Automatic Detection of Interesting Cellular Automata

CS 294-082 – Experimental Design for Multimedia Machine Learning (Graduate) – Fall 2020

Randy Fan (Student Author)
Electrical Engineering and Computer Sciences
University of California, Berkeley
randyfan@berkeley.edu

Qitian Liao (Student Author)
Electrical Engineering and Computer Sciences
University of California, Berkeley
liao1024@berkeley.edu

ABSTRACT

Our project explores the potential of using Convolutional Neural Networks (CNNs) to detect interesting two-dimensional outer totalistic cellular automata. A cellular automaton is a collection of colored cells on a grid that evolves according to a set of neighborhood rules. The rules are applied iteratively for as many time steps as desired to generate new grid configurations.

A challenge that two-dimensional cellular automata faces is a large parameter search space to generate patterns within. Assuming we use a Moore neighborhood and a maximum of 10 possible transition states, there are 2^9 survival rules, 2^9 born rules, and 2^{10} states, which leads to a total of 2^{28} combinations of rules. Manually searching for these patterns would be unrealistic as users may have to randomly go through hundreds if not thousands of random rules before finding an interesting one.

In this paper, we will discuss our approach to detect rules and patterns that feature gliders using image processing, CNNs, and ImageNet. Note this project solely focuses on interesting rules with gliders, not other rules that could be subjectively labeled as interesting, such as intricate still life and oscillating patterns.

KEYWORDS

Cellular Automata, Image Processing, Convolutional Neural Network, ImageNet, Entropy Evaluation, Dynamic patterns

1 INTRODUCTION / PROBLEM

In this paper, we focus on outer totalistic generations of two-dimensional cellular automata. They consist of a two-dimensional grid of cells, which live, die, or are in a "dying" transition state depending on a set of rules. The cells are updated according to their previous values, and the sum of the values of the other cells in the neighborhood. We decided to use Moore neighborhood for the update rules, which are the surrounding eight cells of a center cell. An alternative would be Neumann neighborhood, which only includes four neighboring cells.

Interesting rules are defined specifically as those that have clear gliders (small patterns that move across the grid with some individual characteristics). Any other rules are classified as boring, meaning they lead to a pattern which is static noise with no discernable patterns moving across the screen. In Figure 1.1 are some of the famous gliders that have been discovered so far[1].

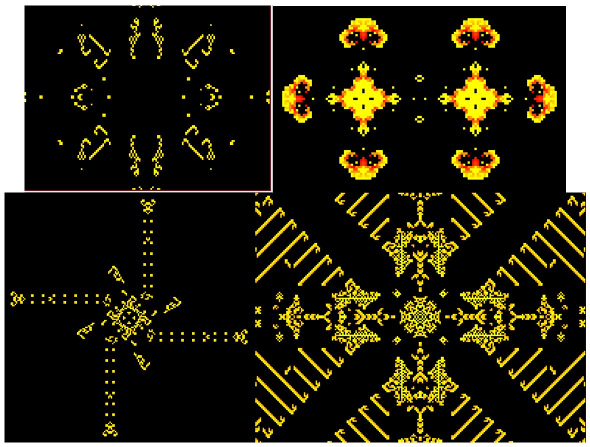


Figure 1.1: Four examples of discovered gilder patterns (Top Left) Brian's brain. (Top Right) Burst. (Bottom Left) Brain6. (Bottom Right) Star Wars.

After manually inspecting 40,000 interesting patterns, it seemed that interesting rules would produce interesting results regardless of the initial configuration. Similarly, boring rules are expected to always produce boring results.

Under most circumstances, it is impossible to tell whether a rule is interesting or boring just by looking at the parameters. A user would have to randomly go through thousands of random rules before finding an interesting one. Even more daunting, the total number of combinations can easily exceed billions. Therefore, automatic detection of interesting rules will be very helpful, and it will make sure that users do not have to manually go through the process.

2 BACKGROUND / RELATED WORK

There have been many attempts to find interesting two-dimensional cellular automata rules. The most naïve approach is to repeatedly create random neighborhood rules and wait to see if it generates an interesting result. Another approach that works for lower dimension cellular automata (specifically one-dimensional cellular automata) is simply brute forcing all the possible sets of rules and manually inspecting which ones are interesting after they have been generated and frames are saved. An alternative naïve approach is to modify existing interesting rules, such as John Conway's Game of Life rules, to generate similar or refined patterns. However, usually just changing a single parameter value could lead to a widely different result. In other words, the interesting results are not necessarily clustered together in the search parameter space.

