# Abstract

What makes a rational actor rational?

\*\*placeholder: I will discuss my conclusions and connect my findings to

# Chess with thinking pawns

All of us are decisionmakers, and all of our decisions implicate each other. The advent of modern technologies of communication, production, and consumption have brought into sharp relief the relevance of economic laws to our daily lives. Workplaces, schools, and social spaces are increasingly defined by scores, competition, and rules; test-score-based pay raises have transformed the teaching profession into a game of statistics, and data-driven parenting techniques have become ubiquitous (Levitt & Dubner, 2011). Our cumulative pursuit of profit is destroying our planet because each has more to gain from exploiting resources than from conservation (Newkirk, 2016). Economics is inextricably embedded in the policymaking process, which determines our taxes, healthcare, and prisons. Life becomes a game, and so we are left to wonder how best to play. The task at hand is to build the tools to find our answers.

One of the most descriptive tools we have in life is the study of life itself: biology. Although it has not been widely adopted, evolutionary psychology can provide powerful insight into the nature and origin of human behavioral traits (Burke, 2014). Analyses of precisely why these traits are successful in economically scarce situations governed by asymmetrical incentives is the wheelhouse of game theory, which makes it especially useful in understanding the workings of human societies because they are often extremely asymmetrical and feature constant competition (Gibbons, 1997). These models alone, however, fail to capture probably the most unique aspect of human societies: the human mind. The quest to imitate human intelligence on a computer drove scientists to develop a technique known as deep-learning – a simulation of a single brain function built using increasingly complex “abstractions” which build up simple concepts into more descriptive heuristic evaluators (Goodfellow, Bengio, & Courville, 2016).

Only recently have researchers begun to explore the intersections between these three fields. While the efficiency of genetic algorithms developed with game-theory in mind far outperforms previous evolutionary algorithms in speed of training and accuracy, they still require explicit programming of meta-parameters, such as the mechanisms for imitation, belief, and communication (Yang, 2017). The research does not use a deep-learning framework as “genetic material,” which means while current techniques can deliver interesting results by observing the interactions of a large quantity of simple actors (such as an overall model of a market’s equilibrium price), they miss out of observing more direct interactions of complex actors (Protopapas, Kosmatopoulos, & Battaglia, 2009). Those kinds of nonlinear interactions are necessary to accurately simulate how intelligent actors interact in real life due to intrinsic variations and fluctuations in how we think, the fact that nobody is perfectly rational, and contextual psychological effects such as paranoia or social awkwardness (Osoba & Davis, 2018). Likewise, although complex machine thinking has indeed been achieved through neuro-evolution of deep networks (e.g., David & Greental, 2017), these methods are unable to comprehensively simulate a system of actors; they may be able to model and gradually evolve a population of neural networks, but the fitness systems used are not rooted in game theory and thus are closer to test scores than actual natural selection. Despite these limitations, an intersection between these two fields has the potential to greatly enhance existing studies of game theory by more accurately modelling intelligent decision-makers. Specifically, applying neuro-evolution to deep-learning can greatly speed up selection of hyper-parameters and make the task of tuning the network much easier, such as the number of neurons or the expression used to represent a model’s accuracy (Miikkulainen et al., 2017). Particularly in the context of games-playing, deep learning might outperform other methods because it can easily learn to recognize features in a variety of contexts, instead of relying on a more calcified initial input layer to generate results. For example, recognizing that a particular set of pixels constitutes a particular shape, regardless of its position on the screen, might allow a deep network playing Tetris to succeed where a Bayesian network based solely on discrete probabilities might fail. Deep learning, unlike other methods, makes better use of massive data sets because it does not require a scientist to intervene and label training data, but rather develops its own organic understandings (Mahapatra, 2018). Interestingly, certain pure genetic algorithms which don’t even incorporate neural networks in their architecture have been able to outperform deep learning networks in myriad Atari games (Dar, 2018). As it stands, these genetic algorithms rely on purely random generation of code, which means it is impractical to apply them to more complicated, nonlinear games like Dota 2 or Fortnite, which have a multitude of players and combat features. This means deep learning, which could simplify such complex features, might be combined with genetic algorithms which can efficiently process those simplified features.

Given the potential of an intersectional study using genetic algorithms, game theory, and deep learning, I propose a simulated game of competing deep-neural networks, iteratively evolved through artificial selection, cross-over and mutation, to shed light on economic theory when the actors are irrational and developing. I hypothesize that game-theory based predictions of the networks’ behavior will be mostly accurate, but certain rogue machine intelligences will think in a revolutionary way which defies explanation.

