REVERSED NEURAL NETWORK - AUTOMATICALLY FINDING NASH EQUILIBRIUM

Anonymous authors
Paper under double-blind review

ABSTRACT

Contrary to most reinforcement learning studies emphasizing on approximating the output layer of a neural network to certain strategies, this paper proposes a reversed way for reinforcement learning. We call this "Reversed Neural Network". In short, after sufficiently training a canonical deep feed-forward neural network according to a strategy-and-environment-to-payoff table, we randomize part of the neurons in the input layer and propagate the error between the generated output and the desired output back to the part of the neurons in the "input layer" of the trained deep neural network recurrently. And we view the final form of the neurons in the "input layer" as the fittest strategy for a neural network.

1 INTRODUCTION

Contrary to most reinforcement learning studies (as in "Playing Atari with Deep Reinforcement Learning (Mnih et al., 2013)."), which emphasize on training a deep neural network to approximate its output layer to certain strategies and view the output layer of a deep neural network as its strategy in a certain natural environment, this paper provides a reversed point of view and also a revolutionary way of revisiting the traditional deep neural network as we know today.

Also, contrary to most Game Theory studies, which emphasize on the role of human experts to device smart mathematics algorithm (as in "When Machine Learning Meets AI and Game Theory (Agrawal & Jaiswal, 2012)."), this paper emphasizes on the role of machine and builds up the bridge between A.I. territory, Game Theory, and, hopefully, Control Theory and neural science.

The method here we will use is rather simple. In short, after training a deep neural network according to a strategy-and-environment-to-payoff (input-to-output) table that records possible strategies and the environments along with the consequential payoffs, then we randomize the strategy input in the "input layer" and propagate the error between the actual output and the desired output back to the strategy input of the same trained deep neural network recurrently to perform a task similar to "human deduction". And we view the final strategy input in the "input layer" as the fittest strategy for a neural network confronting the observed environment input from the world outside.

Since the information is sent back to the input layer in a reversed way like those in Hopfield Neural Network (although not exactly the same), we will call this "Reversed Neural Network". To avoid confusion with Recurrent Neural Network, we will use "Rev-NN" as the abbreviation for Reversed Neural Network and "R-NN" for Recurrent Neural Network.

By using the same trick, we can force Rev-NN to automatically generate the Nash Equilibrium in a Simultaneous Discrete Game as well as Sequential Discrete Game (this will be explained later).

Since Nash Equilibrium is the optimal combination of strategies for two players, which no player will be willing to deviate from, the ability to find the Nash Equilibrium in advance in a complex system indicates certain intelligence of the species. And so is the machine. Therefore, Rev-NN posses a certain intelligence needed to survive in a certain environment.

Future application of "Rev-NN" is tremendously abundant, which includes (but is not limited to) game A.I. reinforcement, Elo-rating-system, smart-grid manipulation, economic analysis, human policy consultancy, Game Theory, and Control Theory, etc.. For the sake of page limitation, we will only focus on its application on Simultaneous Discrete Game and Sequential Discrete Game in Game Theory for the time being.

2 ALGORITHM: REVERSED NEURAL NETWORK

To grasp the ideology behind Rev-NN, we will take a mosquito for an example as shown in Figure 1.

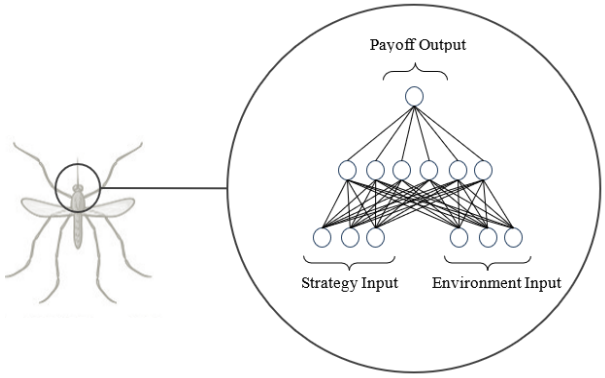
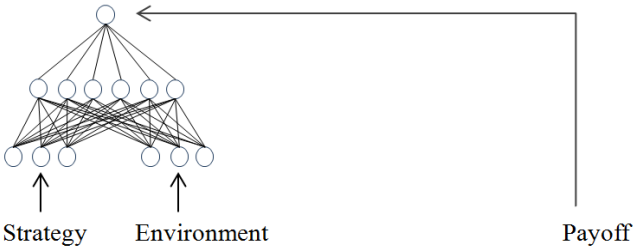


Figure 1: example of a simplified brain of a mosquito

Imagine the brain of a mosquito in Figure 1 consists of a deep neural network with only one input layer, one hidden layer, and one output layer. A part of the neurons in the input layer records information concerning the strategies that the mosquito has adopted in response to certain environment information, which is also instantaneously recorded by the other part of the neurons in the input layer as well. Meanwhile the output layer also instantaneously records the payoff to the mosquito when it adopts a certain strategy under a certain environment.

