Project Report

Conway's Reverse Game of Life 2020

By
Jacob Rodriguez
James Ding
Parshwa Gandhi

Google Colab links for models and data analysis referenced in this report:
Model Training
Data Analysis

1. Introduction

Game of Life was invented by a British mathematician John Conway in 1970. Since its inception, this game has been very popular among computer scientists. This is a zero player game, which means that after the initial configuration of the board, the next state is determined from the previous state. A cell is surrounded by eight neighboring cells and its state depends on the state of its neighbors. A live cell survives if 2 or 3 out of the 8 neighboring cells are alive. If 4 or more cells are alive, the cell dies of overpopulation. If less cells are alive, it dies. A dead cell can be alive if exactly 3 neighboring cells are alive. This game is many to one patterns, meaning many initialized patterns can lead to one final pattern. This competition is an experiment to see if any algorithm can predict the game of life in reverse.

2. Related Work

The initial challenge back in 2013 was on a board with 20x20 for a total of 400 cells. Previously, this task was carried out with the help of a Convolution Neural Network with 8 hidden layers and each layer is followed by a Batch Normalization.

Previously, there had been a paper publication by Jacob M. Springer *et. all* called It's Hard for Neural Networks To Learn the Game of Life. In the paper, they explore the game's setup process in relation to a neural network's random initial weights that it starts with for a given model. This is referenced as the Lottery ticket hypothesis. In this, Springer states that the game itself exhibits the primary characteristic of the hypothesis that the size of the networks required to learn a function are often significantly larger than the minimal network required to implement the same function. This disparity between function and solution of function is explored in a way that puts emphasis on the model initialization even before training has begun. In short, the initial circumstances of a model relate heavily to its final success. This randomness is captured in the game itself and perhaps life itself, so it can be seen as a difficult task to accurately model for any neural network approach.

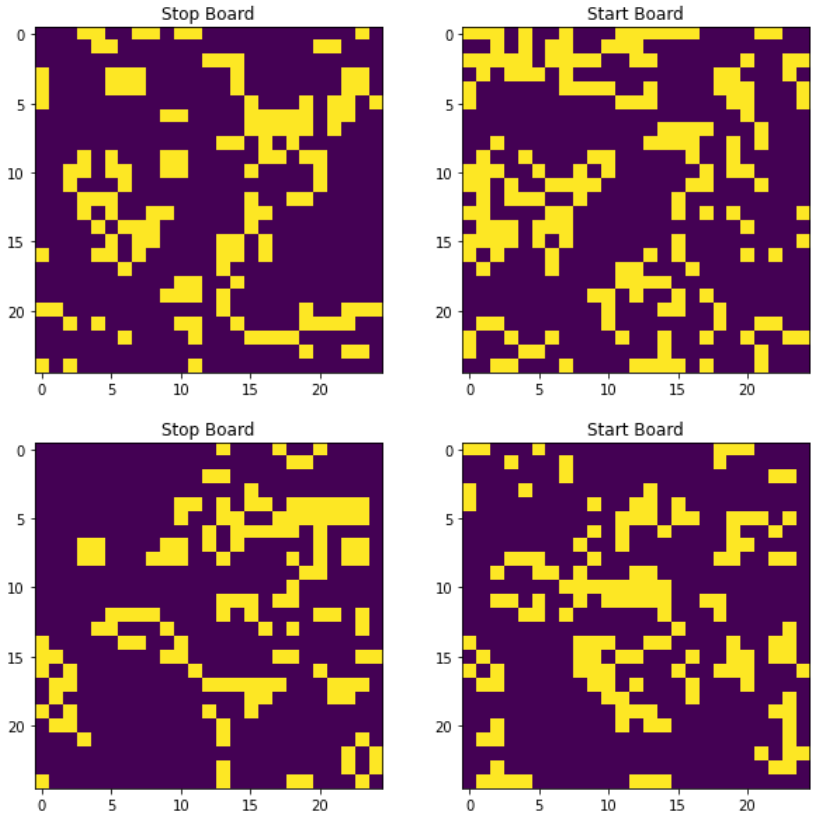
3. Data Exploration & Data Processing

The dataset comes from the Kaggle competition and is already preprocessed in the sense that there are no missing rows or incomplete values. In terms of exploration, the initial format of the data is split into training and test csv files. The training data contains 50,000 rows that map to 50,000 separate game instances. Each row is divided into 4 separate data types:

