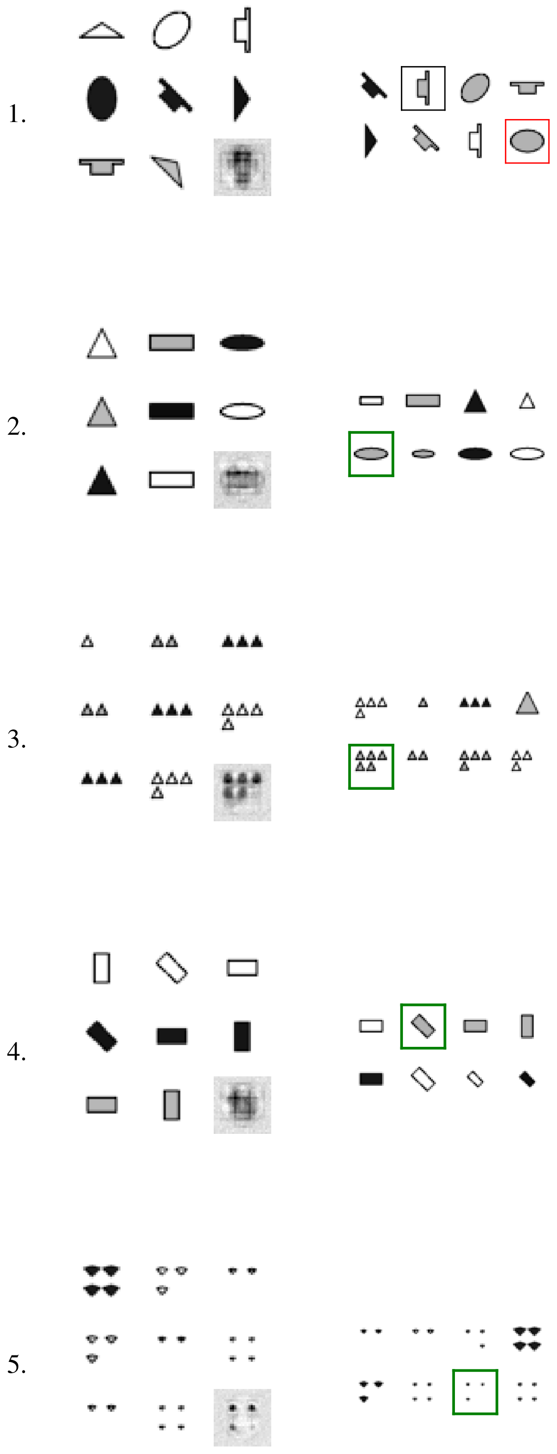
A non-exhaustive hyperparameter search setting is displayed in the listing below. We compute the Cartesian product of all these lists, producing a massive number of possible hyperparameter settings - in this case 6 967 296 possibilities. We then sample a small percentage of them - usually what amounts to 10k architectures.

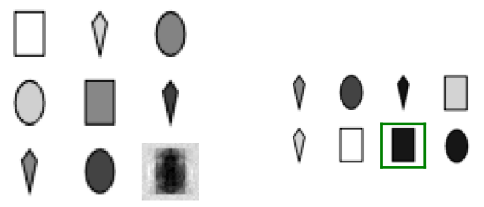
Listing 1 Example JSON file specifying a hyperparameter search space

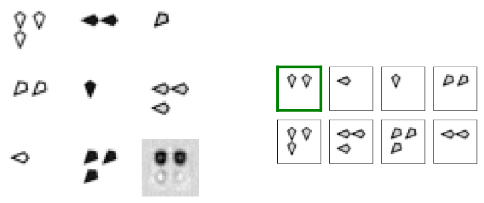
```
{
    "img_size": [28],
    "encoding_size": [20, 50, 100,
      150, 200, 250, 300, 400,
      500],

    "ae_type": ["ff", "conv"],
    "ae_hidden_sizes": [
      [200, 100, 50], [400, 200],
      [300, 200], [300, 100],
      [500, 200], [300], [200]
    ],
    "ae_pretrain": [false],
    "ae_freeze": [false],

    "ra_type": ["ff", "rnn"],
    "ra_hidden_size": [300, 200,
      100, 50],
    "ra_num_layers": [1, 2, 3],
    "ra_nonlinearity": ["relu",
      "sigmoid"],
    "ra_network_type": ["vanilla",
      "lstm"],
    "ra_use_batchnorm": [true],

    "clf_type": ["lse", "pw"],
    "clf_hidden_sizes": [[100],
      [200, 100], [100, 50]],
    "clf_nonlinearity": ["relu"],
    "clf_use_batchnorm": [true],

    "learning_rate": [0.05, 0.1,
      0.3],
    "momentum": [0, 0.1, 0.3,
      0.5],
    "weight_decay": [0, 0.05],
    "num_epochs": [60],
    "epoch_patience": [12],
    "batch_size": [8, 16, 32, 64]
}
```

