Recurrent Convoluted Neural Networks For Abstract Reasoning

Ethan Kulman
University of Rhode Island

May 1, 2020

Abstract

The measurement of artificial general intelligence is a widely debated area of artificial intelligence. One proposed benchmark of artificial general intelligence is the Abstraction and Reasoning Corpus (ARC). The ARC is a dataset which contains a series of abstract pattern recognition tasks with very few training examples. This study proposes that a recurrent convolutional neural network (RCNN) can learn to solve the puzzles in the ARC dataset. The RCNN architecture will attempt to represent each task in the dataset as a *Cellular Automata* (CA). A *Cellular Automata* is a dynamical system whose state changes based on a transition rule. The proposed recurrent convolutional neural network will be used to attempt to learn the transition rule for each CA. We find that this approach obtains moderate success in training, but does not generalize well to evaluation and test data.

1 Introduction

There is no standardized method for measuring the intelligence of a machine. [1] There have been many tests designed to measure specific intelligence capabilities, but very few have been accepted to measure what is known as *artificial general intelligence*. *Artificial general intelligence* in this context refers to the ability of a machine to achieve high skill across many different tasks. [1] One test proposed to measure artificial general intelligence is the *Abstraction and Reasoning Corpus*. The *Abstraction and Reasoning Corpus* (ARC) is a dataset which comprises of a series of training and test examples which attempt to measure the ability of a machine to perform abstract reasoning. This data set was produced in parallel with an in-depth paper titled *On the measure of intelligence*, written by Francois Chollet. The paper by Chollet thoroughly discusses the previously accepted definitions of artificial general intelligence, and why they fall short of providing a framework which scientists can use to accurately measure their progress toward achieving the goal of artificial general intelligence. Chollet then proposes a new actionable definition of artificial general intelligence, and then provides the ARC dataset as a way to measure artificial general intelligence under this new definition.

Along with the release of this dataset, there is a related competition being held on the *Kaggle* website. *Kaggle* is a website dedicated to crowdsourcing solutions for unsolved problems in the data science field. The competition being held on *Kaggle* seeks to compel data scientists to attempt to create a model which can learn how to solve the puzzles presented in the ARC dataset.

The aim of this study is to determine to what extent a recurrent convolutional neural network can learn to solve the puzzles in the ARC dataset. Each task in the ARC dataset will be represented as a *Cellular Automata* (CA). A *Cellular Automata* can be defined as a dynamical system represented by a matrix, where the matrices values update based on each local entries current value, and the entries surrounding it. How the CA values update is known as a *rule*. A recurrent convolutional neural network will be used to attempt to learn the rules for each CA.

This paper describes the effort to utilize a Recurrent Convolutional Neural Network architecture to represent a *Cellular Automata* ruleset which can solve the tasks in the ARC dataset. The following sections detail the effort underwent to develop this model. These sections are titled: Related Works 2, Methodology 3, Experiments 4, Results 5, and Analysis 6, and finally the Conclusion 7.

2 Related Work

2.1 Cellular Automata

Using deep learning to mimic cellular automata (CA) has been an active area of research in recent years due to the ability of CA to mimic dynamical systems. [2] At the local level, a particular entry in the cellular automata matrix denoted as C_{ij}, where i and j are the row and column respectively of the entry, will update its value based on its current values as well as d other entries in the matrix. [2] All of the local entries which interact with each other to update a particular C_{ij} are known as a neighborhood of cells. How a CA updates its state depends on the rules or functions that a particular CA has defined. [5]

2.2 Using Recurrent Convolutional Neural Networks to Represent Cellular Automata

There have been several works which have discussed the use of recurrent convolutional neural networks (RCNN) to represent cellular automata. [2] [5] Through previous research, it has been shown that any cellular automata can be represented by a convolutional neural network consisting of a convolutional layer with an $n \times n$ filter, followed by m convolutional layers with 1×1 filters. The convolutional layer which applies an $n \times n$ filter is used to learn how to extract a neighborhood of cells which interact with each other for a particular rule. Then the convolutional layers with 1×1 filters learn the cell level details of a particular rule. It has been shown that many different rules for a CA can be learned by an RCNN architecture, and the networks internal representation is able to weigh the importance of different rules. [2] In these networks, there are often no pooling layers and the input dimensionality is maintained as a sample passes through the network.