# Instrumentation and Variables

I use the Haskell programming language for this experiment (Jones, 2003). As a purely “functional” language, its inputs and outputs are fully predictable. This is opposed to a state-based or “imperative” language, such as Java, where internal randomness or errors in allocation of memory might crash a program mid-operation. Instead of risking memory loss in the middle of a crucial program execution, Haskell’s compiler catch errors before the program runs, almost guaranteeing safety. Haskell’s terse syntax and support for rapid composition of functions means relatively complex programs can be written very efficiently. This is important for the present project due to the multilayered nature of the algorithms involved. For example, being able to “link” functions together from input to output makes training a typical neural network much easier, because one can calculate the initial base “gradient” (essentially a small tweak to the network which makes it incrementally more accurate) with a single function and then pass the result to other functions which apply that gradient to the neural network proper, which is then passed along to another function which verifies if the new network has indeed improved. Especially in this context, where I layer deep learning, genetic algorithms, and game theory, being able to port functions together is vital, in a very literal sense, to explore interdisciplinary ideas.

I employ Haskell bindings of Google’s TensorFlow– a piece of machine learning software that nicely abstracts away most of the complex mathematics behind deep learning (Abadi et al., 2016).

I use Atom and Emacs as my text editors because I am familiar with both and they have excellent inbuilt support for Haskell () and enable fast editing using convenient keyboard shortcuts.

I be taking algorithms and equations from the MIT Press guide to deep learning (Goodfellow et al., 2016).

For debugging and compilation, I rely on the standardized Haskell platform toolkit, including the Glasgow Haskell Compiler and specifications (Himmelstrup, 2006).

Finally, version control is handled by git, a tool which helps keep track of changes and updates to one’s projects.

Experimental Procedures

First, I program basic mathematical functions which serve me throughout experimentation. This includes a linear algebra library which handles matrix transposes, multiplication, and working with eigenvalues (Appendix A). These are necessary to facilitate the calculation of a neural network’s final output given inputs. I also create a statistics and probability library; these are useful in game-theory calculations and to randomize mutations and cross-over during the evolution phases of my project. Finally, I program critical number-manipulation algorithms, such as the equations of a neural network’s accuracy or its “activation functions,” which help make a network more flexible or stable in what it can represent by mapping its internal calculations to functions with unique shapes.

Second, I create the architecture for the neural networks in Haskell (Appendix B). I create individual data types to represent a single perceptron, a whole layer of perceptrons, and finally an entire neural network (perceptrons are a representation of a human neuron). The neural network will be convolutional (pre-programmed to handle image-based data efficiently) because the game I create will have data in a two-dimensional image form.

Third, I program the deep-learning component of the networks, also using Haskell. This will likely be a moderate-to-small sized network due to processing constraints but will be powerful enough to play the game strategically because deep learning can simplify a complicated set of inputs into basic abstract concepts, which are then easily manipulated and interpreted to make decisions. This is the best method because traditional machine learning models are too rigid to effectively play a dynamic game with multiple players. The networks take as input the 15 by 15 grid which represents the game’s state, and output a “decision,” which I will describe later along with the game.

Fourth, I need a process for genetic evolution of each neural network. These networks calculate end results using a series of weights and biases, which are a series of matrices and vectors which represent the transformation of the input in a certain direction in multidimensional space. These are the “genes” of each algorithm. Varying these quantities, one can dramatically alter what a neural network does. Thus, randomizing these values and testing whether the resulting network has high fitness can yield an iteratively better “gene” pool through selection. Successful genetic codes will be passed on to later generations, with minor mutations (random values added to particular weights or biases), or cross-over (combining traits from two successful codes). In addition, the hyper-parameters of the networks, such as the number and size of layers they have, are also modified by the genetic process to achieve an efficient structure. The top two to ten networks in terms of fitness scores will be selected for, along with any exceptional scorers, such as those who eliminated the most players or survived for the longest time.

Fifth, I program two special players which are not neural networks. One will be a traditional AI which makes decisions based on optimal game-theory calculations. This will serve as a benchmark for peak performance. The other will decide its next move based on pure randomness: a benchmark for low performance. These “controls” will help me contextualize the fitness scores of the deep-learning networks.

Sixth, I will create the parameters of the game to be played. Much like the prisoner’s dilemma, there are multiple players who have a choice to defect (attack another player) or collaborate (remain peaceful) but may also choose to commit to an “alliance” with other players wherein they must attack anyone their ally attacks or is attacked by. A player who chooses to defect (attack) another player who is not attacking back will gain many points at their expense. However, if two players attack each other, neither will gain points. A player who is peaceful will gain a moderate amount of points at nobody’s expense. The more people a player chooses to attack, the less effective the attacks become. Thus, if the target attacks back, the original attacker will lose points and the retaliator will gain.

