Abstract and Reasoning Challenge: Final Report

John Bello

Introduction

The purpose of this project is to create a model capable of solving abstract tasks as defined in the Abstract and Reasoning Challenge posted by François Chollet on Kaggle.com.

This model will be given a task consisting of a set of two to ten training samples, in the form of a grid, and asked to learn a set of abstract rules. The model will then be given a test input grid and asked to create the expected output pattern. The model will be trained on 400 tasks and evaluated on 100 new tasks.

The thought process behind this competition is that, by creating a model of learning abstract patterns given very little training data, machine learning models would be better able to mimic human thought. This ability to mimic human thought would dramatically improve the range of tasks machine learning models are capable of as well as the speed at which they learn these new tasks.

Data Structure

The data is structured within three folders, a training folder, an evaluation folder, and a test folder. For the sake of this project, the test folder will be omitted. Within the training and evaluation folders are JSON files, 400 in the training folder and 100 in the evaluation folder. The structure of the JSON files are the same in each folder, shown below in Figure 1.

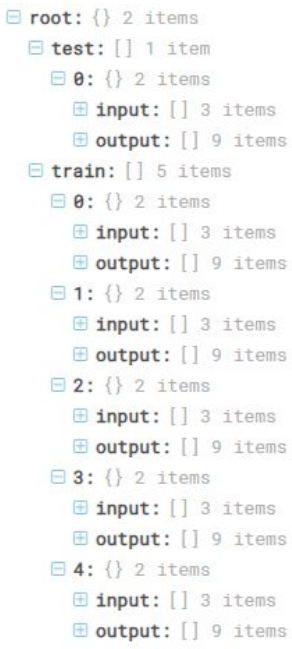


Figure 1. JSON structure.

Each file is separated into a train and test set. The train set has a varying number of input and output pairs while there is always only a single training input in the test set. The dimensions of the inputs and outputs vary between each task (JSON file).

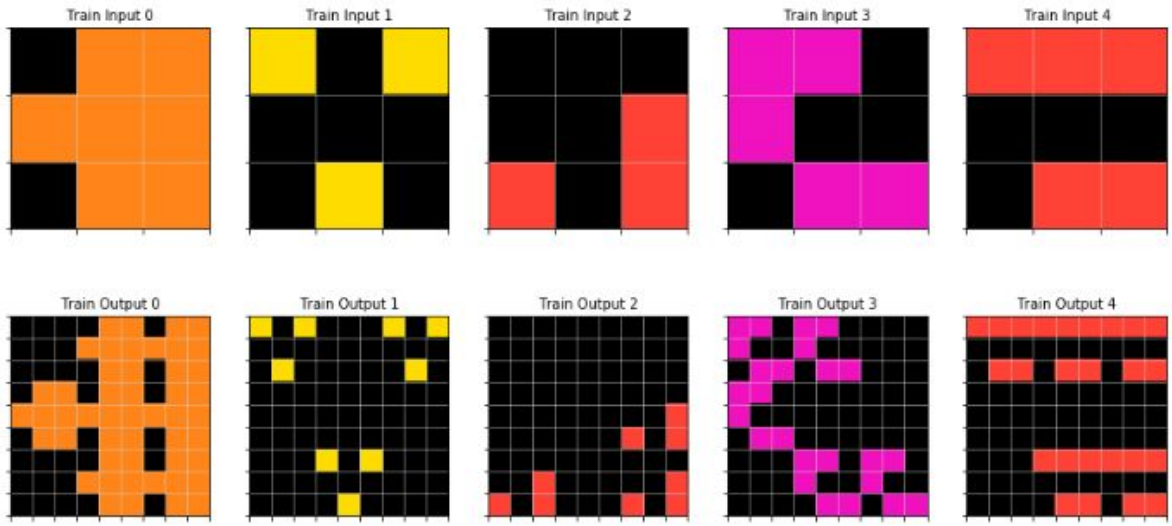
Data Wrangling

Because the data is taken from a Kaggle competition, the data is already well formatted. During this step, what was done was the data was read in, and plotted using the code below.