3 Methodology

3.1 Considerations

There are two considerations that had to be made during the development of models to learn the ruleset in the Abstraction and Reasoning Corpus dataset. These two considerations are: whether or not to augment the original dataset, and how to develop a model that works well given the hardware limitations imposed by the *Kaggle* competition.

Ultimately, it was decided that the original dataset would not be augmented, due to evidence presented by Francois Chollet in his paper *On the measure of intelligence*. Chollet described how traditional deep learning models with enormous training sets do not teach us anything about how to develop a generalizable model that can perform many different tasks. [1] This supports the idea that augmenting the original Abstraction and Reasoning Corpus data set to increase the number of training samples for each task will not improve generalizability of the model. Increasing the number of training examples could instead give the model unlimited data or prior knowledge about one particular task. [1] Therefore, it was decided that the deep learning model developed for the Abstraction and Reasoning Corpus dataset will not require the dataset to be augmented.

The competition rules on the *Kaggle* website required that all solutions must be able to run in a

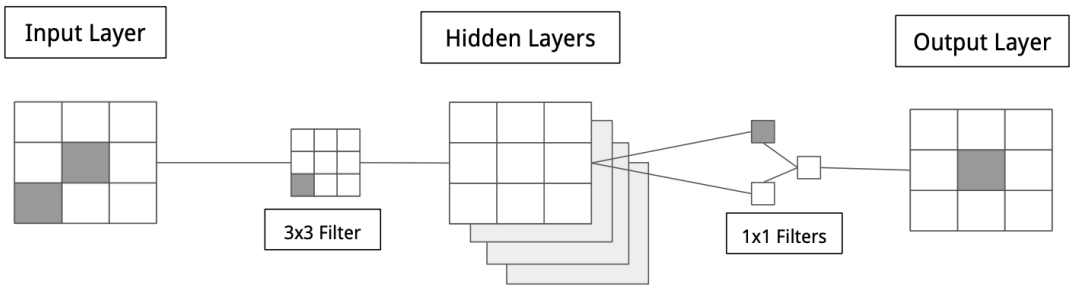

Figure 3.1: General architecture for the implementation for recurrent convolutional neural network.

IPython kernel on the *Kaggle* website. This means that submissions to the competition could only use a maximum of 1 GPU, and 16 CPU cores. The competition rules limit the hardware capacity available, and thereby imposes restrictions on how complex of a model can be developed. These limitations had to be considered throughout model development.

3.2 Description of Data

The training and test examples in the ARC dataset are all $m \times n$ grids represented by a matrix. Each cell in a training or test example matrix contains a number ranging from 0 to 9. Each task in the dataset presents a pattern of how the input matrix cell values should change to achieve the correct output matrix. Each task consists of 3-5 training examples, and one test example. In total, there are 400 tasks in the training dataset which can be used to develop potential models. There are then 400 evaluation tasks which a trained model can be evaluated with. And finally, there are 400 test tasks which a trained model can perform inference on. The output from the test tasks are meant to be submitted to the competition on the *Kaggle* website. The correct outputs of the test tasks are not supplied in the dataset. A submission of the inferences on the test data to the *Kaggle* website returns a loss value from 0 to 1.

3.3 RCNN Network Architecture

The RCNN network architecture developed for the ARC dataset is based off of previous architectures which were able to successfully represent cellular automata. [2] The input layer for these types of networks is a convolutional layer with an $n \times n$ filter for consolidating information about a neighborhood of matrix entries. The hidden layers are a series of convoluted layers with 1×1 filters used to improve expressivity of the network. [7] The output layer of these networks is a convolutional layer which outputs the features back to the same dimensions as the input data. The implementation of the RCNN architecture developed for the ARC dataset is described in the following subsections: 3.3.2, 3.3.3, 3.3.4. A general diagram of this implementation can be seen in Figure 3.1.