There exist other more computational methods used to determine the type of cellular automata generated. For example, Chris Langton created a cellular automata lambda value that is computed based on the number of cells that have been born at that time step and dividing it by the total number of cellular automata cells. This gave a decimal value between 0 and 1. Based on his classification system, lambda values within 0.1 and 0.15 means a good rule that could require further investigation[2].

As far as we know, none of these described methods have been able to reliably detect rules that may be subjectively determined as interesting (e.g., they usually find static noise). Thus, detecting interesting gliders in two-dimensional outer totalistic cellular automata is an unsolved problem that we attempt to tackle in this project.

3 METHOD

We first collected the two-dimensional cellular automata data needed for training and testing. This entailed building a data collection pipeline from scratch to generate a sequence of raw frames and then stitching the selected images to represent each of the patterns. Later, several models were tested and analyzed for the best results. We trained the data with a CNN, and performed hyperparameter tuning, image feature extraction via NASNetLarge, and entropy analysis.

3.1 Data Collection Pipeline

To obtain boring rules, we manually went through random examples, and collected rules that died out immediately, generated static noise, or boring non-glider patterns. For interesting rules, we used Cellular Automata Rules Lexicons[1,3], and examples provided in Visions of Chaos[5] and only recorded those with gliders. We ended up with 105 boring and 35 interesting sets of rules.

We generalized the cellular automata generation algorithm as follows:

Cellular Automata Generation Algorithm

```
if center_cell == max_state:
    for num_neighbors in survive_arr:
        if total - 1 == num_neighbors:
            return center_cell
    return center_cell - 1
elif center_cell != 0 and center_cell != max_state:
    return center_cell - 1
else:
    for num_neighbors in born_arr:
        if total == num_neighbors:
            return max_state
        return 0
```

We used the algorithm above to generate frames and create images. We ran the rules using a random initial configuration. Next, we applied data augmentation due to the limited number of rules that we identified. We reused the boring rules 10 times, and the interesting rules 30 times with different initial configuration. We generated 140 images for each of the patterns, and saved images 80-120.

Our next step is image stitching. Frames 100–108 of the evolution were stitched together in a 3×3 grid. This was designed to increase the algorithm's robustness and to control better for interesting configurations that have some seemingly uninteresting frames interspersed throughout their evolutions. As raw data, these images were quite large given our RAM allocation. Running the notebook tended to crash the kernel so we settled for less resolution, downsampling the 1188×1188 pixel images to 300×300. This tradeoff allowed us to manipulate and do machine learning on the data without too much computational expense. We assembled training data with a 50% interesting and 50% boring split.

As a final preprocessing step, the [0, 255] valued matrices representing the images were normalized using simple division to [0, 1]. This improved performance greatly in practice. Many of the CNN architectures we tried produced sub-baseline results before this step. The next part of the optimization process was a question of model architecture and hyperparameter tuning.

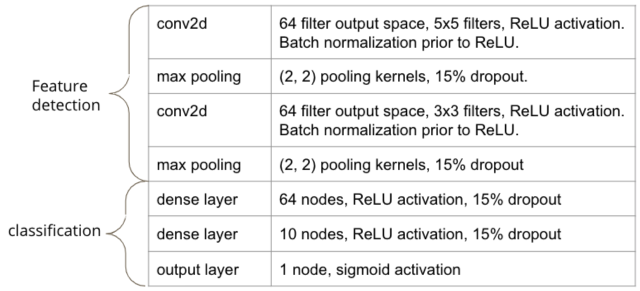
3.2 Data Training with CNN

For our model architecture, we implemented a CNN. This seemed like the logical choice given the fact that we were trying to classify images. We tried many architectural parameters and hyperparameters in order to create the best model. Some of these included: convolutional filter size, dense layers at output, pooling kernel size, type of pooling, dropout, and batch normalization.