The brain of the mosquito is trained to memorize the consequential payoff of every combination of a certain strategy and a certain environment. For example, the consequence of seeing human's hand and not flying away is recorded. Of course, the consequence is death, in which case the payoff to the mosquito is set to [0] due to no life signal. Even though its hardware is damaged, the software in the cloud (or the eternal soul, arguably) of the mosquito records the consequential payoff of every combination and trains the brain (or the neural network) of the mosquito according to Figure 2 as below.



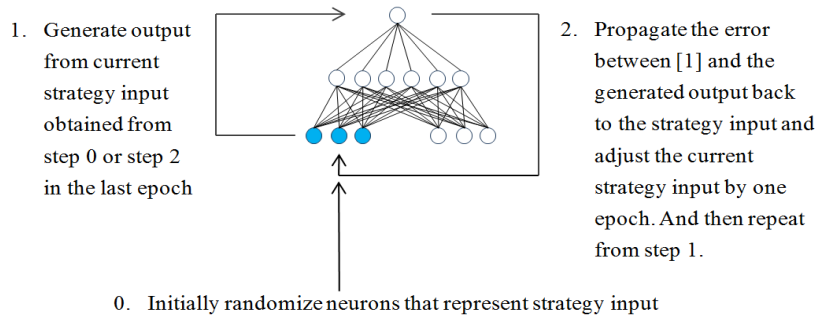
	Strategy	Environment		Payoff	
fly away	[1, 0, 0]	[1, 0, 0]	human hand exists	[1]	alive
stay	[0, 0, 1]	[1, 0, 0]	human hand exists	[0]	death
fly away	[1, 0, 0]	[0, 0, 1]	human hand not exist	[1]	alive
stay	[0, 0, 1]	[0, 0, 1]	human hand not exist	[1]	alive

Figure 2: example of the learning process in a mosquito's brain

When the brain (or the neural network) of the mosquito is trained well (by using Stochastic Gradient Descent) according to the table in Figure 2, upon seeing a human's hand at presence, the brain (or the neural network) of the mosquito must figure out what strategy it shall adopt in order to stay alive.

In another word, how could the neural network figure out a strategy that generates the payoff [1] when put into the neural network along with environment input [1, 0, 0]?

The answer that Rev-NN gives is rather simple and effective – Just randomize the strategy input in the input layer of the already-well-trained neural network and let the neurons that represent the strategy input to adjust itself gradually to render the neural network to generate output [1] by propagating the error between the generated output and the desired output [1] back to the neurons in the strategy input itself every time as long as the generated output (of the adjusted strategy input obtained in the last round) does not adhere to [1]. The whole process can be shown in Figure 3.



*** The desired output, the neurons that represent the environment input and the weight of the synapse are all intact and left unchanged during all the epochs ***

times	strategy	environment	actual payoff	desired payoff
0	[0.4, 0.5, 0.6]	[1, 0, 0]	[0.3]	[1]
100	[0.5, 0.5, 0.5]	[1, 0, 0]	[0.3]	[1]
200	[0.6, 0.5, 0.3]	[1, 0, 0]	[0.4]	[1]
300	[0.8, 0.6, 0.2]	[1, 0, 0]	[0.5]	[1]
…	…	…	…	…
N	[0.9, 0.6, 0.1]	[1, 0, 0]	[0.99]	[1]

Figure 3: example of the deduction process in a mosquito's brain

Finally, after fair epochs, the mosquito will obtain a strategy input that approximate [1, 0, 0] when the argmax index is one-hotted, which means the mosquito better just flies away. The information of the final strategy input will later be sent to other parts of the body to initiate the procedure of flying such as twitching wing tensors.

Imagine this simple three-layered deep neural network constitutes the very core of the mosquito's decision-making mechanism. It is a simple test code that we can easily finish in no time and testify the mechanism above.

Therefore, the Rev-NN algorithm can be dissected into two phases. The first is the training or learning phase, and the second is the deduction phase. In the training or learning phase, the mosquito learns about the consequence in response to the combination of strategies and environment (at the cost of its life, arguably). In the deduction phase, the mosquito must figure out a strategy that can make it stay alive when the hand of human is at presence.

If the mosquito is a satellite in the space, it can adjust its trajectory by initiating different ejectors according to the location signal it receives in order to getting back to the right trajectory (in order to generate an output of a desired location). Therefore, it might be an alternative method to the widely used Control Theory by the engineers in industry.

The pseudo code of Rev-NN likewise contains two parts. The first part is the training or learning phase, which is just the same as traditional feed-forward neural network with Back Propagation and Stochastic Gradient Descent as we all know. This is illustrated in Algorithm 1.

Then the most eccentric and interesting part comes afterward. In the second phase (deduction phase), the pseudo code can be written in the form of Algorithm 2.