3.3.1 Input Example Modifications

Each input example C is a $p \times q$ dimension matrix. When fed into the network, the input example will be expanded into $10 \times p \times q$ dimensions. Each j^{th} layer, where $0 \leq j \leq 9$, of the 10 different layers represents which which entries in $C_{p \times q}$ are part of the j^{th} class. All training and inference will be done in $10 \times p \times q$ dimensions.

3.3.2 Input Layer

The input layer for this network is a convolutional layer with an $n \times n$ filter. Its purpose as previously stated is to consolidate neighborhood information from the input data. It has 10 input channels for each of the 10 different classes that a particular entry in the matrix can be. This layer has 100 output channels which it feeds to the hidden layers.

3.3.3 Hidden Layer

The hidden layers in this network consist of a series of convolutional layer with a 1×1 filter. Each hidden layer has 100 inputs channels and 100 outputs channels. Each hidden layer will perform normalization of its output values, and utilize the rectified linear unit activation function described in Section 3.3.5.

3.3.4 Output Layer

The output layer in this network is a convolutional layer with 100 input channels and 10 output channels. It has a 1×1filter, and will output the data in the same dimensions as it entered the network. This output will be fed into a softmax activation function described in section 3.3.6. This output is what will be used to determine loss.

3.3.5 Rectified Linear Unit

The activation function used at the input and hidden convolutional layers in this network architecture is the *Rectified Linear Unit* (ReLU). This activation function bring any values below zero to zero, and leaves all positive values as they are. [6] The formal mathematic definition of this function is as follows:

$$f(x) = \begin{cases} x_i, if x_i \geq 0 \\ 0, if x_i < 0 \end{cases}$$

This activation function is the part of the key to eliminating the issue of vanishing gradient that is common to recurrent neural networks.

3.3.6 Softmax

The Softmax function is used to compute a vector of probabilities related to what class a particular entry might be. The total sum of its outputs is equal to one. It is used to make the final determination about the classification of a training example. [6] Its mathematic definition is as follows

$$f(x_i) = \frac{exp(x_i)}{\sum_j exp(x_j)}$$

3.4 Loss Function

The loss function utilized in the proposed RCNN network is *Cross Entropy Loss*. Cross entropy loss is defined as follows:

$$\sum_{j}^{M} \mathbf{y}^{(j)} log \sigma(\mathbf{o})^{(\mathbf{j})}$$

where M is the number of classes, j is the the index of j^{th} class, $\mathbf{y}$ is a vector of binary values where the index of the correct class is 1, and $\sigma(\mathbf{o})$ is a vector of probabilities that the observation is a particular class. [3]

In the context of the ARC dataset, M would be 10 since the values for a particular entry in a sample is an integer from 0 to 9.

3.5 Optimizer

The optimizer used in this network architecture is the Adam optimizer. The Adam optimizer uses stochastic gradient descent but adjusts the learning rate based off of several different variables such as the moving average gradients and squared gradient. [4] This optimizer has been used in other RCNNs which attempt to represent cellular automata. [2]

4 Experiments

4.0.1 Input Filter Size

The $n \times n$ filter size for the first layer in the network was selected by testing several different sizes for n. Depending on the value of n, a certain level of padding was required to ensure the input dimensions were maintained. Specifically, n was tested at sizes of 3, 5, and 7. Through testing with several subsets of the dataset, it was determined that the size of n which maximized prediction accuracy was 3.