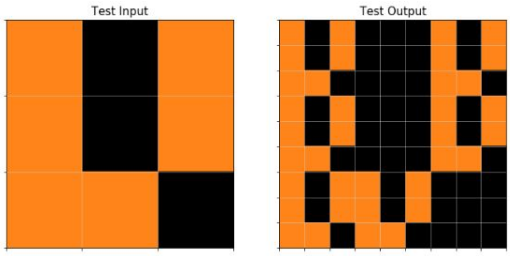
```
def plot_grid(grid, ax, title, size):
    ax.imshow(grid, cmap=cmap, norm=norm)
    ax.grid(True,which='both',color='lightgrey', linewidth=0.5)
    ax.set_yticks([x-0.5 for x in range(1+len(grid))])
    ax.set_xticks([x-0.5 for x in range(1+len(grid[0]))])
    ax.set_xticklabels([])
    ax.set_yticklabels([])
    ax.set_title(title, size=size)

def plot_task_pairs(pairs, task_type):
    if task_type == 'Test':
        title_size = 15

        pair = pairs[0]
        task_input = pair['input']
        task_output = pair['output']

        fig, axs = plt.subplots(ncols=2, figsize=(12, 7))

        plot_grid(task_input, axs[0], task_type + ' Input', title_size)
        plot_grid(task_output, axs[1], task_type + ' Output', title_size)

    elif task_type == 'Train':
        title_size = 10
        num_train = len(pairs)

        fig, axs = plt.subplots(ncols=num_train, nrows=2, figsize=(15, 7))

        for i in range(num_train):
            task_input = pairs[i]['input']
            task_output = pairs[i]['output']

            plot_grid(task_input, axs[0, i], task_type + ' Input ' + str(i), title_size)
            plot_grid(task_output, axs[1, i], task_type + ' Output ' + str(i), title_size)
```

Figure 2. Plotting Functions

The `plot_task_pairs()` function takes either the training set or testing set from a single JSON file and plots the grids according to the colors shown in the app used to visualize the data. This app is located at the github page of the competition(https://github.com/fchollet/ARC). The output for the first JSON file is shown below in Figure 3.



(a)



(b)

Figure 3. File: 007bbfb7.json
(a) Visualizations of training set. (b) Visualizations of testing set.

Additionally, during this step, a data frame was created containing summary statistics for each of the tasks in the training set. This data frame contains the following columns:

1. **task_label** : The JSON file identifier.
2. **num_train** : The number of input-output pairs in the training set of the specific JSON file.
3. **train_input_widths** : A list containing the widths of each input in the train set.
4. **train_input_heights** : A list containing the heights of each input in the train set.
5. **train_output_widths** : A list containing the widths of each output in the train set.
6. **train_output_heights** : A list containing the heights of each output in the train set.
7. **test_input_widths** : A list containing the widths of each input in the test set.
8. **test_input_heights** : A list containing the heights of each input in the test set.
9. **test_output_widths** : A list containing the widths of each output in the test set.
10. **test_output_heights** : A list containing the heights of each output in the test set.
11. **unique_colors** : A list containing the unique colors occuring in the train set.

The functions used to create this dataframe are shown in Figure 4 below.

```
def find_unique_colors(task_set, unique_list):
    for task in task_set:
        for io_set in task.values():
            result = set(x for l in io_set for x in l)

            for val in result:
                if val not in unique_list:
                    unique_list.append(val)

def calculate_widths_and_heights(task_set):
    num_set = len(task_set)

    input_widths = []
    input_heights = []
    output_widths = []
    output_heights = []

    for i in range(num_set):
        task_input = task_set[i]['input']
        task_output = task_set[i]['output']

        task_input_width = len(task_input[0])
        task_input_height = len(task_input)

        task_output_width = len(task_output[0])
        task_output_height = len(task_output)

        input_widths.append(task_input_width)
        input_heights.append(task_input_height)
        output_widths.append(task_output_width)
        output_heights.append(task_output_height)

    return (input_widths, input_heights, output_widths, output_heights)
```

Figure 4. functions involved in creating summary data frame

Data Storytelling

In order to gain a better understanding of the data contained in the training set, the summary data frame created in the previous section of the project was used to create visualization. First, the sizes of the training set of each JSON file was counted. This plot is shown in Figure 5.