Algorithm 1 : training phase for the mosquito example

$\mathbf{I} \leftarrow samples\,of\,(strategy, environment)$
$\mathbf{O} \leftarrow samples\,of\,payoff(strategy, environment)$
$\mathbf{W}_0 \leftarrow initialize\,randomized\,weights\,at\,time\,zero$
$\alpha \leftarrow learning\,rate$
$N_{epochs} \leftarrow number\,of\,epochs$
$N_{samples} \leftarrow number\,of\,training\,samples$
for t **in** $[N_{epochs}]$ **do**
    $index \leftarrow randomint(N_{samples})$
    $\mathbf{i} \leftarrow \mathbf{I}[index]$
    $\mathbf{o} \leftarrow \mathbf{O}[index]$
    $\mathbf{o}^{'} \leftarrow feedforward(\mathbf{i}, \mathbf{W}_t)$
    $\mathbf{W}_{t+1} \leftarrow \mathbf{W}_t - \alpha \frac{\partial}{\partial \mathbf{W}_t} E(\mathbf{o} - \mathbf{o}^{'})$
$\mathbf{W}_{trained} \leftarrow \mathbf{W}_{N_{epochs}}$

Algorithm 2 : deduction phase for the mosquito example

$\mathbf{i}_0^{strategy} \leftarrow initialized\,random\,strategy\,input\,at\,time\,zero$
$\mathbf{i}^{nature} \leftarrow initialized\,input\,from\,environment$
$\mathbf{o}^d \leftarrow initialized\,output\,which\,is\,desired$
$\alpha \leftarrow learning\,rate$
$N_{epochs} \leftarrow number\,of\,epochs$
for t **in** $[N_epochs]$ **do**
    $\mathbf{o}_t^{'} \leftarrow feedforward(concatenate(sigmoid(\mathbf{i}_t^{strategy}), \mathbf{i}^{nature}), \mathbf{W}_{trained})$
    $\mathbf{i}_{t+1}^{strategy} \leftarrow \mathbf{i}_t^{strategy} - \alpha \frac{\partial}{\partial \mathbf{i}_t^{strategy}} E(\mathbf{o}^d - \mathbf{o}_t^{'})$
$optimal\,strategy\,under\,\mathbf{i}^{nature} \leftarrow onehot(argmax(sigmoid(\mathbf{i}_{N_{epochs}}^{strategy})))$

For the sake of simplicity, we will summarize the deduction phase in just one single sentence - ***Tuning the neurons in the input layer to generate the desired output.***

3 Experiment I: Finding the Nash Equilibrium in a Simultaneous Discrete Game

The first experiment here we will examine is simply the extension of the notion of the mosquito example stated above.

If we take the mosquito and the hand of human as two players A and B in a Simultaneous Discrete Game in Game Theory, we can train a neural network according to the strategies-to-payoff table for player A and B and then tune the neurons in the strategy input for player A to generate the desired output for player A, and vice versa. After we interchangeably and iteratively tune these two sets of neurons each represents the strategy input for player A and B, these two sets of neurons will converge to the Nash Equilibrium.

For example, in Simultaneous Discrete Game, we often see the strategies-to-payoff table for player A and B as shown in Figure 4.

		Player B strategy	
		[1, 0, 0]	[0, 0, 1]
Player A strategy	[1, 0, 0]	[0.5, 0.5]	[1.0, 0.0]
	[0, 0, 1]	[0.0, 1.0]	[0.7, 0.7]

Figure 4: example of prisoner's dilemma

In the same manner, in the training or learning phase, we first train a neural network according to the strategies-to-payoff table as shown in Figure 4 for player A and B. This is illustrated in Figure 5 as below.

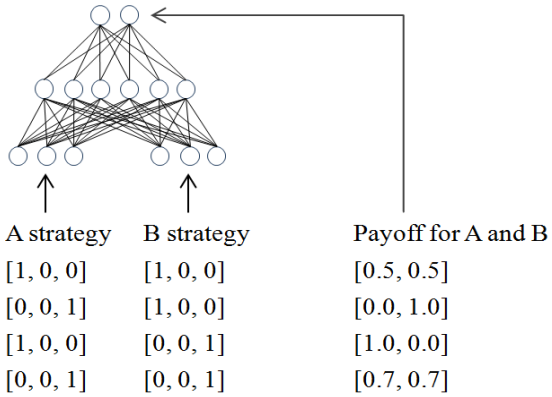


Figure 5: example of the learning process for prisoners

Second, in the deduction phase, since it is not a zero-sum game, we will tune the neurons which represent the strategy input for player A to generate the desired payoff output for player A, namely [1, *] (where "*" stands for whatever the payoff for player B already is), and then we will tune the neurons which represent the strategy input for player B to generate the desired output for player B, namely [*, 1] (where "*" stands for whatever the payoff for player A already is). We will keep this procedure for fair epochs as shown in Figure 6.