Column name(s)	**Association**
id	The ID of the game
delta	Steps in time from Start Board to Stop Board
*start_**	Labels from 0 to 625 that indicate if a given tile is 1 or 0 (Live or dead)
*stop_**	Features from 0 to 625 that indicate if a given tile is 1 or 0 (Live or dead)

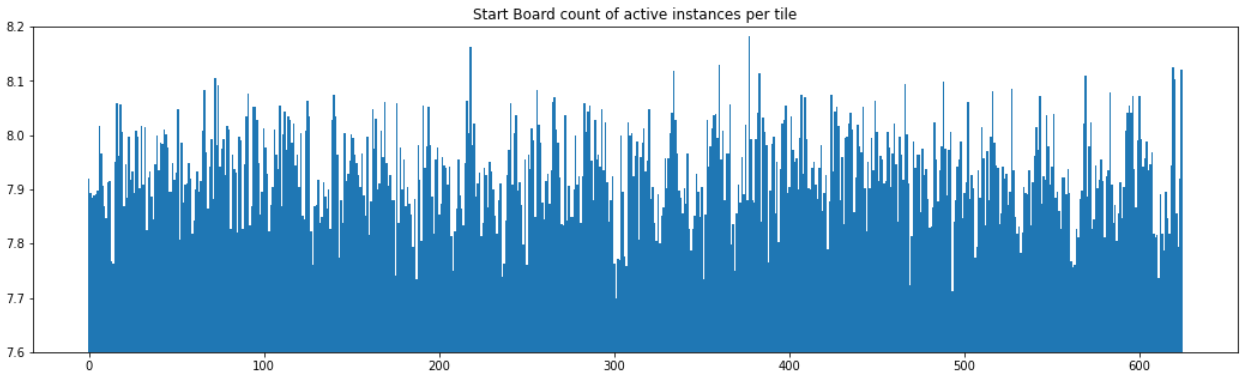
The same format is true for the testing dataset except for the removal of the start labels, since it is a competition.

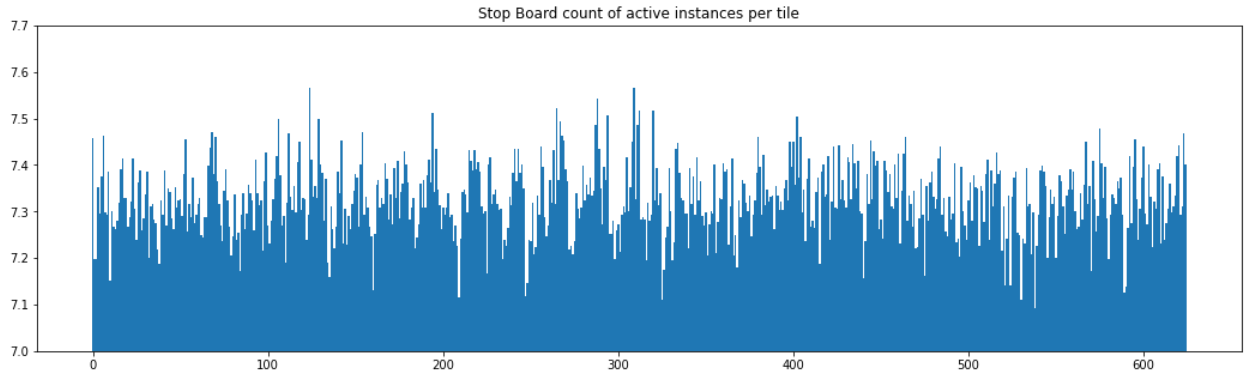
In order to actually explore the data, we first had to split up the start and stop columns. Since *id* carried no particular interest, we decided to drop the column from the dataset. Once the data was split into both start and stop points, we reshaped the flat row into a 25x25 matrix to properly visualize the dataset as a gameboard. Below are some of the start and stop boards from the training dataset. Note that Yellow indicates a tile is live (1).