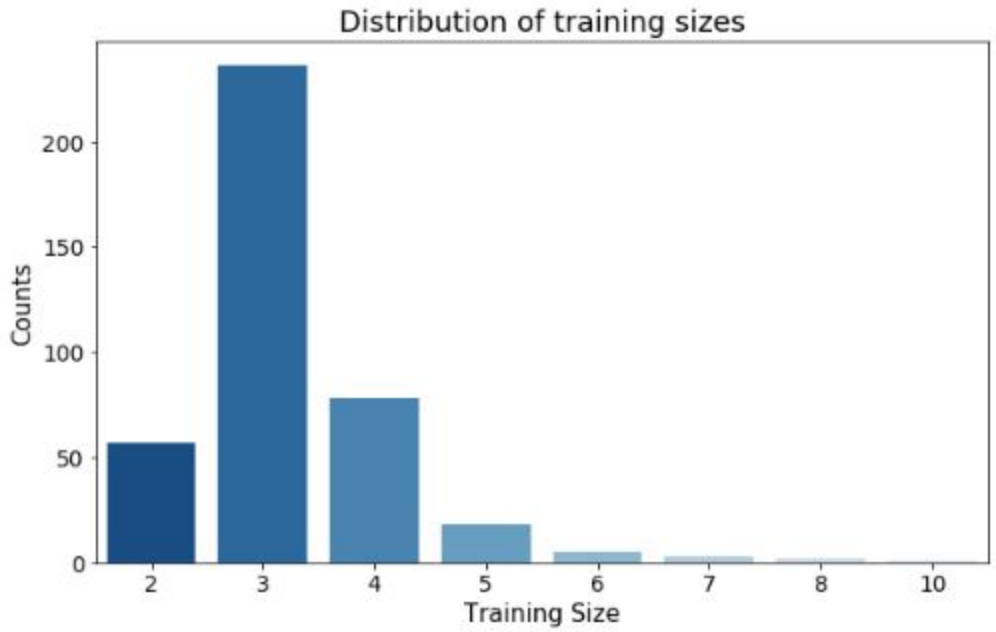


Figure 5. Distribution of training sizes.

Clearly, the majority of the tasks contain three training samples. The model must be able to accept a varying number of training samples per task. Next, the frequency of each color per training size was plotted. An example is shown below for tasks containing three training samples.

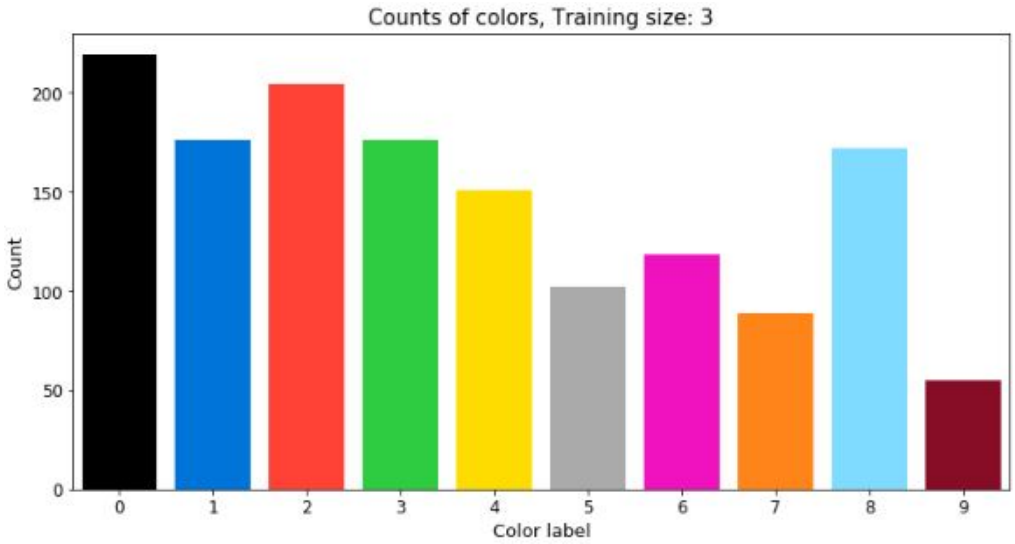


Figure 6. Plot showing how frequently each color occurs within tasks with three training samples.

From this plot we can see that colors occuring most frequently in tasks with three training samples are black, blue, red, green, and light blue. The other tasks can be analyzed similarly using their respective plots. Finally, the dimensions of the inputs and outputs were plotted to see if there were any trends between them. These are displayed below in Figure 7. Regarding the widths versus the heights of the inputs and outputs, they primarily followed a linear trend. Additionally, there were clearly many cases where the dimensions of the input matched the dimensions of the output, however, for the cases they dimensions did not match, the input dimension tended to be larger than the output dimension. This suggests that many of the transformations within the tasks were compressive in nature.