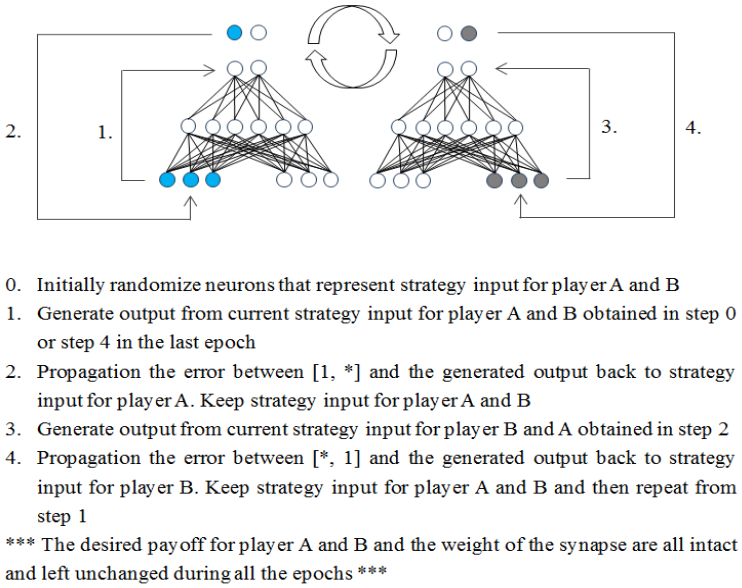


Figure 6: example of the deduction process for prisoners

After fair epochs, the neurons which represent the strategy input for player A will converge to [1, 0, 0] when the argmax index is one-hotted. And the neurons which represent the strategy of player B will also converge to [1, 0, 0] when the argmax index is one-hotted. This means that player A and player B will both adopt strategy [1, 0, 0], which is exactly the Nash Equilibrium in the strategies-to-payoff table in Figure 4.

In fact, this strategies-to-payoff table is exactly the famous Prisoner's Dilemma, and strategy input [1, 0, 0] is exactly the "betrayal" strategy. Rev-NN just automatically figures out the Nash Equilibrium in the famous Prisoner's Dilemma which is sadly that both players will still betray each other (as in "Primer in Game Theory (Gibbons, 1994).").

For more evidence and result for the experiment, please consult the appendix attached to this paper.

The pseudo code here can also be dissected into two phases, namely the training or learning phase and the deduction phase.

For the training phase, we simply duplicate the training phase in the mosquito example except that this time the environment input was replace by strategy input for player B, and the original strategy input was replaced by strategy input for player A. The output will be the consequent payoff for player A and B as shown in Algorithm 3.

Algorithm 3 : training phase for the prisoners' dilemma

$\mathbf{I}^{AB} \leftarrow samples\, of\, (player A strategy, player B strategy)$
$\mathbf{O}^{AB} \leftarrow samples\, of\, payoff(player A strategy, player B strategy)$
$\mathbf{W}_0 \leftarrow initialize\, randomized\, weights\, at\, time\, zero$
$\alpha \leftarrow learning\, rate$
$N_{epochs} \leftarrow number\, of\, epochs$
$N_{samples} \leftarrow number\, of\, training\, samples$
for t **in** $[N_{epochs}]$ **do**
    $index \leftarrow randomint(N_{samples})$
    $\mathbf{i} \leftarrow \mathbf{I}^{AB}[index]$
    $\mathbf{o} \leftarrow \mathbf{O}^{AB}[index]$
    $\mathbf{o}^{'} \leftarrow feedforward(\mathbf{i}, \mathbf{W}_t)$
    $\mathbf{W}_{t+1} \leftarrow \mathbf{W}_t - \alpha \frac{\partial}{\partial \mathbf{W}_t} E(\mathbf{o} - \mathbf{o}^{'})$
$\mathbf{W}_{trained} \leftarrow \mathbf{W}_{N_{epochs}}$

In the deduction phase, we simply tune the neurons which represent the strategy input for player A to generate the desired output for player A, and vice versa. We let Rev-NN repeat this movement for fair epochs, and then the Nash Equilibrium will be found as in Algorithm 4.

Algorithm 4 : deduction phase for the prisoners' dilemma

$\mathbf{i}_0^A \leftarrow initialized\, random\, input\, strategy\, for\, player\, A\, at\, time\, zero$
$\mathbf{i}_0^B \leftarrow initialized\, random\, input\, strategy\, for\, player\, B\, at\, time\, zero$
$p^{dA} \leftarrow initialized\, payoff\, which\, is\, desired\, by\, player\, A$
$p^{dB} \leftarrow initialized\, payoff\, which\, is\, desired\, by\, player\, B$
$\alpha \leftarrow learning\, rate$
$N_{epochs} \leftarrow number\, of\, epochs$
for t **in** $[N_{epochs}]$ **do**
    $\mathbf{o}_t^{'A} \leftarrow feedforward(concatenate(sigmoid(\mathbf{i}_t^A), sigmoid(\mathbf{i}_t^B)), \mathbf{W}_{trained})$
    $\mathbf{o}_t^{dA} \leftarrow \mathbf{o}_t^{'A}$
    $\mathbf{o}_t^{dA}[0] \leftarrow p^{dA}$
    $\mathbf{i}_{t+1}^A \leftarrow \mathbf{i}_t^A - \alpha \frac{\partial}{\partial \mathbf{i}_t^A} E(\mathbf{o}^{dA} - \mathbf{o}_t^{'A})$
    $\mathbf{o}_t^{'B} \leftarrow feedforward(concatenate(sigmoid(\mathbf{i}_{t+1}^A), sigmoid(\mathbf{i}_t^B)), \mathbf{W}_{trained})$
    $\mathbf{o}_t^{dB} \leftarrow \mathbf{o}_t^{'B}$
    $\mathbf{o}_t^{dB}[1] \leftarrow p^{dB}$
    $\mathbf{i}_{t+1}^B \leftarrow \mathbf{i}_t^B - \alpha \frac{\partial}{\partial \mathbf{i}_t^B} E(\mathbf{o}^{dB} - \mathbf{o}_t^{'B})$
$optimal\, strategy\, for\, player\, A \leftarrow onehot(argmax(sigmoid(\mathbf{i}_{N_{epochs}}^A)))$
$optimal\, strategy\, for\, player\, B \leftarrow onehot(argmax(sigmoid(\mathbf{i}_{N_{epochs}}^B)))$