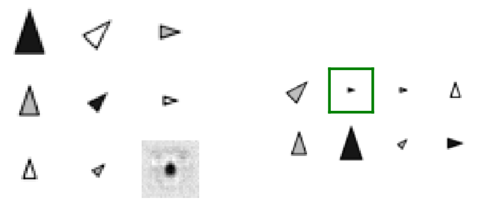
B. Results of Agent 139 on entire validation set

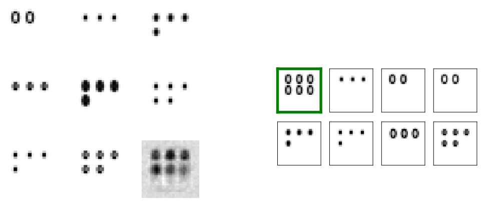
Below we present the results of our best agent on the entire validation set. The left column consists of the 8 setup images, with the reconstruction of the model's prediction filling in the blank 9th image. The right column is the 8 possible answers, with a black border around our prediction and a red border around the correct answer or a green border if our choice is correct.

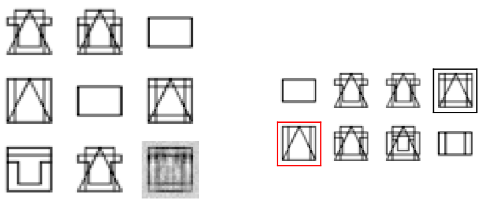


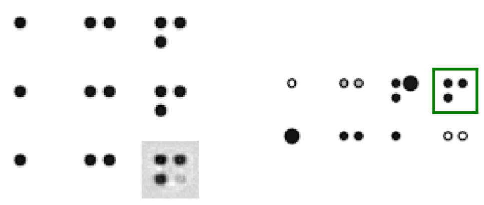
6.


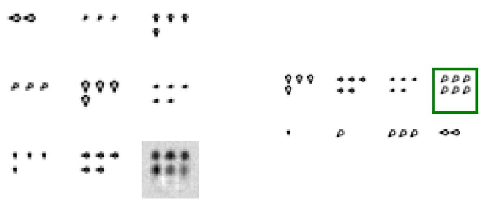
12.


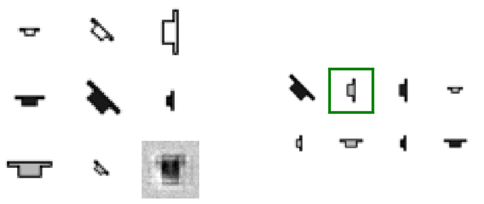
7.


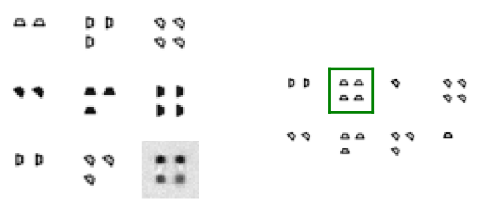
13.


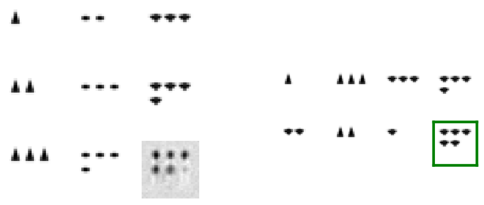
8.