4.0.2 Normalization Of Inputs To Hidden Layers

Through testing, it was found that the hidden layers of the network were still prone to the vanishing gradient problem even when using the *rectified linear unit* activation function described in Section 3.3.5. Through trial and error, it was found that normalizing the inputs to each hidden layer of the network increased the number of training examples that the model correctly predicted. The number of training examples predicted correctly went from 5/400, to 13/400 after implementing normalization between hidden layers.

4.0.3 Hyper-parameter Tuning

Several different experiments were performed for tuning the following hyper-parameters: number of hidden layers in the network, starting learning rate, number of linear decreases of the learning rate, and the number of epochs per learning rate increment. The metric used to determine which set of parameters is optimal for the training data is the cross entropy loss function described in Section 3.4.

Each of these different parameters were tuned via a grid search. The grid search was performed on several subsets of the training data to obtain a sense of how varying the parameters impacted model accuracy, without requiring large amounts of computing power. The search parameters for the number of hidden layers, starting learning rate, number of linear decreases of the learning rate, and the number of epochs is summarized in Table 4.1. The number of hidden layers refers to the number of convolutional layers with a 1×1 filter size. The starting learning rate refers to what the Adam optimizer is initialized

Hyper Parameter	Values Used In Grid Search
Number of Hidden Layers	1, 3, 5, 10, 15, 20
Starting Learning Rate	1, 0.5, 0.2, 0.1, 0.01
Number of Learning Rate Linear Decreases	1, 5, 10, 15, 20
Number of Epochs Per Learning Rate	1, 5, 10, 15, 20

Table 4.1: Grid search parameters for number of hidden layers, starting learning rate, number of learning rate linear decreases, and number of epochs per learning rate.

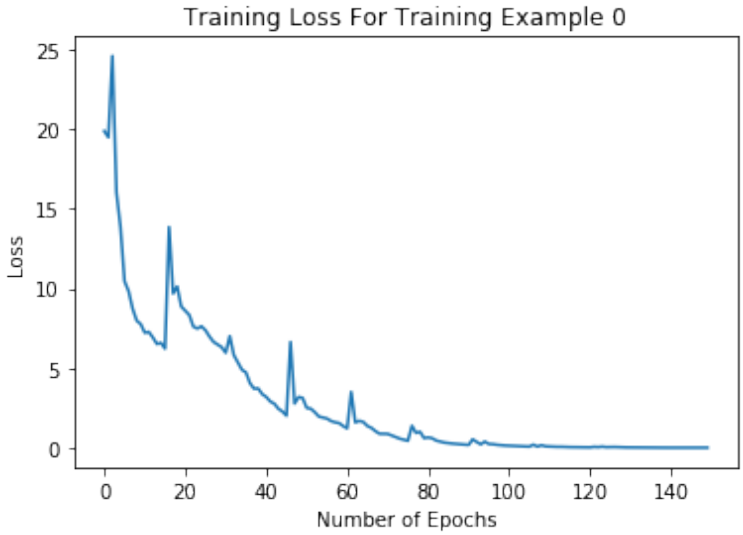

Figure 4.1: Training loss for the first training example

with as a learning rate. The number of linear decreases in the learning rate is a parameter used to decrease the learning rate used in the Adam optimizer after a certain number of epochs. The number of epochs is how many epochs will take place for a particular learning rate. The total number of epochs per training example can be calculated by multiplying the number of linear decreases in the learning rate by the number of epochs per learning rate.

The grid search resulted in the following findings for optimizing model accuracy when using different subsets of the training data. The optimal number of hidden layers was determined to be somewhere between 5 and 10 hidden convolutional layers. This is similar to the number of hidden layers found to be successful by other studies looking to use RCNN models to represent cellular automata. [2] The final number of hidden convolutional layers chosen was 10.

The number of epochs per learning rate, as well as the number of learning rate linear decreases were found to be optimal for most training examples when the total number of epochs surpassed 100. This phenomenon is highlighted in Figure 4.1, where the loss for a specific training example is shown. This training example will be denoted as training example 0. Therefore, it was decided that the total number of linear decreases in learning rate would be 10, and the total number of epochs per learning rate would be 15. This would allow 150 total epochs to occur. The starting learning rate was found to be optimal was it was initialized at 0.1. Loss increased greatly if the starting learning rate was initialized above 0.2.