We found that the greatest improvements happened after adding dropout and batch normalization. There was also a significant increase in accuracy after increasing the convolutional filter size of the first convolutional layer to 5×5 from 3×3. We believed this is because 3×3 is too small to capture much of the complexity of the interesting configurations. Given a 3×3 window, many of the interesting shapes look like noise.

We tried many things that did not work in addition to those that did. Increasing the pooling kernel size, using average pooling instead of max pooling, increasing the number of filters in the convolutional layers (from 64 in each), and increasing the second convolutional layer's filter size from 3×3 to 5×5, all resulted in worse performance by the validation accuracy metric. We found that increasing the epochs past 30 resulted in overfitting.

Eventually, we used the architecture described in Figure 1.2.



Feature detection	conv2d	64 filter output space, 5x5 filters, ReLU activation. Batch normalization prior to ReLU.
	max pooling	(2, 2) pooling kernels, 15% dropout.
	conv2d	64 filter output space, 3x3 filters, ReLU activation. Batch normalization prior to ReLU.
	max pooling	(2, 2) pooling kernels, 15% dropout
classification	dense layer	64 nodes, ReLU activation, 15% dropout
	dense layer	10 nodes, ReLU activation, 15% dropout
	output layer	1 node, sigmoid activation

Figure 1.2: Structure of the convolutional neural network.

We were able to achieve 93.44% training accuracy and 84.12% testing accuracy on the testing set with 10% interesting data. The test recall is 100%, indicating every interesting configuration has been correctly labeled as such.

3.3 Feature Extraction

We eventually need to feed the results into Brainome.ai, which requires column data. Therefore, we decided to move on to feature extraction and convert our data into a well-organized CSV format. We used a pre-trained NASNet-Large Model, which is a convolutional neural network that is trained on more than a million images from the ImageNet database. For each of the stitched images, the model returns 1000 selected features.

Image Feature Extraction Algorithm

```
model_name="nasnetalarge"
model=pretrainedmodels.__dict__[model_name](num_classes
=1000, pretrained='imagenet')
model.eval()
load_img = utils.LoadImage()
tf_img = utils.TransformImage(model)
features_file = open("file.csv", "ab")
feature_data = []
for i in range(len(image_paths)):
        input_img = load_img(image_paths[i])
        input_tensor = tf_img(input_img)
        input_tensor = input_tensor.unsqueeze(0)
        input = torch.autograd.Variable(input_tensor,
requires_grad=False)
        output_logits = model(input)
        output_features = model.features(input)
        output_logits = model.logits(output_features)
        output_logits = output_logits[0].detach().numpy()
        row_data = np.append(output_logits, labels[i])
        feature_data = np.append(feature_data, row_data)
```

We then fed the data into Brainome.ai and obtained the following information about Decision Trees and Neural Networks. The decision tree has 1026 parameters, and the estimated memory equivalent capacity for neural networks is 11034 parameters. Expected generalization using Decision Tree is 2.05 bits/bit and using a Neural Network is 0.19 bits/bit.

3.4 Entropy Evaluation

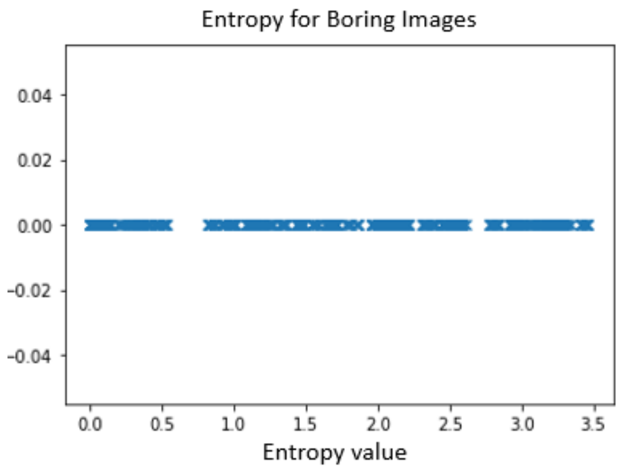
We decided to measure the entropy of the input images to determine if there is a relationship between the label (boring and interesting) and the statistical measure of randomness in the images.