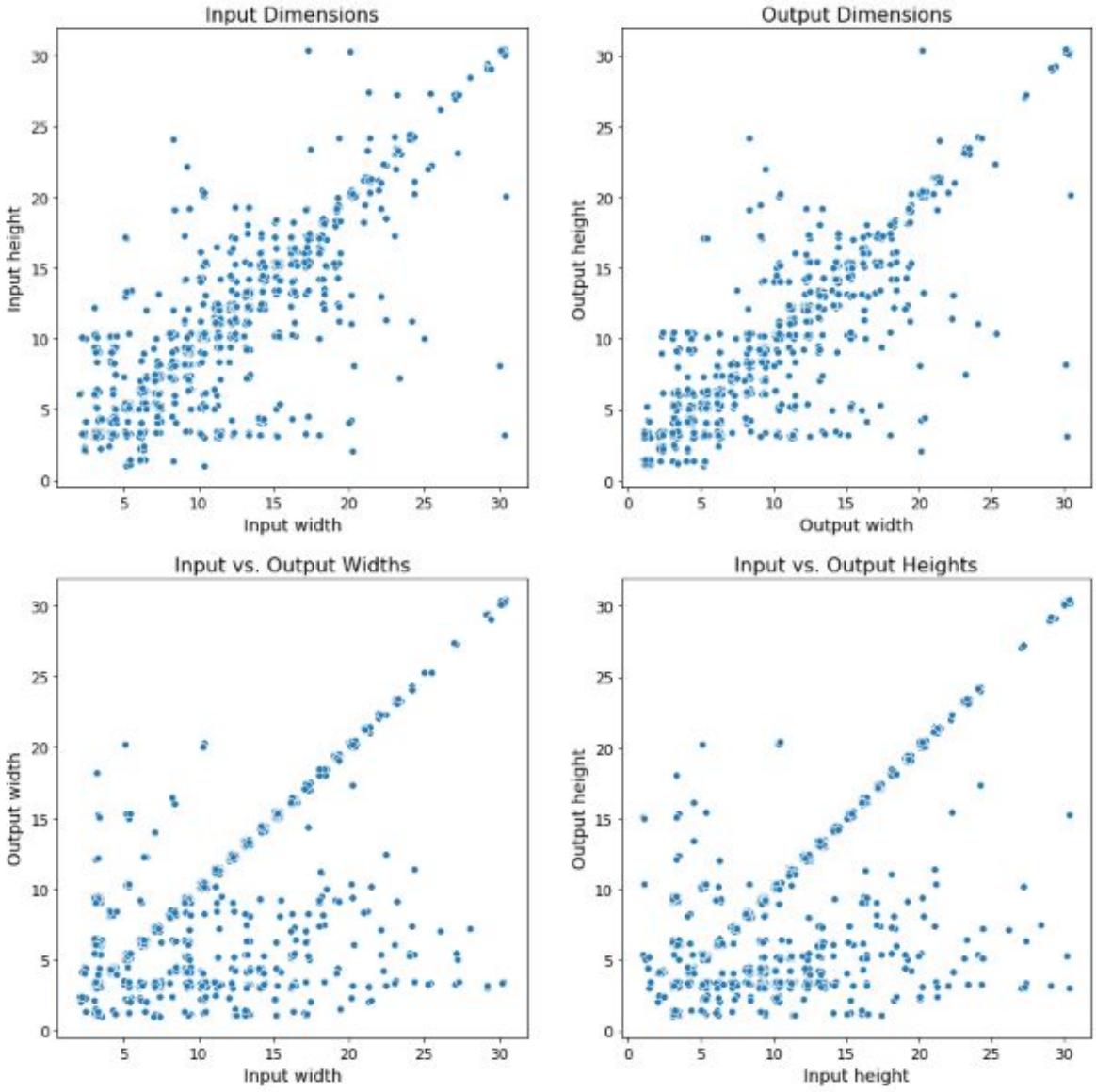


Figure 7. Dimension relationships.

Statistical Data Analysis

In order to verify many of the observations seen in the visual analysis, statistical data analysis was done. First, the correlation between colors and training sizes was plotted. This can be seen in Figure 8 below. From the correlation plot, we cannot see a clear relationship between the occurrence of the different colors and the number of samples in the training set of each file.

