Reversing Conway's Game of Life

Jim Posen (jep37), Dennis Li (dkl12)

1 Abstract

The Game of Life is a cellular automata in which cells exemplifies deterministic chaos. Running the game of life backwards in time is a problem which has received little attention. In this paper we investigate the application of various machine learning techniques reverse the game state by one step. We used both an MCMC based approach and a technique with iterated MAP estimates of cells. Results were validated by simply running the boards in the forward direction. The iterated MAP approach worked well and converged quickly, especially with sparsely populated boards. On more chaotic, densely populated boards the algorithm reached a good approximation. Simulated annealing was ineffective due to the high dimensionality of the search space.

2 Introduction and Motivation

The game of life (GoL) is a cellular automata created by John Conway. The board consists of cells that are each either alive (1) or dead (0). The next state of the board is then determined by simple rules: (a) Any live cell only remains lives if it has two or three live neighbors because of under/over population. (b) Any dead cell with exactly three live neighbors becomes a live cell through reproduction. (c) Otherwise, the cell is dead. But how does one run it in the reverse direction? Kaggle poses this as a question: "If machine learning (or optimization, or any method) can predict the game of life in reverse." [5].

A naive approach to the problem is a brute force solution but the state space grows exponentially larger with a bigger board. In the case of the Kaggle competition data, the boards are 20×20 which means there are 2^{400} possible boards which is computationally impossible to compute directly and so must turn to machine learning and statistics.

The only directly related work online or in literature was a blog post and video by Bickford [2]. Bickford proposes a number of different heuristics in which the board is broken up into smaller pieces but as the author points out, it quickly runs into difficulties with larger patterns and took "several days to run" [2].

Another attractive alternative is to look towards the field of image analysis instead. Many analogies can be drawn between the challenge at hand and binary pixel classification. Each cell is classified into one of two states. Both boards and images can be represented by a $2-D$ grid. A popular and natural method for binary pixel classification employ Markov Random Fields (MRFs) [1, 7, 6]. However, the analogies end here. The primary goals of research in image analysis is in image denoising, image reconstruction, and smoothing which assume certain properties not present in the system of GoL, such as smoothness between neighboring pixels.

There are also important observations/factors inherent to the particular instance of GoL. Namely, the boards are 20×20 and have un-stitched edges. In addition, the boards have all been "warmed up" for five iterations which means certain board configurations are impossible.

The methods we describe below are solutions to the problem of finding an exact predecessor state to a Game of Life board.

3 Models and Methods

3.1 Mathematical Definition

Let Y represent a future board state, and X represent the initial board state. Y is a vector with $Y_i \in \{0, 1\}$. X is a vector with $X_i \in [0, 1]$. X_i represents the probability that the cell is alive. n_i represents the eight neighboring cells of X_i. The function $p_k(n_i)$ is the probability that exactly k members of the set n_i are alive. Finally, let $C^k_{n_i}$ be the set of all k-combinations of n_i.

$$p_k(n_i) = \sum_{c \in C^k_{n_i}} \prod_{j \in n_i} x_j^{1(j \in c)} (1 - x_j)^{(1-1(j \in c))} \tag{1}$$

Using the GoL rules outlined in the introduction, we give the likelihood function of board Y.

$$p(y|x) = \prod_i \text{Bern}(y_i\,; p_3(n_i) + x_i p_2(n_i)) \tag{2}$$

3.2 MAP over the entire board

Our first approach was to find the maximum a posteriori (MAP) estimate of the state x. By imposing a prior that makes impossible boards less likely, we hypothesized that we could create an algorithm that could quickly converge to a valid state. There is no prior distribution conjugate to the likelihood, so the MAP estimate cannot be found analytically.

Simulated annealing was employed to compute the maximum of the posterior distribution whose energy function is the log posterior probability. To compute the posterior probability of state x, we can write the log likelihood as shown below.

$$\log \mathcal{L}(x) = \sum_x y_i \log(p_3(n_i) + x_i p_2(n_i)) + (1 - y_i) \log(1 - p_3(n_i) - x_i p_2(n_i)) \tag{3}$$

3.2.1 Dirichlet Prior

For the MAP estimation to work better than MLE, the prior distribution must assign higher probability to states generated by the Game of Life. For each state, one can compute the number of live neighbors each cell has. The number of cells with a given number of neighbors seemed like a comparable statistic between the boards in the training set. Instead of using the raw counts, we added one to each count and normalized so that the sum is 0. The prior distribution over this sufficient statistic is a Dirichlet distribution.

The parameter α for the Dirichlet distribution was found using MLE estimation using the training boards. We used gradient descent to maximize the likelihood of the training boards with respect to the parameter alpha [4].