Statistical Procedures

Each tick, I will gather data for each player which includes the decision that player made, the number of players they have eliminated, their position on the board (and the type of block they were on), the values and structure of their neural networks, and their fitness scores.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Tick 1 | Tick 2 | Tick 3 | Tick 4 | Tick 5 |
| Player a | data | Data | data | data | data |
| Player b | data | Data | data | data | data |
| Player c | data | Data | data | data | data |
| Player d | data | Data | data | data | data |
| Player e | data | Data | data | data | data |

Using the R statistical language, I will perform linear regression to determine the correlation between the strategies chosen and the fitness earned. For example, if players who chose to fire bullets tended to do better, I might conclude that an aggressive (or deterrent) strategy is most effective. I will categorize strategies into clusters using an unsupervised neural network. Each cluster will contain neural networks that use similar strategies. Measuring the number of networks in each one, the average fitness of each cluster, and trends in these two variables, can reveal what strategies are selected for over time, or if new behaviors emerge altogether.

I accumulate 15 by 15 arrays which represents the whole grid at each tick. From this, I can compare regular players to the two special players by feeding the “image” of the game at a specific tick to one of the special player algorithms and observing whether they make the same choices as the regular player. For example, in one context, the perfect game-theory player might move upwards while player B would shoot a bullet downwards.

# Results

\*\*placeholder: I will create a “clustering” neural network which categorizes the different types of game strategies adopted among the population of players based off context-decision data. The entire record of a player’s moves would be their “decisions,” whereas the decisions that other players made in the previous turn provides the “context” for that move. Some players will tend to launch attacks in a peaceful context (nobody else is attacking), whereas others will form alliances during wartime. The individual categories can be put in a table with key attributes (listed):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Aggressive | Passive | Tit-for-tat-with-forgiveness | Tit-for-tat | Alliance-builder |
| Distribution (% of population) | 21 | 7 | 2 | 44 | 26 |
| Prob (attack | attacked) | 98 | 3 | 88 | 96 | 57 |
| Prob (attack | peace) | 100 | 1 | 4 | 9 | 12 |
| Prob (cooperate | peace) | 2 | 91 | 76 | 28 | 94 |
| Fitness score | 212 | 28 | 789 | 522 | 480 |

\*\*placeholder: I will conduct a chi-squared test to test for significant differences in the distribution of strategies between different training algorithms. Specifically, I will test genetic evolution, neural networks trained with gradient descent, and neural networks trained with gradient descent and genetic evolution.

\*\*placeholder: I will generate some quadratic regression graphs in excel to look for associations between final fitness scores and variables such as the tendency to attack, the size of neural network layers, or how dynamic the player is (i.e. the extent to which they make diverse decisions in response to diverse contexts)

\*\*placeholder: I will do a game-theory-based analysis of the game, identifying dominant strategies/Nash equilibriums and comparing them to the tactics adopted by the AI’s. Specifically,

# Discussion

\*\*placeholder: I will discuss the implications of

\*\*placeholder:

\*\*placeholder: In order to reach a broader audience, I will create a graphical representation of my project in the JavaScript programming language, which is commonly used for animations on the web. In particular, the p5.js library has built-in tools for drawing dynamic figures and shaped. I will represent the game played by the AI’s as a circle of inward-facing rectangles, as of chairs surrounding a circular table. Each rectangle will be of a different color in order to distinguish between players. Players who collaborate will be colored with stripes that represent the color of each ally.

# Appendix A

import Debug.Trace

import Data.List

data Tensor a = Tensor [Int] [a] deriving Show

instance Functor Tensor where

fmap function tensor@(Tensor shape info) = Tensor shape (map function info)

makeTensor :: [Int] -> [a] -> Tensor a

makeTensor shape info = Tensor shape info

--concat but I made it

wombat a

| a == [] = []

| tail a == [] = head a

| otherwise = (head a) ++ wombat (tail a)

drawMatrix :: (Show a) => Tensor a -> String

drawMatrix t1@(Tensor shape info) = foldl (\string row -> string ++ wombat (map (\a -> show a ++ (wombat $ take (bigL - (ls a)) $ repeat " ")) row) ++ "\n") "" rows

where rows = splitInto info (shape !! 1)