Final Training/Evaluation Results			
Dataset Being Evaluated	**Number of total examples**	Correct Examples	Percent Accuracy
Training Set	400	13	3.25%
Evaluation Set	400	1	0.25%

Table 5.1: Training and evaluation results from the finalized model.

5 Results

5.0.1 Training, Evaluation, and Testing Results

The final training and evaluation results produced by the proposed network were not very good. Of the 400 training examples, 13 were able to be predicted correctly. Of the 400 evaluation examples only 1 was able to be predicted correctly. The training loss obtained by submitting the inference results on the test data was 1.0. These lackluster results are summarized in Table 5.1. The analysis of these results can be found in Section 6.

6 Analysis

Despite the training, evaluation, and testing results looking quite poor, the results achieved by many of the participants in the competition are similar if not worse. Many of the participants reported in the forums on the *Kaggle* website that they were only able to get 10-15 training examples correct, and 1-5 evaluation examples. In this competition, nearly all of the participants had a testing loss of 1.0, which is the highest loss possible. What this shows is that developing a model to accurately predict the examples in the ARC dataset is extremely difficult.

One of the issues found while developing this model is that the loss for a particular training example might approach zero, but the classification will still be wrong. It was very difficult to find any literature on this phenomenon, but further investigations must be performed since this could be the reason why the network did not perform very well.

One of the major issues with training this network is how slow it is to train given the hardware limitations described in Section 3.1. This made it so small batches of training examples had to be used in hyper-parameter tuning, and the grid search for these parameters had to be limited. Increasing the number of hidden layers in the network slows training tremendously, and so it wasn't possible to test having more than 25 or more hidden layers without having to run the program for an entire day. This made trying new approaches in the network a slow and cumbersome process.

7 Conclusion

The *Abstraction and Reasoning Corpus* dataset attempts to measure the ability of a machine to perform abstract reasoning. The use of a recurrent convolutional neural network architecture to represent a *Cellular Automata* shows some potential, but ultimately fails to come up with any exciting results. During the process of training this network, it was reveal that minor network architecture details drastically change the performance of this type of model. This indicates that it may be better to start with a model that has very few hidden layers, and try to best optimize the model through a more exhaustive grid search of the other hyper parameters. Only once the training accuracy has been maximized using one hidden layer, would then an additional hidden layer be added. This approach might be better for rapidly improving the results of the network given the hardware limitations imposed by the competition rules.

Ultimately, this is a very difficult dataset to successfully train a model on. The solutions in this competition will help forward the progress towards achieving the goal of artificial general intelligence. However, it may be the case that we are unaware of the architectures needed to develop a successful model under the restrictions of this competition. If this is the case, then it will mean that we could be many years away from developing a model that can perform even moderately well on the Abstraction and Reasoning Corpus dataset.

References

[1] Franccois Chollet. On the measure of intelligence. *ArXiv*, abs/1911.01547, 2019.

[2] William Gilpin. Cellular automata as convolutional neural networks. *Physical Review E*, 100(3):032402, 2019.

[3] Katarzyna Janocha and Wojciech Marian Czarnecki. On loss functions for deep neural networks in classification. *arXiv preprint arXiv:1702.05659*, 2017.

[4] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[5] Stefano Nichele and Andreas Molund. Deep learning with cellular automaton-based reservoir computing. 2017.

[6] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation functions: Comparison of trends in practice and research for deep learning. *arXiv preprint arXiv:1811.03378*, 2018.

[7] Maithra Raghu, Ben Poole, Jon Kleinberg, Surya Ganguli, and Jascha Sohl-Dickstein. Survey of expressivity in deep neural networks. *arXiv preprint arXiv:1611.08083*, 2016.