Data: y
Result: x
initialize x;
$T := T_0$;
while *not converged* **do**
  $x' \leftarrow x + \mathcal{N}(0, \sigma)$;
  $e' \leftarrow \log \mathcal{L}(y|x') + \log p(x'|\theta)$;
  $e \leftarrow \log \mathcal{L}(y|x) + \log p(x|\theta)$;
  $a \leftarrow$ random number from 0 to 1;
  if $a < \exp\left(\frac{e'-e}{T}\right)$ **then**
    $x \leftarrow x'$;
  end
  $T \leftarrow T \cdot C_k$;
end

3.3 MAP over each cell

We saw that performing a random walk over the space of all possible states would not converge. Our next approach was to assume a starting state x and iteratively compute the MAP estimate for cell x_i given x_{-i}. This can be computed with the assistance of root finding numerical methods.

The posterior distribution we use here is a beta. The reasons for this is twofold: 1) the log of a beta distribution takes the same form as the likelihood function and 2) we can use the beta to add probability weight to 0 and 1 when $\alpha, \beta < 0$.

One problem with this technique is that by updating one cell at a time, the first cells to be randomly updated cells may be assigned to 0 or 1, which introduces a bias. We used the same method as simulated annealing by having a temperate value T which controls the prior parameters $\alpha_0 e^{C_k T}$ and $\beta_0 e^{C_k T}$. During the early iterations, the probability mass of the prior is in the center, but as the model "cools down" after many iterations, the probability mass of the prior moves to 0 and 1.

First we must show some properties of the function p_k. Assume cells i and j are neighbors and $k > 0$. We let $n_{j,-i}$ represent the set $n_j - \{i\}$, or the set of neighbors of j excluding i. This follows from equation 1.

$$p_k(n_j) = x_i p_{k-1}(n_j - \{i\}) + (1 - x_i) p_k(n_j - \{i\}) \tag{4}$$

$$\frac{dp_k(n_j)}{dx_i} = p_{k-1}(n_{j,-i}) - p_k(n_{j,-i}) \tag{5}$$

Now we can calculate the MAP for cell x_i. Starting from equation 3, we compute the log posterior as a function of x_i.

$$\begin{aligned}
\log p(x_i|x_{-i}, y) &= \sum_j \left[y_j \log(p_3(n_j) + x_j p_2(n_j)) + (1 - y_j) \log(1 - p_3(n_j) - x_j p_2(n_j)) \right] \\
&\quad + (\alpha - 1) \log x_i + (\beta - 1) \log(1 - x_i) \qquad (6) \\
&= y_i \log(p_3(n_i) + x_i p_2(n_i)) + (1 - y_i) \log(1 - p_3(n_i) - x_i p_2(n_i)) \\
&\quad + \sum_{j \in n_i} y_j \log \left[x_i (x_j (p_1(n_{j,-i}) - p_2(n_{j,-i})) + p_2(n_{j,-i}) - p3(n_{j,-i})) + p_3(n_{j,-i}) + x_j p_2(n_{j,-i}) \right] \\
&\quad + (1 - x_j) \log \left[1 - x_i (x_j (p_1(n_{j,-i}) - p_2(n_{j,-i})) + p_2(n_{j,-i}) - p_3(n_{j,-i})) - p_3(n_{j,-i}) - x_j p_2(n_{j,-i}) \right] \\
&\quad + (\alpha - 1) \log x_i + (\beta - 1) \log(1 - x_i) + \text{constant} \qquad (7) \\
\frac{d \log p(x_i)}{dx_i} &= y_i \frac{1}{x_i + \frac{p_3}{p_2}} + (1 - y_i) \frac{1}{x_i + \frac{p_3 - 1}{p_2}} \\
&\quad + \sum_{j \in n_i} y_j \frac{1}{x_i + \frac{p_3 + x_j p_2}{x_j (p_1 - p_2) + p_2 - p3}} + (1 - y_j) \frac{1}{x_i + \frac{p_3 + x_j p_2 - 1}{x_j (p_1 - p_2) + p_2 - p3}} \\
&\quad + \frac{\alpha - 1}{x_i} + \frac{\beta - 1}{x_i - 1} \qquad (8) \\
0 &= \frac{1}{x_i + \frac{p_3 - 1 + y_i}{p_2}} + \sum_{j \in n_i} \frac{1}{x_i + \frac{p_3 + x_j p_2 - 1 + y_j}{x_j (p_1 - p_2) + p_2 - p3}} \\
&\quad + \frac{\alpha - 1}{x_i} + \frac{\beta - 1}{x_i - 1} \qquad (9) \\
& \qquad (10)
\end{aligned}$$

We find that we can compute the log posterior quickly and maximize it by finding roots of the derivative. The derivative $\frac{d \log p(x_i|y_i)}{dx_i}$ is of the form $\sum_i \frac{1}{x - a_i}$. We know that when $\alpha, \beta > 1$ that there is exactly one root on the interval $[0, 1]$. Since it is bounded, we can use bisection or Newton's method to find the root, and thus the global maxima. The algorithm proceeds as follows.