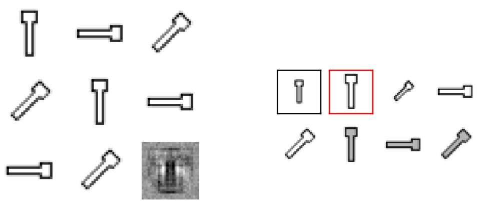
14.


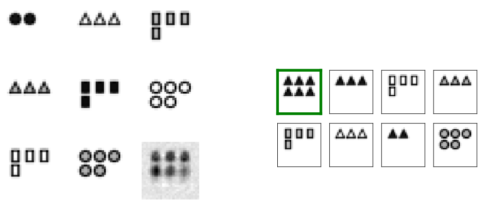
9.


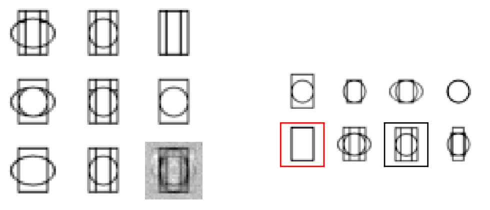
15.


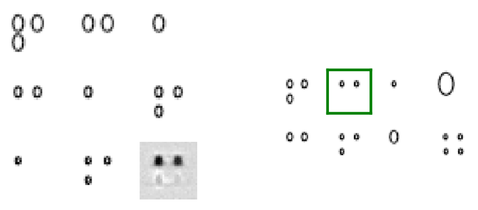
10.


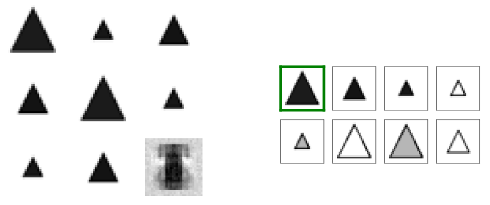
16.


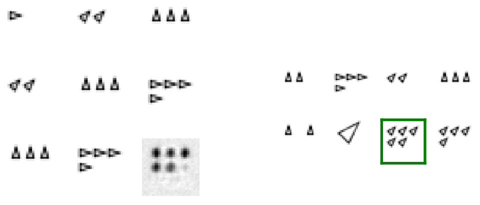
11.


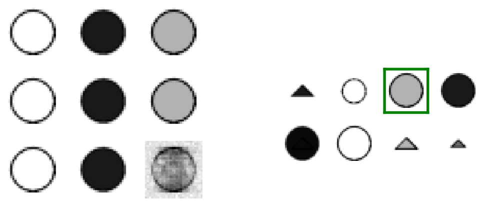
17.


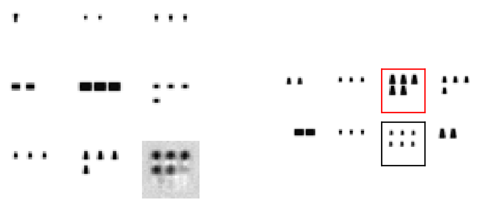
18. 

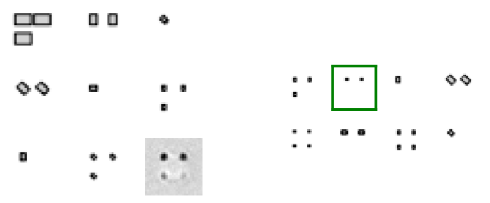
24. 

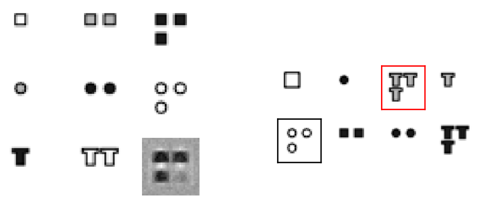
19. 

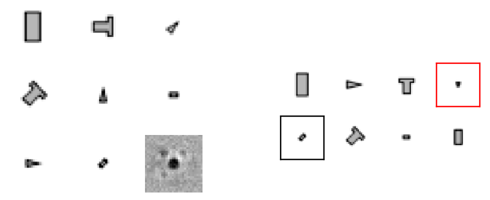
25. 