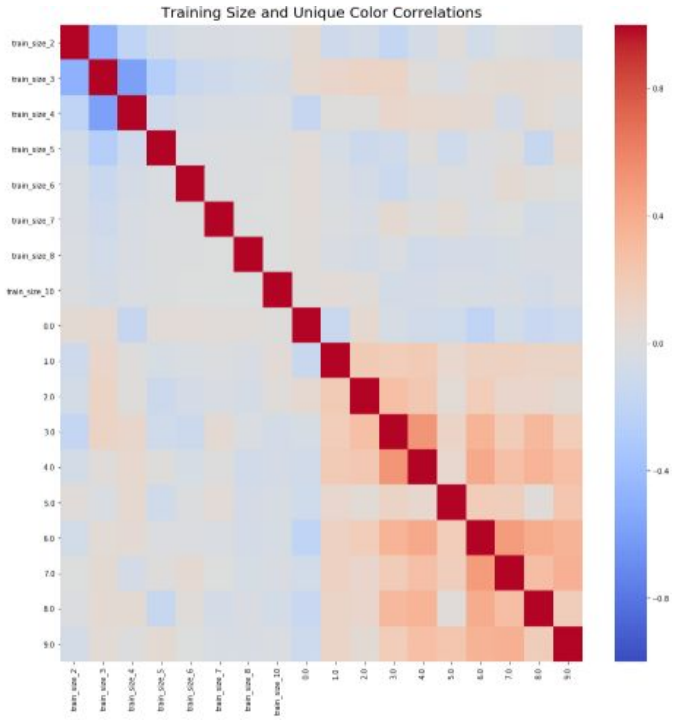


Figure 8. Correlation plot between training size and colors.

Next, the input dimensions were inspected using bootstrapping. For this section, only the largest groups were analyzed (training size = 2, 3, 4). An example of the output is shown below:

```
perform_bootstrap_analysis(col, 2, 4)
```

```
p-value: 0.0413
Observed mean difference:  1.1606
```

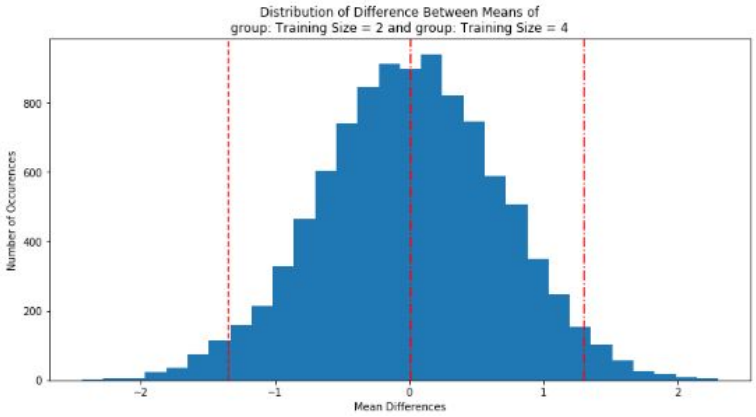


Figure 9. Bootstrapping output example.

This example analyzes the mean of the input widths of the tasks with training samples of size two and four. From the example, we can see that there is a significant difference between the means of each group's input widths, namely that tasks with two training samples tend to have larger widths than tasks with four training samples. A summary of the findings are shown in the table below:

Table I. Bootstrap findings summary table.

Group 1	Group 2	Feature	P-value	Observed mean difference	Reject null hypothesis?
2	3	train_input_width	0.8710	-0.4953	No
		train_input_height	0.9741	-0.8489	No
		train_output_width	0.0780	0.5856	No
		train_output_height	0.2092	0.3171	No
2	4	train_input_width	0.0413	1.1606	Yes
		train_input_height	0.0529	1.0353	No
		train_output_width	0.0016	1.8789	Yes
		train_output_height	0.0002	1.9909	Yes
3	4	train_input_width	0.0001	1.6559	Yes
		train_input_height	0.0000	1.8841	Yes
		train_output_width	0.0027	1.2932	Yes
		train_output_height	0.0003	1.6738	Yes

Model Building

There are three major challenges regarding this set of data:

1. The tasks have varying input and output sizes
2. There are very few training samples per task
3. It is difficult to generate new samples(to increase the size of the training set of each task) due to the abstract rules governing each set of training pair

Preprocessing

In order to tackle the first challenge, the data will be zero-padded into square grids based on the largest input and output dimensions. For example, if the largest input dimension in a task is 15x20 and the largest output dimension is 30x20, the inputs will be zero-padded to a 20x20 grid, while the outputs will be padded into a 30x30 grid. This allows the same model to tackle the varying input sizes present in that particular task. This was accomplished by first determining the largest width and heights within each task. The code used to calculate these values is shown below.

```python
def calc_max_dim(df, test_or_train):
    input_width_col = 'train_' + test_or_train + '_widths'
    input_height_col = 'train_' + test_or_train + '_heights'
    output_width_col = 'test_' + test_or_train + '_width'
    output_height_col = 'test_' + test_or_train + '_height'

    input_max_width = max(df[input_width_col])
    input_max_height = max(df[input_height_col])
    output_max_width = max(df[output_width_col])
    output_max_height = max(df[output_height_col])

    return max(input_max_width, input_max_height, output_max_width, output_max_height)

task_summary['max_input_dim'] = task_summary.apply(func=calc_max_dim, axis=1, args=('input',))
task_summary['max_output_dim'] = task_summary.apply(func=calc_max_dim, axis=1, args=('output',))
```

Figure 10. Code used to calculate maximum dimensions.

Next, the training data had to be padded in according to these maximum dimensions. This was done using the function shown in Figure 11. Using numpy's `.pad` function, zeros were added to the right and bottom of the arrays.

```python
def pad_samples(group, max_input_dim, max_output_dim):
    padded_inputs = []
    padded_outputs = []

    for io_pair in group:
        sample_input = io_pair['input']
        sample_output = io_pair['output']

        input_width = len(io_pair['input'][0])
        input_height = len(io_pair['input'])

        output_width = len(io_pair['output'][0])
        output_height = len(io_pair['output'])

        input_row_padding = max_input_dim - input_height
        input_col_padding = max_input_dim - input_width

        output_row_padding = max_output_dim - output_height
        output_col_padding = max_output_dim - output_width

        padded_input = np.pad(sample_input, ((0, input_row_padding), (0, input_col_padding)), 'constant')
        padded_output = np.pad(sample_output, ((0, output_row_padding), (0, output_col_padding)), 'constant')

        padded_input = padded_input.reshape(1, max_input_dim, max_input_dim)
        padded_output = padded_output.reshape(1, max_output_dim, max_output_dim)

        padded_inputs.append(padded_input)
        padded_outputs.append(padded_output)

    padded_inputs = np.array(padded_inputs)
    padded_outputs = np.array(padded_outputs)

    return padded_inputs, padded_outputs
```

Figure 11. Code used to pad input and output arrays

CNN Model Building

Regarding the second challenge, two approaches were taken. First, from the visual analysis, it was observed that the outputs are related to the inputs either by a transformation of the input data or by a set of spatial rules. Two models with the potential to capture these abstract rules are convolution neural networks(CNNs) and variational autoencoders(VAEs). Convolutional neural networks are exceptional at modeling the features of a given dataset, and due to the grid-like nature of this data, they can be thought of as images. On the other hand, VAEs are able to learn a latent variable model for its input data, or in other words, VAE's have the potential to learn the rules that created the input and output pairs.