As we all know, Nash Equilibrium is the combination of strategies that no player is willing to deviate from. Finding the Nash Equilibrium in advance indicates some certain intelligence in a system.

Why can Rev-NN automatically figure out the Nash Equilibrium? There is no definite answer. However, this paper proposes two possible explanations.

The first possible explanation is that, In Kakutani's fixed point theorem, a topology space kept mapped to itself (if closed, continuous) will converge to a certain point. This point is the fixed point under every mapping. This point is also the Nash Equilibrium (as in "Introduction to Topology: Pure and Applied (Adams & Franzosa, 2008)."). Possibly, strategy input for A and B under Rev-NN algorithm guarantee to converge to a certain point under the algorithm of Deep Learning after thousands of iterations.

The second possible explanation is that, imagine strategy input for A and B are two small ants in two different mountain sides, they both want to climb to the highest point of their own mountains and the highest points of the two mountains are in different locations, however they overlap each other (phantom space), and, wherever ant A moves, ant B will be carried to the same direction due to the quantum-entanglement-like phantom effect, vice versa. If ant A moves first and then ant B moves repeatedly, in the end, ant A and B will converge to a point, though this point is never the highest point in their own mountain sides.

4 Experiment II: Finding the Nash Equilibrium in a Sequential Discrete Game

The experiment so far touches upon only the combination of Rev-NN and Deep Feedforward Neural Network in Simultaneous Discrete Game.

However, there are still Recurrent Neural Network in A.I. and Sequential Discrete Game in Game Theory, and we can still combine Rev-NN and Recurrent Neural Network in Sequential Game.

Imagine a recurrent neural network with one fully connected input layer, hidden layer and output layer, and the interconnection between the hidden layers is still fully connected. Each layer is represented by a set of neurons marked in the shape of rectangle.

First, in the training or learning phase, we will train the RNN according to a game tree (shown in Figure 7) in Sequential Discrete Game in Game Theory. The whole training or learning phase is shown in Figure 8. This game tree may be generated by fixed program in a chess game or by trail-and-err in the real world like the case of mosquito.

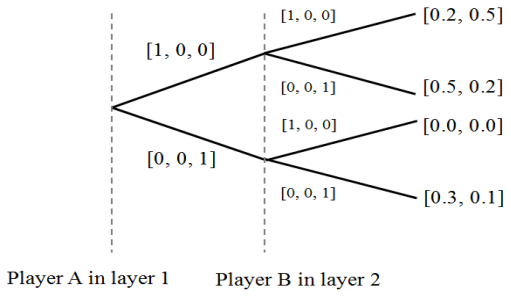


Figure 7: game tree for sequential game

In the deduction phase, we first randomize strategy input for A and B. Then, by using the notion of "Back Induction" in Game Theory (as in "Games of Strategy, Second Edition (Dixit & Skeath, 2004)."), we tune the neurons in the strategy input for player B in layer 2 to meet the desired output [*,1] for player B, and then we tune the strategy input for player A in layer 1 to meet the desired output [1,*] for player A. At last we repeat the process again, and the strategy inputs obtained in the last epoch will be kept to the next epoch. The whole procedure can be shown in Figure 9.

Finally, the final strategy input for player A in layer 1 in figure Figure 9 is the optimal strategy for player A in layer 1 in the real world. After player A chooses its strategy, in the next layer of the game in the real world, player B can exploit the same technic to obtain the optimal strategy for himself or