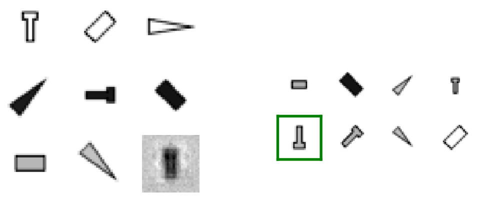
20. 

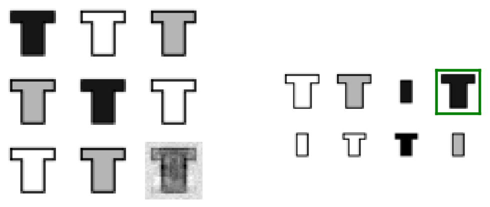
26. 

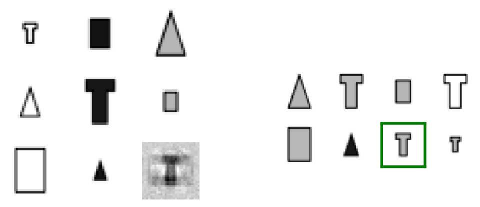
21. 

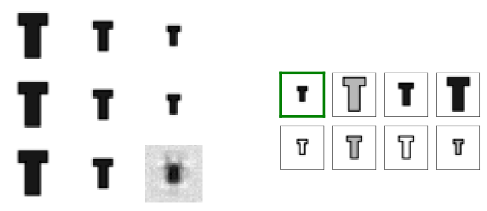
27. 

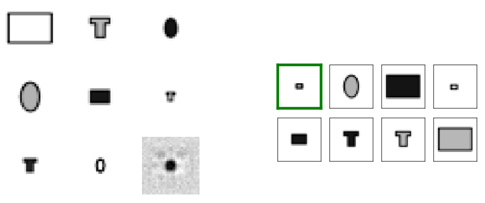
22. 

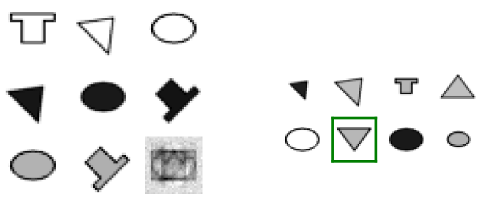
28. 

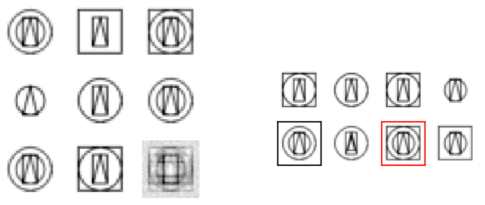
23. 

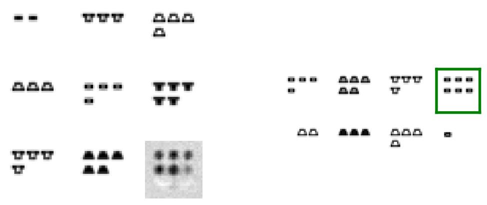
29. 

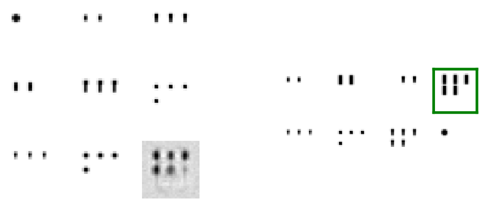
30.


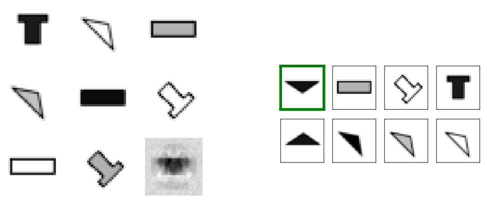
36.


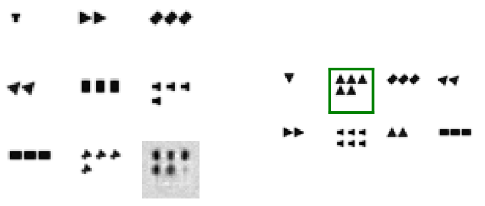
31.


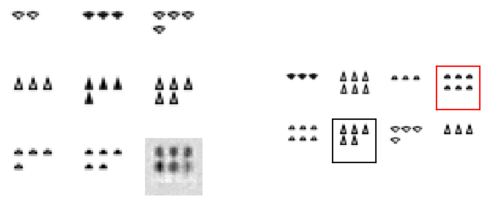
37.