Data: y
Result: x
$T := T_0$;
$x :=$ vector of random samples from Beta(T, T);
while *not converged* **do**
  $i \leftarrow$ random cell;
  $x_i \leftarrow \hat{x}_{iMAP}(x, y, T)$;
  $T \leftarrow TC_k$;
end

4 Results

4.1 Simulated Annealing

The first step was to use the training set to learn the parameter α for the prior distribution with gradient descent. These values correctly assign a higher probability to having fewer neighbors.

Given the prior distribution, we ran $10,000$ iterations of simulated annealing. This took about 8 hours to run and the state did not come close to converging. Reflecting on this approach, performing a random walk on a high dimensional space is unlikely to ever converge [3, p322-331].

4.2 MAP over cells

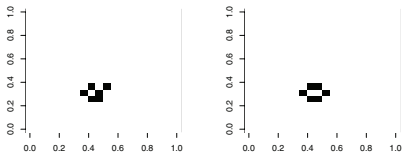

Figure 1: Left: Predicted start board, Right: Stop board

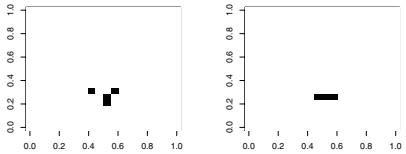

Figure 2: Left: Predicted start board, Right: Stop board

The cell based algorithm was initially run against sparsely populated, simple configurations. Convergence of the start board cell probabilities to $\{0, 1\}$ was reached reached quickly.

Observe that running the start board in the forward direction leads to the stop board, as intended. Shown in Fig 4.2, 4.2, left, are one of several start boards found which could generate the stop boards (Fig 4.2, 4.2 right).

While the cell based algorithm performed well on small patterns, on the sample boards given in the Kaggle training set, the algorithm failed to converge on a state with discrete state assignments to each cell. The results of running on board 22 of the training set are shown in Figure 3. The top-left board is our resulting start board, and the top right board is our estimate advanced one time step. The bottom left is the actual start board and the bottom right is the actual stop board.

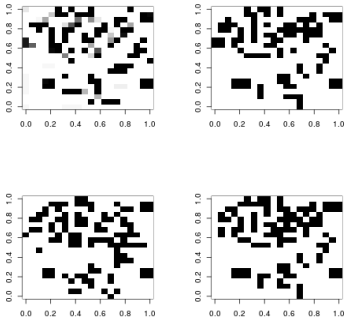

Figure 3: Top: Estimations, Bottom: Actual boards

5 Conclusions

We were able to successfully implement an iterated maximum a posteriori estimation approach to create optimal soft assignments to cells in the Game of Life. In the case of sparsely populated boards, MAP proved so effective that the final soft cell assignment values of being alive would converge onto integer values. Running the predicted initial boards in the forward direction, without further processing through the standard rules of GoL then yielded the original stop board.

In the case of chaotic boards, MAP was less effective. As illustrated in Figure 3, the final values did not always converge to 0 or 1. Instead the larger solutions would converge to a non-deterministic steady state. However, by rounding the probabilities and evolving the boards we produced results qualitatively similar to the desired boards.

Simulated annealing was an attempt to use an MCMC method to find the maximum a posteriori estimate of the board without updating one cell at a time. This method was also attractive because it did not require a conjugate prior. Unfortunately, random walks in high dimensional spaces caused issues with convergence and we were not able to produce meaningful results.

References

[1] Marc Berthod, Zoltan Kato, Shan Yu, and Josiane Zerubia. Bayesian image classification using markov random fields. *Image and Vision Computing*, 14(4):285 – 295, 1996.

[2] Neil Bickford. Reversing the game of life for fun and profit. http://nbickford.wordpress.com/2012/04/15/reversing-the-game-of-life-for-fun-and-profit/, 2012.

[3] Steven R. Finch. *Mathematical Constants*. Cambridge University Press, 2003.

[4] Jonathan Huang. Maximum likelihood estimation of dirichlet distribution parameters.

[5] Kaggle. Reverse the arrow of time in the game of life. http://www.kaggle.com/c/conway-s-reverse-game-of-life/, October 2013.