Image Entropy Function

```
def compute_entropy(signal):
    lensig = signal.size
    symset = list(set(signal))
    numsym = len(symset)
    propab = [np.size(signal[signal == i]) / (1.0 * lensig) for i in
symset]
    entropy = np.sum([p * np.log2(1.0 / p) for p in propab])
    return entropy
```

We iterated through all the stitched frames and computed their entropies using the above function. We then plotted the entropy values of the boring and interesting images on one-dimensional line plots (Figure 1.3).

Boring images had entropy values that spread evenly from 0 to 3.5, with a small gap between 0.5 and 0.75. Interesting images were concentrated in the 0 to 2 range, which makes intuitive sense since interesting images have less noise and entropy compared to boring images on average. It should be noted the minimum entropy for boring images was 0.0 while the minimum entropy for interesting was 0.0318. This is because frames that had no live cells were always labeled and classified as boring.



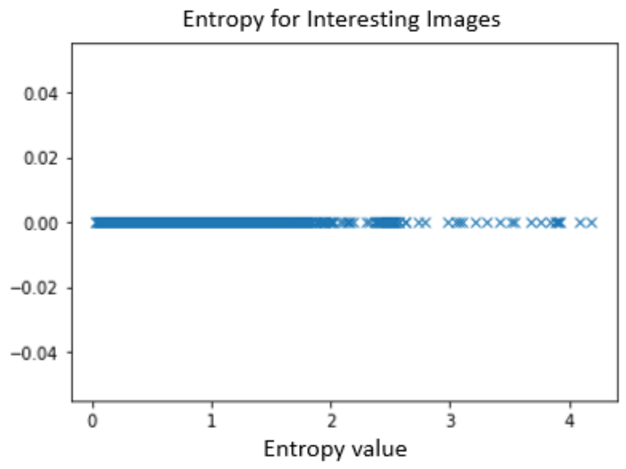


Figure 1.3 Entropy for Boring and Interesting Images

The entropy values suggest adding features identifying if the image entropy is above 3 or equal to exactly 0 may be beneficial for the model accuracy.

4 FUTURE WORK

Automatic detection of interesting cellular automata is a project that has not yet been successfully done in the past. This means our project is breaking new grounds and has potential to explore many alternative possibilities. Our next step includes customized entropy and feature engineering. We still plan to use the frames from 100 to 108. But instead of computing the entropy of the stitched image directly, we will load the nine images as arrays and calculate an elementwise (pixel-by-pixel) difference. The norm of the difference will be used as the new entropy, and the difference vector will be used as the feature vector. We hope that the customized entropy and features will provide us with even more accurate results.

5 CLASS QUESTIONNAIRE

Here the paper will answer the eight questions posted by Professor Friedland to evaluate machine learners.

Q1: What is the variable the machine learner is supposed to predict? How accurate is the labeling? What is the annotator agreement (measured)?

Given a set of rules and an initial configuration, the machine learner aims to predict whether the generated cellular automata pattern will be interesting or boring. CNN gives a training accuracy of 93.44% and testing accuracy of 84.12%.

We found the boring rules by going through random rules together and picking all the ones that did not have glider patterns. To obtain the annotator agreement for interesting rules, we calculated Cohen's kappa when labeling rules from lexicons and other sources. These sources contained a total of 147 potentially interesting rules. We both agreed to include 35 of those rules that had clear glider patterns and excluded 107. There were 3 rules member #1 wanted to include and the other did not, and 2 rules member #2 wanted to include that was not included by member #1. This gives us 96.59% agreement and a Cohen's k equal to 0.91, signifying almost perfect or perfect agreement.

Q2: What is the required accuracy metric for success? How much data do we have to train the prediction of the variable? Are the classes balanced? How many modalities could be exploited in the data? Is there temporal information? How much noise are we expecting? Do we expect bias?

The classes are balanced because the data contains exactly 50% interesting rules and 50% boring rules. There are no modalities that could be exploited in the data. There is temporal information because we use sequences of frames, which record the state of the particles with respect to time. Noise is possible but very rare according to empirical rules. They occur because random initial configuration may lead to results different than expected. For instance, a supposedly boring set of rules might seemingly generate glider patterns given a particular initial configuration.

Q3: What is the Memory Equivalent Capacity for the data (as a dictionary). What is the expected Memory Equivalent Capacity for a neural network?