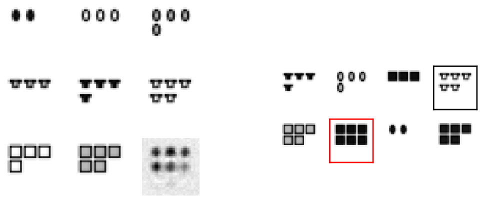
32.


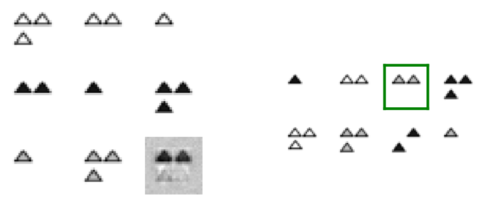
38.


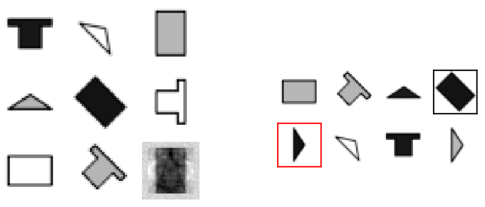
33.


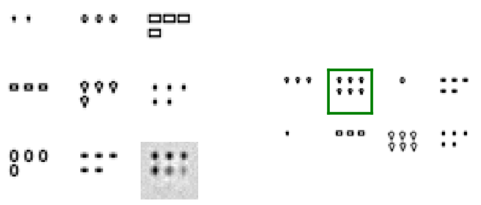
39.


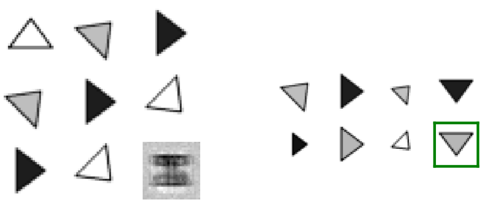
34.


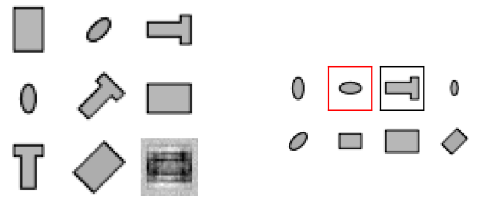
40.


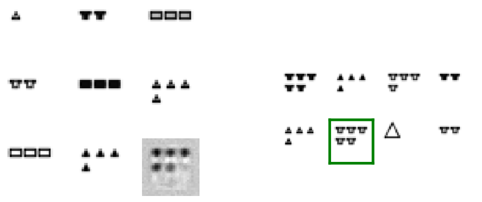
35.


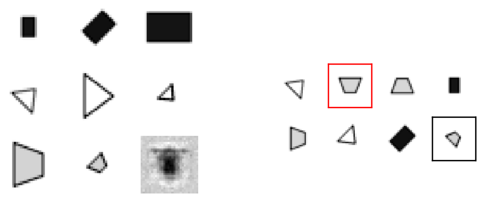
41.


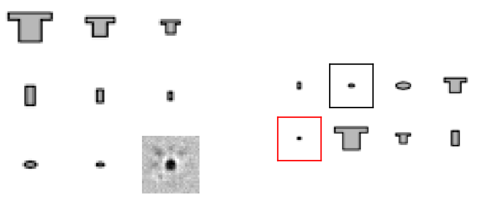
42.


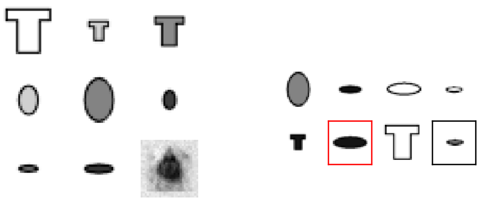
48.


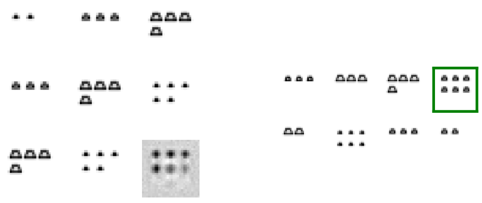
43.