From these visuals, we were able to get a better idea of how to approach this problem because it transformed our view from a number sequence problem to a picture-type of problem.

Another avenue of exploration was to see if there was an imbalance of live or dead tiles throughout the training dataset. From the training data, we took a count of how many times each tile was active over the entire training dataset. Doing this for both the Start and Stop boards, we arrived at the following two bar plots:





Two notes on these graphs is that they are scaled down by 1000 and the numbering below does not properly show which tile is an edge tile but rather just the rows as a flat instance. This means that although a tile looks to be in the middle, the reason it might actually be lower than the ones around it could be that it is an edge tile in the 25x25 board. Since edge tiles have a harder time to meet the conditions for being live, this would explain why there are lower counts of live instances. However, overall there is not a significant difference of frequency between tiles on the board.

As noted before, due to the nature of the Kaggle competition, no preprocessing was needed, save for the dropping of the *id* column and the splitting and reshaping of the board data itself. Since all features are needed to make a complete board, they each contribute a necessary component to solving this problem. After viewing the frequency counts above, it didn't make sense for us to drop columns because they all appeared to carry a significance. So, as a result, we decided against PCA and other feature reduction techniques and instead proceeded to modeling the dataset.

4. Problem Formulation & Model Selection

We tried several different models to tackle this problem. The first model we attempted was CNN, or convolutional neural networks, using the Keras library. CNN is well suited for problems involving image analysis. Filters carry out the convolution operation by applying themselves on small subsections of the overall grid, shifting to the right and then down until it traverses the whole grid. The nature of this problem is such that the rules of the problem all are related to the number of neighboring tiles that are alive. As such, a CNN could potentially learn the rules of the game by applying a 3x3 or larger filter. The first CNN model consisted of 5 hidden layers of 2D convolution layers with *elu* activation. Upon research, we found that *elu* was better than *relu* for this type of problem due primarily to its ability to avoid the dying reLU problem, which might arise given our dataset. Below is the model summary:

```
Model: "functional_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 25, 25, 1)]       0
_________________________________________________________________
conv2d (Conv2D)              (None, 25, 25, 32)        320
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 25, 25, 64)        18496
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 25, 25, 128)       204928
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 25, 25, 64)        73792
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 25, 25, 32)        18464
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 25, 25, 1)         33
=================================================================
Total params: 316,033
Trainable params: 316,033
Non-trainable params: 0
```

The actual implementation details can be found on the Colab Master notebook: Link

A second attempt with CNN was to build more layers into the original CNN model. Again, the model summary is below:

```
Model: "functional_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 25, 25, 1)]       0
_________________________________________________________________
conv2d (Conv2D)              (None, 25, 25, 32)        320
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 25, 25, 64)        18496
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 25, 25, 64)        36928
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 25, 25, 128)       204928
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 25, 25, 128)       409728
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 25, 25, 64)        73792
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 25, 25, 64)        36928
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 25, 25, 32)        18464
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 25, 25, 1)         33
=================================================================
Total params: 799,617
Trainable params: 799,617
Non-trainable params: 0
_________________________________________________________________
```

Implementation details can be found on the same master notebook link.

Another model we chose was random forest. For this model, we performed an additional preprocessing step. We converted the 25x25 alive/dead grid to a 625 feature vector with each entry representing the number of neighboring alive tiles (index 625 would correspond to 25, 25 in the original matrix).

Alive/Dead matrix

1	1	0
0	0	0
0	0	1

Transformed to

1	1	1

2	2	1
0	1	0

Or [1, 1, 1, 2, 2, 1, 0, 1, 0]

Unfortunately, random forest was not able to learn the problem well - it returned 100% 0 predictions. It did this when we passed it the original grid and the transformed neighbors vector.