ls = length . show

bigL = succ $ maximum $ map (\a -> ls a) info

shapeOf :: Tensor a -> [Int]

shapeOf (Tensor shape info) = shape

infoOf :: Tensor a -> [a]

infoOf (Tensor shape info) = info

--splits Into chunks of size "size"

splitInto :: [a] -> Int -> [[a]]

splitInto list size

| length list == 0 = []

| otherwise = [take size list] ++ splitInto (drop size list) size

vm :: (Num a) => [a] -> [a] -> a

vm v1 v2 = sum $ zipWith (\*) v1 v2

pp :: [Int] -> [Int]

pp l = [0..(pred $ product l)]

sd :: Tensor a -> [Int] -> Tensor a

sd t@(Tensor shape info) fd = Tensor newShape (map (\a -> valueAt t $ (flip $ toCoordinate a newShape)) (pp shape))

where newShape = flip shape

flip l = map (\b -> l !! b) fd

valueAt :: Tensor a -> [Int] -> a

valueAt (Tensor shape info) coordinate = info !! (toIndex coordinate shape)

toIndex :: [Int] -> [Int] -> Int

toIndex coordinate dimensions = vm coordinate sizes

where sizes = (map (\a -> product $ drop a dimensions) [1..(pred $ length dimensions)]) ++ [1]

toCoordinate :: Int -> [Int] -> [Int]

toCoordinate index dimensions

| length dimensions == 0 = []

| otherwise = [div index dTail] ++ toCoordinate (mod index dTail) (tail dimensions)

where dTail = product $ tail dimensions

--simple 2d matrix multiplication. I split the first matrix into its rows, and the second into its columns, and multiple

mm :: (Num a) => Tensor a -> Tensor a -> Tensor a

mm (Tensor shape1 info1) t2@(Tensor shape2 info2) = Tensor [dim1, dim4] newInfo

where (dim1, dim2) = (shape1 !! 0, shape1 !! 1)

(dim3, dim4) = (shape2 !! 0, shape2 !! 1)

columns = (splitInto (infoOf $ sd t2 [1,0]) dim3)

newInfo = foldl (\newInfo row -> newInfo ++ map (vm row) columns) [] (splitInto info1 dim2)

--apply a function elementwise to the corresponding elements of two Tensors

te :: Tensor a -> Tensor a -> (a -> a -> a) -> Tensor a

te t1@(Tensor s1 i1) t2@(Tensor s2 i2) func = Tensor s1 (zipWith (func) i1 i2)

--elementwise tensor addition

ta :: (Num a) => Tensor a -> Tensor a -> Tensor a

ta t1 t2 = te t1 t2 (+)

--elementwise tensor multiplication: Hadamard product

tm :: (Num a) => Tensor a -> Tensor a -> Tensor a

tm t1 t2 = te t1 t2 (\*)

--tensor contraction. multiple of all the elements. ahh

--takes in tensors t1 and t2 and an "overlap," the number of dimensions shared between the two. The resultant tensor is t3. Say t1 is dimensions (a,b,c) and t2 is dimensions (c,b,d). We have overlap 2. This means that t3 has dimensions (a,d) because we "contract" the overlapping dimensions, aka do this: t3(a0,d0) = sum(b,c) (t1(a0,b,c) \* t2(c,b,d0))

--frontCoordinates are the list of every t1(a0,b,c), and backCoordinates are the list of every t2(c,b,d0). I then multiply every value at corresponding frontCoordinates/backCoordinates, such as (a0, 1 ,2) and (2, 1, d0), and the sum of that is the value of t3 at (a0, d0). I do this for every combination of a and d and map those results to a new tensor with shape (shape1 with the overlap dimensions dropped, + shape2 with overlap dimensions dropped)

tc2 :: (Num a, Show a) => Tensor a -> Tensor a -> Int -> Tensor a

tc2 t1@(Tensor shape1 info1) t2@(Tensor shape2 info2) overlap = Tensor newShape newInfo

where (fs1, bs1) = splitAt ((length shape1) - overlap) shape1

(fs2, bs2) = splitAt overlap shape2

newShape = fs1 ++ bs2

getValue newCoordinates = (sum $ (zipWith (\*) (map (valueAt t1 $) (frontCoordinates newCoordinates)) (map (valueAt t2 $) (backCoordinates newCoordinates))))

frontCoordinates newCoordinates = map (\a -> (take halfLength newCoordinates) ++ toCoordinate a bs1) (pp bs1)

backCoordinates newCoordinates = map (\a -> (reverse $ toCoordinate a fs2) ++ (drop halfLength newCoordinates)) (pp fs2)