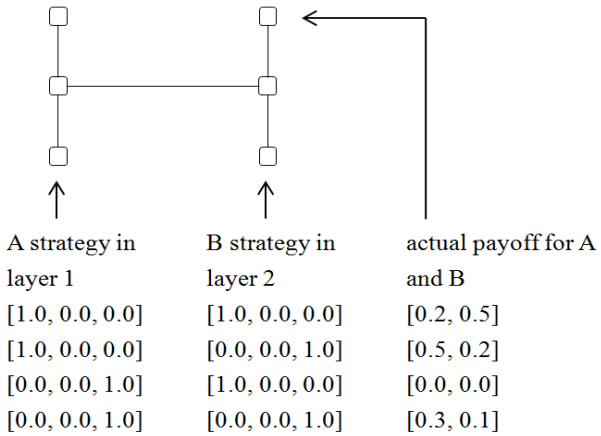
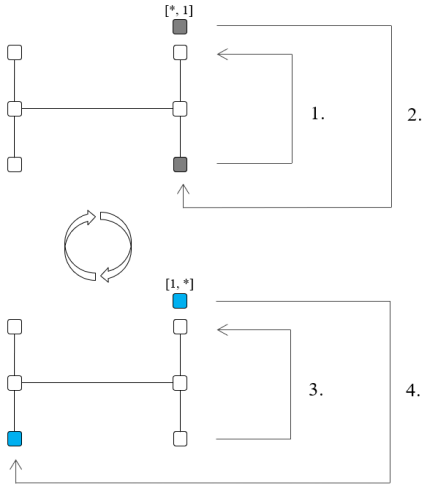


Figure 8: training or learning phase in Sequential Discrete Game



0. Initially randomize neurons that represent strategy input for player A and B
1. Generate output from current strategy input for player A and B obtained in step 0 or step 4 in the last epoch
2. Propagation the error between [*, 1] and the generated output back to strategy input for player B. Keep strategy input for player A and B
3. Generate output from current strategy input for player A and B obtained in step 2
4. Propagation the error between [1, *] and the generated output back to strategy input for player A. Keep strategy input for player A and B and then repeat from step 1

*** The desired payoff for player A and B and the weight of the synapse are all intact and left unchanged during all the epochs ***

Figure 9: deduction phase in Sequential Discrete Game

herself in layer 2. Both players will converge to the Nash Equilibrium (route) in Sequential Discrete Game in Game theory in the end. The same methodology also applies to Sequential Discrete Game with arbitrary numbers of layers.

5 CONCLUSIONS

The experiments shown here are not all inclusive, but the preliminary experiments indicate that Rev-NN can perform the similar task to human deduction by simply propagating error further back to the input layer. In this sense, the input layer of a neural network does not only serve as an information receiver (as we always envisage) but also an information sender, which information is further sent to other parts of the body such as muscles or even other neural networks to enhance further usage of the sent information.

The whole traditional deep neural network can be viewed as a gigantic memory reservoir of strategy, environment and payoff. In a word, a simple traditional deep neural network itself is already a memory pool, ready to have its information be extracted for later usage, and Recurrent Neural Network just augmented this function by allowing variable inputs to be considered. This finding might implement the notion of "memory" as we envisaged (as in "Deep Learning (LeCun et al., 2015).").

By back-propagating error back to the input layer, a traditional deep neural network forms an internal information exchange route similar to the internal information exchange in Hopfield Neural Network. The only difference is that the former uses Back Propagation algorithm while the latter exploits the notion of thermodynamics (Hopfield Neural Network can be viewed as a Restricted Boltzmann Machine with visible layers mapping to itself in supervised learning, while a Boltzmann Machine can be viewed as a Hopfield Neural Network with hidden layers and simulated annealing, the latter as in "Connectionist architectures for Artificial Intelligence (Fahlman & Hinton, 1987)."). This internal information from other pathways does not only train the synapses itself as we always envisage (as in "How neural networks learn from experience (Hinton, 1992)."), but also the source (neurons in the input layer) as well.

There are pros and cons to Rev-NN. First, Rev-NN is highly intuitive and can be easily coded. Second, Rev-NN can be seamlessly combined with other types of neural networks as long as the back-propagation algorithm or back-propagation-like algorithm is compatible with the latter (such as Rev-NN combined with Neural Turing Machine or LSTM). Third, Rev-NN can not only deal with discrete strategy but also continuous strategy (digit between 0 and 1).

However, in the process of experiment we find that in some application, Rev-NN requires heavy epochs on the training or learning phase in order to achieve high precision in the deduction phase.

There are still abundant discoveries that we have made, yet due to the limitation of the pages, we cannot introduce every discovery.

REFERENCES

Colin Adams and Robert Franzosa. *Introduction to Topology: Pure and Applied*. Prentice Hall Books, 2008.

Anurag Agrawal and Deepak Jaiswal. When machine learning meets ai and game theory. *cs229.stanford.edu*, 2012.

Avinash K. Dixit and Susan Skeath. *Games of Strategy, Second Edition*. W.W. Norton, 2004.

S. E. Fahlman and G. E Hinton. Connectionist architectures for artificial intelligence. *IEEE Computer*, pp. 100–109, 1987.

Robert Gibbons. *Primer in Game Theory*. Prentice Hall Books, 1994.

G.E. Hinton. How neural networks learn from experience. *Scientific American*, pp. 145–151, 1992.

VYann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521:436–444, 2015.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *NIPS Deep Learning Workshop*, 2013.