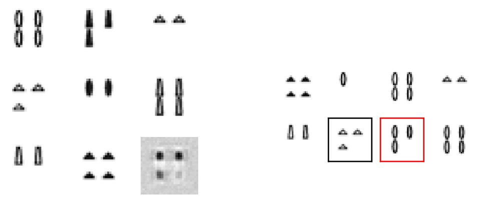
49.


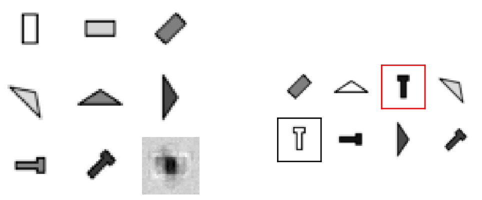
44.


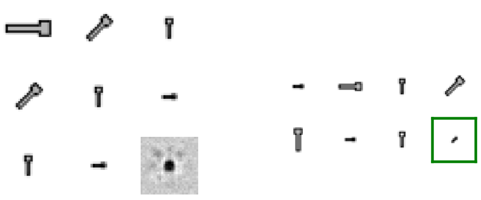
50.


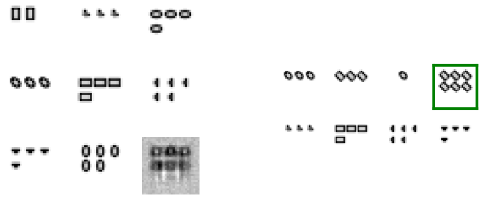
45.


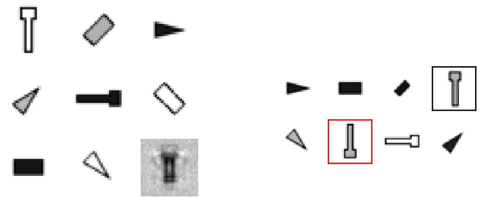
51.


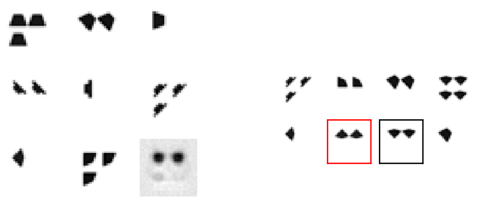
46.


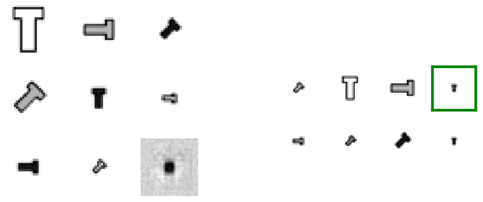
52.


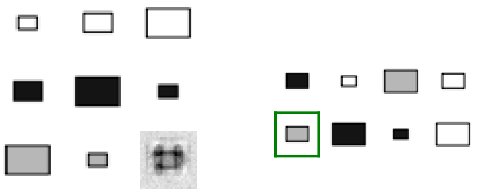
47.


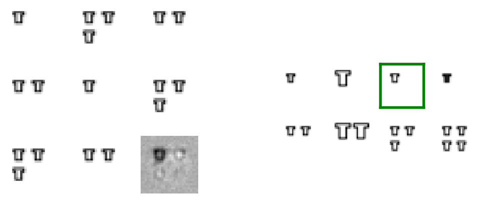
53.


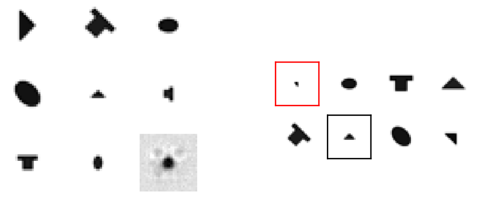
54.


60.


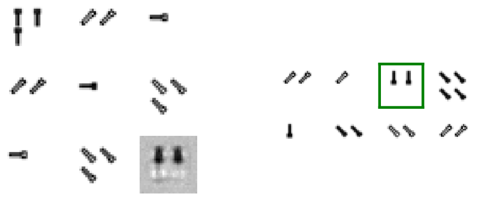
55.