newInfo = map (\newInfoIndex -> getValue $ toCoordinate newInfoIndex newShape) (pp newShape)

halfLength = div (length newShape) 2

# Appendix B

import LinearAlgebra

type Layer = ((Acti, Flavor), [Tensor Double])

data Acti = Tanh | Sigmoid | Relu deriving Show

data Flavor = Feed | Recur | Convol deriving Show

crossEntropy :: Double -> Double -> Double

crossEntropy activation target = negate $ (target \* (log activation)) + ((1.0 - target) \* (log $ 1.0 - activation))

sigmoid :: Double -> Double

sigmoid number = 1.0 / (1.0 + (exp number))

sigmoidD :: Double -> Double

sigmoidD number = negate $ (exp number) \* ((1 + (exp number)) ^ (-2))

type Transform a = ((Tensor a -> Tensor a), (Tensor a -> Tensor a))

feed :: (Num a) => Tensor a -> Tensor a -> Tensor a -> Tensor a

feed input weight bias = ta (mm input (sd weight [1,0])) bias

feedRelu w b = feed2 w b (\a -> maximum a 0.0)

feedSigmoid w b = feed2 w b (\a -> exp a / (exp a + 1.0)

--takes a layer as input and "applies" the layer to an input, with an output of activations

feedControl :: Layer -> Tensor Double -> Tensor Double

feedControl layer@((acti, flavor), datum) input = case flavor of Feed -> fmap acti $ feed input (datum !! 0) (datum !! 1) acti

Recur -> fmap acti $ feedRecurrent datum

Convol -> fmap acti $ feedConvolution datum

# Works Cited

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., … others. (2016). Tensorflow: a system for large-scale machine learning. In *OSDI* (Vol. 16, pp. 265–283).

Burke, D. (2014). Why isn’t everyone an evolutionary psychologist? *Frontiers in Psychology*, *5*, 910. https://doi.org/10.3389/fpsyg.2014.00910

Dar, P. (2018). Evolutionary Algorithm - The Surprising and Incredibly Useful Alternative to Neural Networks. Retrieved from https://www.analyticsvidhya.com/blog/2018/07/evolutionary-algorithm-perfect-useful-alternative-neural-network/

David, E., & Greental, I. (2017). Genetic Algorithms for Evolving Deep Neural Networks. *CoRR*, *abs/1711.0*. Retrieved from http://arxiv.org/abs/1711.07655

Gibbons, R. (1997). *An Introduction to Applicable Game Theory* (Technical Working Paper Series). https://doi.org/10.3386/t0199

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.

Himmelstrup, D. (2006). Interactive debugging with GHCi. In *Proceedings of the 2006 ACM SIGPLAN workshop on Haskell* (p. 107).

Jones, S. P. (2003). *Haskell 98 language and libraries: the revised report*. Cambridge University Press.

Levitt, S. D., & Dubner, S. J. (2011). *Freakonomics: A Rogue Economist Explores the Hidden Side of Everything*. HarperCollins. Retrieved from https://books.google.com/books?id=wNPnl5zYA-cC

Mahapatra, S. (2018). Towards Data Science - Why Deep Learning over Traditional Machine Learning? Retrieved from https://towardsdatascience.com/why-deep-learning-is-needed-over-traditional-machine-learning-1b6a99177063

Miikkulainen, R., Liang, J. Z., Meyerson, E., Rawal, A., Fink, D., Francon, O., … Hodjat, B. (2017). Evolving Deep Neural Networks. *CoRR*, *abs/1703.0*. Retrieved from http://arxiv.org/abs/1703.00548

Newkirk, V. R. (2016). Is Climate Change a Prisoner’s Dilemma or a Stag Hunt? Retrieved September 20, 2018, from https://www.theatlantic.com/notes/2016/04/climate-change-game-theory-models/479340/

Osoba, O., & Davis, P. (2018). *An Artificial Intelligence/Machine Learning Perspective on Social Simulation: New Data and New Challenges*. Santa Monica, CA. Retrieved from https://www.rand.org/pubs/working\_papers/WR1213.html.

Protopapas, M. K., Kosmatopoulos, E. B., & Battaglia, F. P. (2009). Coevolutionary Genetic Algorithms for Establishing Nash Equilibrium in Symmetric Cournot Games. *CoRR*, *abs/0905.3*. Retrieved from http://arxiv.org/abs/0905.3640

Yang, G. (2017). Game Theory-Inspired Evolutionary Algorithm for Global Optimization. *Algorithms*, *10*(4).