We have 2100 images in total, and the classification of each image requires 1 bit since there are two classes. Therefore, the Memory Equivalent Capacity for the data is 2100 bits. The expected Memory Equivalent Capacity for a neural network is provided by Brainome.ai, which is 11034 bits. Additionally, we computed the worst-case Memory Equivalent Capacity with the formula (log2(thresholds + 1) * d), which results in 993317 bits.

Q4: What is the expected generalization in bits/bit and as a consequence the average resilience in dB? Is that resilience enough for the task? How bad can adversarial examples be? Do we expect data drift?
The average resilience is not covered this semester.

Q5: Is there enough data? What does the capacity progression look like?
Yes, we have enough data. We are getting high test accuracy when we use 50% of our interesting data and high test accuracy for the 10% interesting test set as well.

According to the results from Brainome.ai when using feature extracted data, the capacity progression (# of decision points) is [8, 9, 10, 11, 11, 12].

Q6: Train your machine learner for accuracy at memory equivalent capacity. Can you reach near 100% memorization? If not, why (diagnose)?
Yes, we can increase the number of layers and neurons, which will allow us to reach Memory Equivalent Capacity easily.

Q7: Train your machine learner for generalization: Plot the accuracy/capacity curve. What is the expected accuracy and generalization ratio at the point you decided to stop? Do you need to try a different machine learner? How well did your generalization prediction hold on the independent test data? Explain results. How confident are you in the results?
We defined the model's generalization capacity G to be

$$G = \frac{N}{C}$$

where N is the number of correctly classified points in the test set and C is the worst-case Memory Equivalent Capacity of the model.

The formula (log2(thresholds + 1) * d) is used to compute the Memory Equivalent Capacity. We used 900, 1200, 1500, 1800, and 2100 (the entire data set) stitched images for training, and generated different test accuracies and Memory Equivalent Capacity, which is illustrated in Figure 1.4. We tuned the hyperparameters (more detailly described in Section 3.2) to improve the accuracy. The model generalizes our data very well, which is shown in the increasing recall ratios and high test accuracies in Figure 1.5.

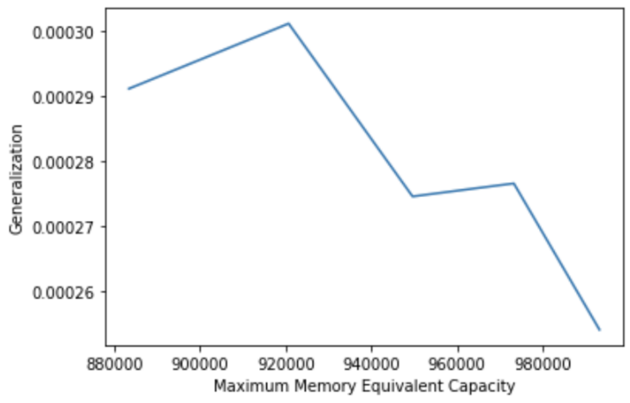


Figure 1.4: Generalization vs. Memory Equivalent Capacity

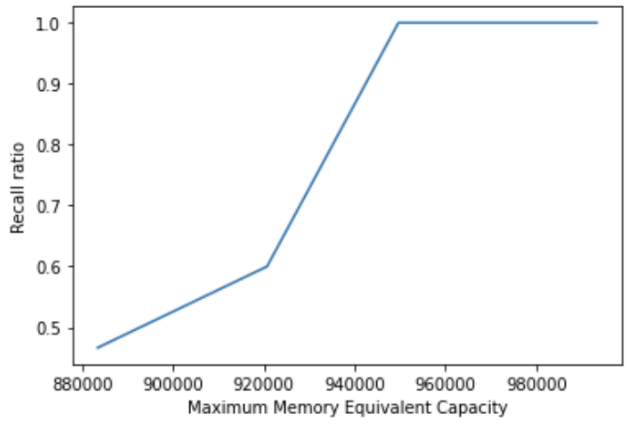


Figure 1.5: Recall Ratio vs. Memory Equivalent Capacity

Q8: Comment on any other quality assurance measures possible to take/the authors should have taken. Are there application-specific ones?
Some quality assurance measures we could have taken include increasing the number of annotators, avoiding adding too many obviously boring rules (e.g., grid turning into all alive or dead), and finding more interesting rules randomly rather than depending on lexicons.