In order to implement the convolutional neural network, Tensorflow's keras api was used to describe the architecture of the model. An example summary of the model architecture is shown below in Figure 12.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 32, 8, 8)          320
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 31, 7, 64)         2112
_________________________________________________________________
dropout (Dropout)            (None, 31, 7, 64)         0
_________________________________________________________________
flatten (Flatten)            (None, 13888)             0
_________________________________________________________________
dense (Dense)                (None, 128)               1777792
_________________________________________________________________
dropout_1 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 81)                10449
=================================================================
Total params: 1,790,673
Trainable params: 1,790,673
Non-trainable params: 0
_________________________________________________________________
```

Figure 12. Example of CNN architecture

The shapes of the input layer, second convolutional layer, and output layer all varied due to the differing sizes of each of these tasks. Additionally, the dimension of the output was determined by the size of the flattened output array. This array was later reconstructed into the shape of the original output grid and then plotted. An example of a prediction is shown below in FIgure 13.

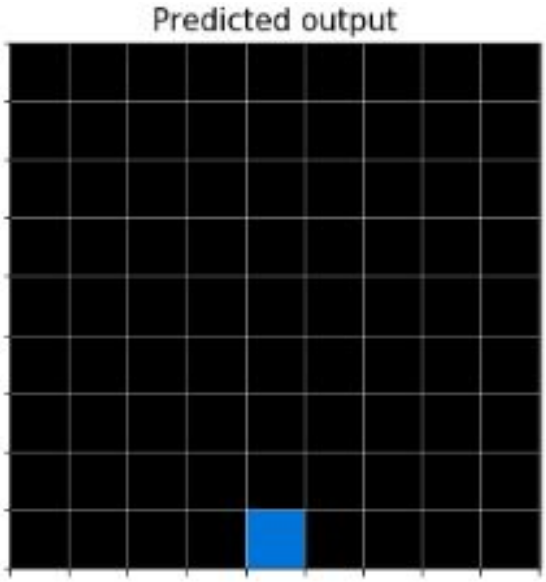


Figure 13. Example output grid.

CNN Evaluation

Ultimately the convolutional neural network approach was unsuccessful achieving an accuracy of 0. There are two likely reasons behind this. First, the training size of each task ranged from only two to ten samples. This is likely not enough to train a convolutional neural network. Even if it were, the network would likely severely overfit and struggle to accurately predict new cases. Additionally, the architecture of each model is not sophisticated enough to capture the unique patterns behind each task. Each task may require their own unique architecture to make accurate predictions.

A possible improvement to this implementation is to use a recurrent neural network to generate each individual cell value. By using a recurrent neural network, you are able to capture the information from all of the previous predicted cells, potentially allowing the network to understand patterns and how each cell is related to the next.

Variational Autoencoder Model Building

VAE's are a type of generative model that encodes the input as a distribution over a latent space. It is a generative model in that it is possible to sample that latent space to create new models. This latent space has the potential to capture rules of the patterns that created the input and output images.

The VAE is constructed by first defining its encoder. This encoder converts the input samples into two parameters in a latent space, which is then sampled. These samples are then mapped to the original input data through a decoder. A summary of the encoder and decoder architectures are shown below in Figures 5 and 6 respectively.

```
Model: "encoder"
_________________________________________________________________________________________________
Layer (type)                 Output Shape              Param #     Connected to
=================================================================================================
encoder_input (InputLayer)   [(None, 100)]             0
_________________________________________________________________________________________________
dense_20 (Dense)             (None, 256)               25856       encoder_input[0][0]
_________________________________________________________________________________________________
z_mean (Dense)               (None, 2)                 514         dense_20[0][0]
_________________________________________________________________________________________________
z_log_var (Dense)            (None, 2)                 514         dense_20[0][0]
_________________________________________________________________________________________________
z (Lambda)                   (None, 2)                 0           z_mean[0][0]
                                                                   z_log_var[0][0]
=================================================================================================
Total params: 26,884
Trainable params: 26,884
Non-trainable params: 0
_________________________________________________________________________________________________
```

Figure 5. Encoder model summary

```
Model: "decoder"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
z_sampling (InputLayer)      [(None, 2)]               0
_________________________________________________________________
dense_21 (Dense)             (None, 256)               768
_________________________________________________________________
dense_22 (Dense)             (None, 100)               25700
=================================================================
Total params: 26,468
Trainable params: 26,468
Non-trainable params: 0
_________________________________________________________________
```

FIgure 6. Decoder model summary

We were able to successfully train the model on the training data, but unable to incorporate the output data into the latent space. A possible successful implementation would have been to use the input data to train the parameters of the encoder and while using the output data to train the output of the decoder. That way, the latent space of the input data could be mapped to the latent space of the output data, and the model would successfully be able to predict the correct output.

Conclusion

The Abstract and Reasoning Challenge was designed to test the limits of AI and determine whether it was possible to create an architecture capable of learning many different tasks using very little training. These constraints were made to mimic a human's ability to learn. If successful, the architecture/pipeline could vastly improve the capabilities of the modern machine learning models. Ultimately, this challenge proved extremely difficult, and the methods attempted in this project were unsuccessful. Despite this, perhaps more complicated versions of the models used could accomplish the task. The methods tried could be improved upon by future data scientists hoping to tackle this unique challenge.