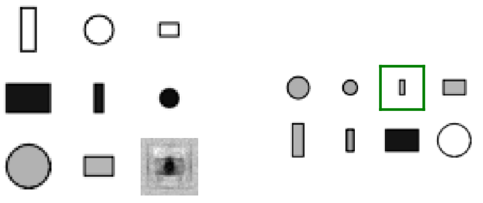
61.


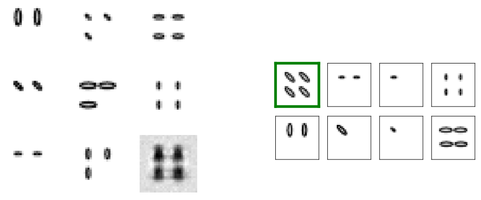
56.


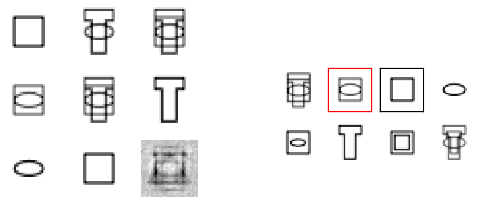
62.


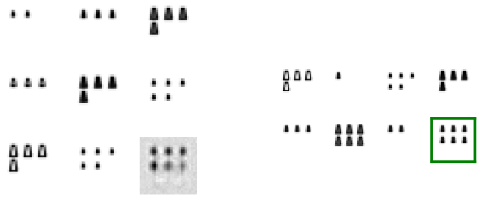
57.

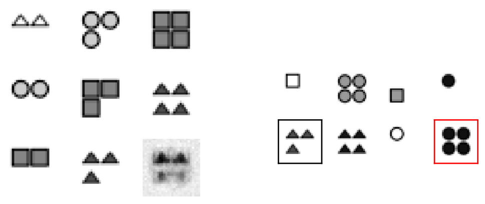
63.


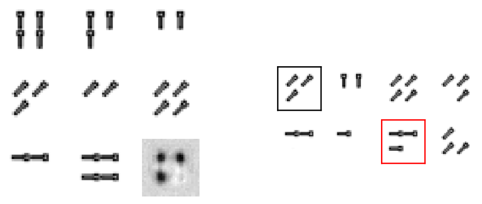
58.


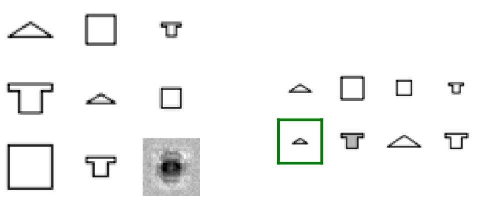
64.


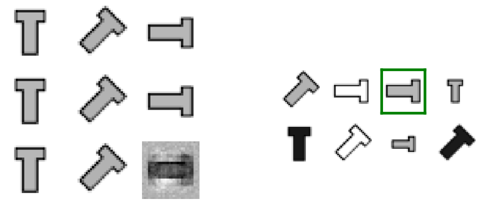
59.


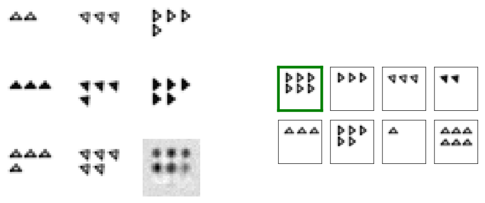
65.


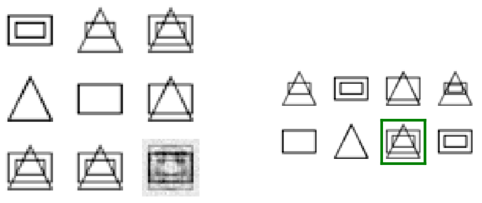
66.


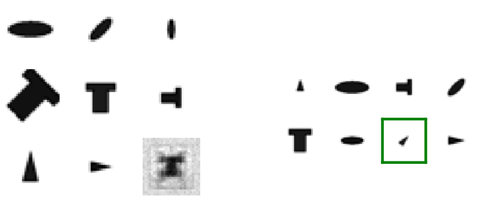
72.