6 CONCLUSIONS

Cellular automata are useful for modelling complicated nonlinear systems in technical industries. Rules that generate glider patterns can potentially benefit existing biology and physics models. Furthermore, cellular automata have a gigantic fan base who are constantly looking for interesting patterns. Due to cellular automata's enormous search space, it is inefficient for humans or other computer programs to systematically do the task. Neural networks provide an alternative approach to solve the problem. Due to their capability to learn after the training phase, they have the potential to automatically detect the interesting cellular automata patterns.

Cellular automata have been heralded for over seven decades as a possible path to artificial general intelligence. Artificial General Intelligence (AGI) is considered by many to be the holy grail of AI research, considered by some to be mankind's next giant leap. AGI differs from classic AI systems because it is generalizable. Although AI systems display great and often superhuman-like abilities on prescribed tasks, there has not yet been a system that has been able to learn any task with few learning examples. There are no standardized requirements for an AI system however most theorists discussing AGI put the following requirements to the system:

- Evolution Assumption: The system must be able to evolve.
- Generalization Assumption: The system must be able to generalize.
- Learning Assumption: The system must optimize for a large swath of tasks.
- Anthropocentric (Core Knowledge) Assumption: The system must be able encode human-like prior knowledge.

In the mid 20th century, John Von Neumann laid the foundations of the mechanism that AGI could be built upon, and that mechanism was based on cellular automata. His mechanism, called a universal constructor, is a cellular automata-based system capable of self-replication and evolution. Between instances of self-replication, perturbations could be introduced into the system causing small changes in "phenotype." This system was a proof of concept and a powerful one. As of yet, nobody has implemented a universal constructor capable of facilitating AGI, but it is evident how such a device could be instrumental in allowing evolution, generalization, and learning.

Despite the large amount of effort and study that has been put into cellular automata, there is still much that is unknown. The space of possible rules is infinite, so the task of determining the interesting ones is pertinent to the continued development of cellular automata theory. We have put forth a brute force solution. While our process is relatively computationally inefficient, it is a good place to start. For the continued development of cellular automata theory and the quest for AGI, we believe our algorithm can be a useful tool to bolster the unsearched frontier.

7 CONTRIBUTIONS

Qitian's role in this project was to stitch images, build, improve, and analyze CNN models, do the feature extractions, feed data into Brainome.ai, and create code for entropy evaluations.

First, Qitian created an algorithm in python to stitch frames 100-108 of each pattern into 3×3 images and store them in designated directories. Then Qitian built a vanilla CNN model with four layers. Later, he also participated in the hyperparameter tuning process with Randy.

For feature extraction, Qitian wrote a script using NASNetLarge and generated 1000 features for each of the stitched images. He stored the data in a styled CSV file, fed it into Brainome.ai, and got useful results for further analysis.

Inspired by Professor Friedland's feedback during the project presentation, Qitian also wrote a script for entropy evaluation, which is later used by Randy to analyze the entropies of interesting and boring patterns.

Randy's role in this project was to create the data collection pipeline from scratch and optimize Qitian's vanilla CNN model. Randy also analyzed the entropy of the stitched images, assisted with model measurements, and modified Qitian's feature extracted dataset so that it could be fed into Brainome.ai.

For the data collection pipeline, Randy had to build it from scratch since there are no pre-existing datasets for this particular project topic. After manually collecting boring and interesting rules and implementing a generalized version of the 2D outer totalistic cellular automata algorithm, he generated a dataset containing approximately 80,000 images.

He also took the data from the pipeline and tuned Qitian's CNN to classify it. This CNN included dozens of hyperparameters, each one needing tuning. In the form of a manual grid-search, Randy found the right combination of hyperparameters that optimized the CNN for its classification task.

The testing phase was designed by Randy. He was responsible for deciding how best to evaluate the system. This required some critical thinking about the best way to measure success and how to split up the dataset into training, test, and validation data.

REFERENCES

[1] http://psoup.math.wisc.edu/mcell/rullex_gene.html
[2] http://math.hws.edu/xJava/CA/EdgeOfChaos.html
[3] http://psoup.math.wisc.edu/mcell/rullex_life.html

[4]https://towardsdatascience.com/emergence-how-artificial-general-intelligence-can-be-computationally-modeled-b5fea4797028
[5] https://softology.com.au/index.htm