APPENDIX

Evidence of Intelligence for Rev-NN in Simultaneous Discrete Game:

For a four-layered pure and traditional deep feedforward neural network with the size of layers of 6 * 600 * 600 * 2, epochs of 1000, and learning rate at 0.1, the experiment results of the strategies-payoff tables with Nash Equilibrium underlined are listed below (all rounded to 2 digits behind decimal):

Table 1 – Prisoner's Dilemma (with NE underlined)

		Player B strategy	
		[1, 0, 0]	[0, 0, 1]
Player A strategy	[1, 0, 0]	<u>[0.5, 0.5]</u>	[1.0, 0.0]
	[0, 0, 1]	[0.0, 1.0]	[0.7, 0.7]

A strategy	B strategy	Payoff for A and B
[1, 0, 0]	[1, 0, 0]	[0.5, 0.5]
[0, 0, 1]	[1, 0, 0]	[0.0, 1.0]
[1, 0, 0]	[0, 0, 1]	[1.0, 0.0]
[0, 0, 1]	[0, 0, 1]	[0.7, 0.7]

Experiment Result for Table 1

times	A strategy	B strategy	actual payoff for a round	desired payoff
0	[0.50 0.50 0.50]	[0.50 0.51 0.50]	[0.61 0.58]	[1, *] and [*, 1]
100	[0.58 0.51 0.4 0]	[0.60 0.52 0.40]	[0.59 0.56]	[1, *] and [*, 1]
200	[0.66 0.51 0.31]	[0.69 0.53 0.31]	[0.57 0.55]	[1, *] and [*, 1]
300	[0.73 0.52 0.23]	[0.76 0.54 0.24]	[0.55 0.53]	[1, *] and [*, 1]
…	…	…	…	…
1000	[0.91 0.58 0.07]	[0.92 0.60 0.08]	[0.52 0.50]	[1, *] and [*, 1]
	Converges to NE	Converges to NE		

Table 2 – Mutated Prisoner's Dilemma (with NE underlined)

		Player B strategy	
		[1, 0, 0]	[0, 0, 1]
Player A strategy	[1, 0, 0]	[0.7, 0.7]	[0.0, 1.0]
	[0, 0, 1]	[1.0, 0.0]	<u>[0.5, 0.5]</u>

A strategy	B strategy	Payoff for A and B
[1, 0, 0]	[1, 0, 0]	[0.7, 0.7]
[0, 0, 1]	[1, 0, 0]	[1.0, 0.0]
[1, 0, 0]	[0, 0, 1]	[0.0, 1.0]
[0, 0, 1]	[0, 0, 1]	[0.5, 0.5]

Experiment Result for Table 2

times	A strategy	B strategy	actual payoff for a round	desired payoff
0	[0.50 0.50 0.50]	[0.50 0.51 0.50]	[0.60 0.61]	[1, *] and [*, 1]
100	[0.39 0.50 0.60]	[0.40 0.51 0.58]	[0.56 0.60]	[1, *] and [*, 1]
200	[0.30 0.50 0.69]	[0.31 0.52 0.66]	[0.55 0.58]	[1, *] and [*, 1]
300	[0.23 0.50 0.75]	[0.24 0.53 0.73]	[0.53 0.57]	[1, *] and [*, 1]
…	…	…	…	…
1000	[0.07 0.50 0.92]	[0.08 0.59 0.90]	[0.49 0.54]	[1, *] and [*, 1]
	Converges to NE	Converges to NE		

Table 3 – Mutated Prisoner's Dilemma (with NE underlined)

		Player B strategy	
		[1, 0, 0]	[0, 0, 1]
Player A strategy	[1, 0, 0]	[0.0, 1.0]	[0.5, 0.5]
	[0, 0, 1]	[<u>0.7, 0.7</u>]	[1.0, 0.0]

A strategy	B strategy	Payoff for A and B
[1, 0, 0]	[1, 0, 0]	[0.0, 1.0]
[0, 0, 1]	[1, 0, 0]	[0.7, 0.7]
[1, 0, 0]	[0, 0, 1]	[0.5, 0.5]
[0, 0, 1]	[0, 0, 1]	[1.0, 0.0]

Experiment Result for Table 3

times	A strategy	B strategy	actual payoff for a round	desired payoff
0	[0.50 0.50 0.50]	[0.50 0.51 0.50]	[0.60 0.61]	[1, *] and [*, 1]
100	[0.38 0.50 0.61]	[0.60 0.52 0.38]	[0.62 0.63]	[1, *] and [*, 1]
200	[0.29 0.50 0.70]	[0.69 0.53 0.30]	[0.64 0.64]	[1, *] and [*, 1]
300	[0.23 0.50 0.76]	[0.75 0.54 0.23]	[0.65 0.66]	[1, *] and [*, 1]
…	…	…	…	…
1000	[0.09 0.50 0.90]	[0.90 0.59 0.09]	[0.67 0.69]	[1, *] and [*, 1]
	Converges to NE	Converges to NE		

Table 4 – Mutated Prisoner's Dilemma (with NE underlined)

		Player B strategy	
		[1, 0, 0]	[0, 0, 1]
Player A strategy	[1, 0, 0]	[1.0, 0.0]	[0.7, 0.7]
	[0, 0, 1]	[0.5, 0.5]	[0.0, 1.0]