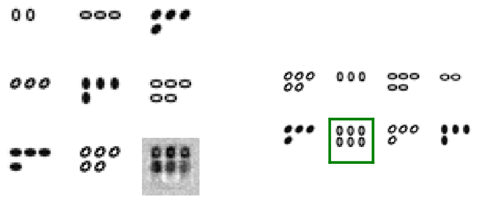
67.


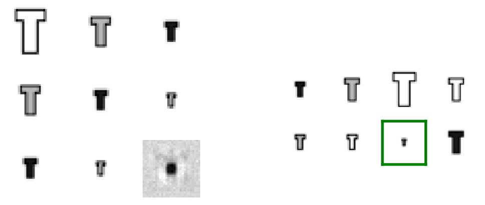
73.


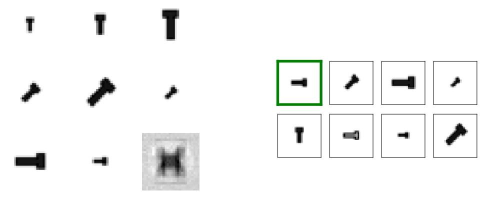
68.


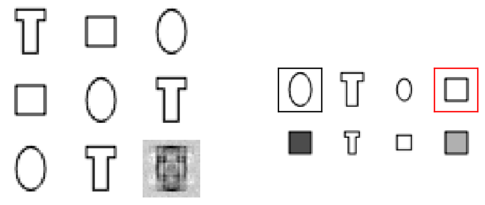
74.


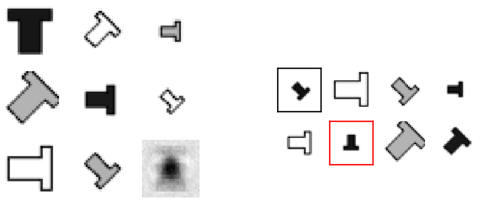
69.


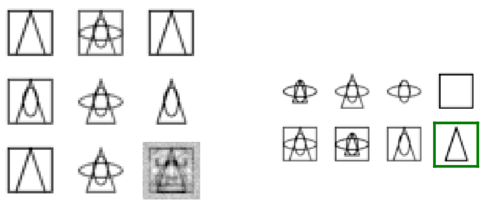
75.


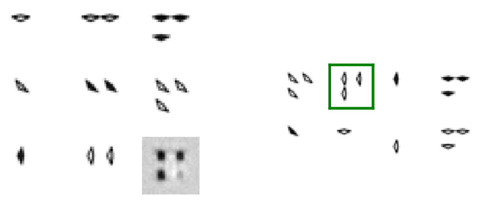
70.


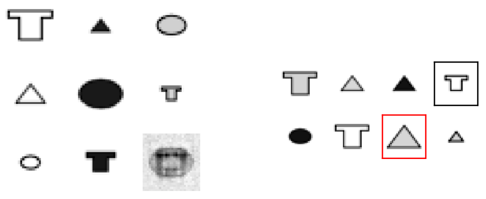
76.


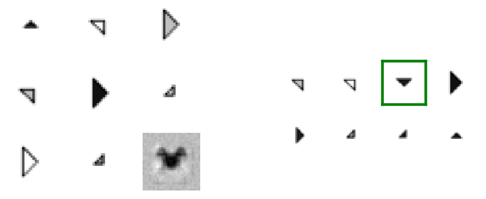
71.


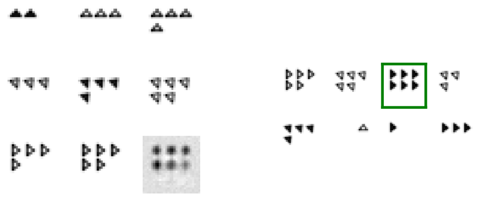
77.


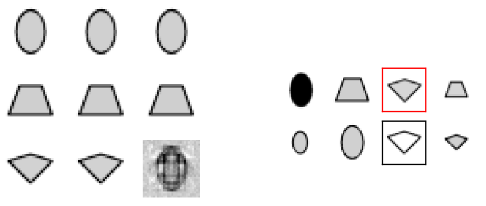
78. 

84. 

79. 

80. 

81. 

82. 

83.