[6] M. Malfait and D. Roose. Wavelet-based image denoising using a markov random field a priori model. *Image Processing, IEEE Transactions on*, 6(4):549–565, 1997.

[7] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.

6 Appendix

```
data <- read.csv("./train.csv", header=TRUE, sep=",", stringsAsFactors=FALSE)
size <- 20

T0 <- 10
C.k <- .9999
eps <- 0.00001


for (row in c(22)) {
    start <- as.numeric(data[row,3:402])
    end <- as.numeric(data[row,403:802])
}

neighbors <- lapply(1:size^2, function (i) {
  r <- (i - 1) %/% size
  c <- (i - 1) %% size

  n <- c()
  for (r.p in (r-1):(r+1)) {
    for (c.p in (c-1):(c+1)) {
      if ((r.p == r && c.p == c) ||
          r.p < 0 || r.p >= size ||
          c.p < 0 || c.p >= size) {
        next
      }
      n <- append(n, r.p * size + c.p + 1)
    }
  }
  n
})

neighbor.probs <- function (x, i, without=NULL) {
  cells <- neighbors[[i]]
  if (!is.null(without)) {
    cells <- cells[-which(cells == without)]
  }
  sapply(cells, function (j) { x[j] })
}
```

```
mle <- function(x, y, i, temperature) {
  n.p <- lapply(neighbors[[i]], function (j) {
    probs <- neighbor.probs(x, j, without=i)
    sapply(1:3, function (k) { pn(probs, k) })
  })

  probs <- neighbor.probs(x, i)
  p <- sapply(1:3, function (k) { pn(probs, k) })

  a <- mapply(function (j, p) {
    c(x[j] * (p[1] - p[2]) + p[2] - p[3], p[3] + x[j] * p[2])
  }, neighbors[[i]], n.p, SIMPLIFY=F)

  alpha <- temperature
  beta <- temperature * 7

  if (temperature < 1) {
    log.likelihood <- function (x.i) {
      lkh <- sum(mapply(function (j, v) {
        pr <- max(0, min(1, c(x.i, 1) %*% v))
        log(dbinom(y[j], 1, pr))
      }, neighbors[[i]], a))
    }

    maxima <- c(0, 1)
    values <- sapply(maxima, log.likelihood)
    if (max(values) == -Inf) {
      log.posterior <- function (x.i) {
        log.likelihood(x.i) + (alpha - 1) * log(x.i) + (beta - 1) * log(1 - x.i)
      }
      optimize(log.posterior, lower = eps, upper = 1 - eps, maximum = T)$maximum
    }
    else {
      maxima[which.max(values)]
    }
  }
  else {
    coefs <- mapply(function (j, v) {
      (v[2] - 1 + y[j]) / v[1]
    }, neighbors[[i]], a)
    coefs <- append(coefs, (p[3] - 1 + y[i]) / p[2])
    coefs <- coefs[coefs != -Inf & coefs != Inf & !is.nan(coefs)]

    f <- function (x.i) {
      dlkh <- sum(sapply(-coefs, function (c.i) {
        1 / (x.i - c.i)
      }))
      dlkh + (alpha - 1) / x.i + (beta - 1) / (x.i - 1)
    }

    uniroot(f, lower = eps, upper = 1 - eps)$root
  }
}

next.state <- function(x) {
  x <- round(x)
  sapply(1:length(x), function (i) {
    alive <- sum(sapply(neighbors[[i]], function (j) x[j]))
    if (alive == 3 | (alive == 2 & x[i])) 1 else 0
  })
```

```
}

find.mle <- function(y, n, x = NULL, temp = T0) {
  if (is.null(x)) {
    x <- rbeta(400, temp, temp * 7)
  }
  for (j in 1:n) {
    print(j)
    i <- sample(1:400, 1)
    x[i] <- mle(x, y, i, temp)
    temp <- temp * C.k
  }
  x
}

pn <- function(probs, n) {
  if (length(probs) < n) {
    return(0)
  }
  sum(apply(combn(1:length(probs), n), 2, function (idx) {
    prod(sapply(1:length(probs), function (i) {
      dbinom(i %in% idx, 1, probs[i])
    }))
  }))
}

plot.nth <- function(grid, start, end) {
  par(mfrow=c(2,2))
  image(matrix(grid, nrow=size, ncol=size), col=gray((32:0)/32))
  image(matrix(next.state(grid), nrow=size, ncol=size), col=gray((32:0)/32))
  image(matrix(start, nrow=size, ncol=size), col=gray((32:0)/32))
  image(matrix(end, nrow=size, ncol=size), col=gray((32:0)/32))
}
```