The final model we attempted was a bidirectional LSTM RNN. RNN, or recurrent neural network, is a class of artificial neural networks that has internal memory. Recurrent means that outputs of the current input depend on the past computation, which is saved in memory. LSTM, or long short term memory, is an extension of RNN, which allows neurons to process previous information that was computed farther back than the most recent. Bidirectional is another extension on this, where each bidirectional layer trains two LSTM's, with the second training on a reversed copy of the input. We hoped that adding bidirectional layers would allow the model to gain further insight into the rules of the game of life, as it will see game boards transform in forward and backwards time.

```
Model: "sequential_54"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
bidirectional_98 (Bidirectio (None, 625, 96)           19200
_________________________________________________________________
bidirectional_99 (Bidirectio (None, 625, 96)           55680
_________________________________________________________________
dense_54 (Dense)             (None, 625, 1)            97
=================================================================
Total params: 74,977
Trainable params: 74,977
Non-trainable params: 0
```

Initially, we used a single model for RNN, passing in all data irregardless of the delta value. We gave it two bidirectional LSTM layers, with 64 units each.

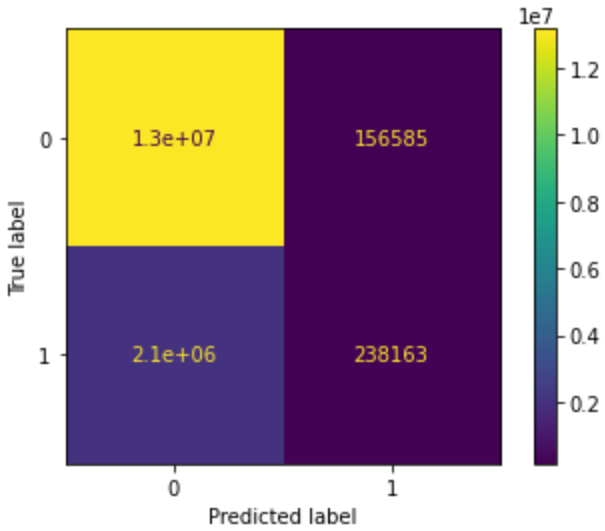
However, for our final RNN model we created 5 separate models and trained them on subsets of the data based on their delta value. We then passed in subsets of the test data separately to each model and concatenated them together at the end.

5. Evaluation & Result analysis

The worst model for this problem was random forest. Although we initially received an accuracy score of 85%, we realized this was due to the imbalance in the dataset. Our model only predicted values of 0, and 0 was the majority class in the target set. This is likely due to how sparse the input vectors were, whether it be in the original state or in the neighbors form.

The first CNN model did alright but still had room for improvement. Although accuracy was at 84.8% for test data from the training dataset, when compared to other Kaggle entries, it ranked at 186 out of 188. With the second CNN model, things didn't fare much better. Although accuracy went up, we still ranked at 186. After review, we determined that this was primarily due to the fact that we excluded the *delta* column from the model feature list. Adding an additional dimension proved to be difficult to integrate into the existing model and as such we excluded it. We did try to train 5 separate models, one for each *delta* and then combine the results into a single output, but it did not give us proper outputs. Due to training time being 4 hours for this model and the time constraints we faced towards the end of this project, we did not proceed further with troubleshooting that model implementation.

RNN performed better, as it was able to successfully predict 1’s in the target set. Unfortunately, it still struggled to predict the 1’s accurately, as there were 2.3 million 1’s in total and it was only able to predict 238,000 accurately.



A possible extension we could have done, given more time, would be to manually add vectors based on game board states between the start and stopping (calculating them according to the rules of the game). This could have given the model more information on the single steps between two states.

Our final kaggle score for RNN was 0.1433, which was in 120th place out of 188 entries. The table below compares our model implementations with scoring of Mean Absolute Error (MAE). Random Forest is excluded due to it only predicting 0's

Model	CNN (6 layer)	CNN (9 layer)	RNN
MAE	0.18406	0.18243	0.1433

The most accurate models for this competition used techniques such as genetic algorithms, Z3 constraints, SAT solvers, and simulated annealing. However, these methods were all beyond the scope of this class, with none of us having had any experience with them. As such, we chose not to employ these methods.