A strategy	B strategy	Payoff for A and B
[1, 0, 0]	[1, 0, 0]	[1.0, 0.0]
[0, 0, 1]	[1, 0, 0]	[0.5, 0.5]
[1, 0, 0]	[0, 0, 1]	[0.7, 0.7]
[0, 0, 1]	[0, 0, 1]	[0.0, 1.0]

Experiment Result for Table 4

times	A strategy	B strategy	actual payoff for a round	desired payoff
0	[0.50 0.50 0.50]	[0.50 0.50 0.50]	[0.60 0.61]	[1, *] and [*, 1]
100	[0.60 0.50 0.38]	[0.38 0.51 0.61]	[0.63 0.63]	[1, *] and [*, 1]
200	[0.68 0.50 0.29]	[0.30 0.51 0.69]	[0.65 0.65]	[1, *] and [*, 1]
300	[0.75 0.51 0.23]	[0.24 0.52 0.75]	[0.66 0.66]	[1, *] and [*, 1]
…	…	…	…	…
1000	[0.89 0.53 0.09]	[0.09 0.54 0.90]	[0.68 0.69]	[1, *] and [*, 1]
	Converges to NE	Converges to NE		

Table 5 – Pizza War (with NE underlined)

		Player B strategy	
		[1, 0, 0]	[0, 0, 1]
Player A strategy	[1, 0, 0]	[0.6, 0.6]	[0.3, 0.7]
	[0, 0, 1]	[0.7, 0.3]	<u>[0.5, 0.5]</u>

A strategy	B strategy	Payoff for A and B
[1, 0, 0]	[1, 0, 0]	[0.6, 0.6]
[0, 0, 1]	[1, 0, 0]	[0.7, 0.3]
[1, 0, 0]	[0, 0, 1]	[0.3, 0.7]
[0, 0, 1]	[0, 0, 1]	[0.5, 0.5]

Experiment Result for Table 5

times	A strategy	B strategy	actual payoff for a round	desired payoff
0	[0.50 0.51 0.50]	[0.49 0.50 0.50]	[0.51 0.51]	[1, *] and [*, 1]
100	[0.47 0.50 0.52]	[0.47 0.51 0.52]	[0.51 0.50]	[1, *] and [*, 1]
200	[0.44 0.50 0.54]	[0.44 0.52 0.54]	[0.51 0.50]	[1, *] and [*, 1]
300	[0.41 0.50 0.57]	[0.41 0.53 0.56]	[0.50 0.50]	[1, *] and [*, 1]
…	…	…	…	…
1000	[0.26 0.51 0.68]	[0.26 0.58 0.67]	[0.49 0.49]	[1, *] and [*, 1]
	Converges to NE	Converges to NE		

Table 6 – Mutated Pizza War (with NE underlined)

		Player B strategy	
		[1, 0, 0]	[0, 0, 1]
Player A strategy	[1, 0, 0]	[0.7, 0.3]	<u>[0.6, 0.6]</u>
	[0, 0, 1]	[0.5, 0.5]	[0.3, 0.7]

A strategy	B strategy	Payoff for A and B
[1, 0, 0]	[1, 0, 0]	[0.7, 0.3]
[0, 0, 1]	[1, 0, 0]	[0.5, 0.5]
[1, 0, 0]	[0, 0, 1]	[0.6, 0.6]
[0, 0, 1]	[0, 0, 1]	[0.3, 0.7]

Experiment Result for Table 6

times	A strategy	B strategy	actual payoff for a round	desired payoff
0	[0.50 0.50 0.50]	[0.50 0.50 0.50]	[0.53 0.54]	[1, *] and [*, 1]
100	[0.53 0.50 0.45]	[0.46 0.51 0.54]	[0.54 0.55]	[1, *] and [*, 1]
200	[0.56 0.50 0.41]	[0.42 0.52 0.57]	[0.54 0.55]	[1, *] and [*, 1]
300	[0.60 0.51 0.37]	[0.38 0.52 0.61]	[0.55 0.55]	[1, *] and [*, 1]
…	…	…	…	…
1000	[0.74 0.52 0.20]	[0.22 0.55 0.76]	[0.56 0.57]	[1, *] and [*, 1]
	Converges to NE	Converges to NE		

Table 7 – Mutated Pizza War (with NE underlined)

		Player B strategy	
		[1, 0, 0]	[0, 0, 1]
Player A strategy	[1, 0, 0]	<u>[0.5, 0.5]</u>	[0.7, 0.3]
	[0, 0, 1]	[0.3, 0.7]	[0.6, 0.6]

A strategy	B strategy	Payoff for A and B
[1, 0, 0]	[1, 0, 0]	[0.5, 0.5]
[0, 0, 1]	[1, 0, 0]	[0.3, 0.7]
[1, 0, 0]	[0, 0, 1]	[0.7, 0.3]
[0, 0, 1]	[0, 0, 1]	[0.6, 0.6]

Experiment Result for Table 7

times	A strategy	B strategy	actual payoff for a round	desired payoff
0	[0.50 0.50 0.50]	[0.50 0.50 0.50]	[0.50 0.51]	[1, *] and [*, 1]
100	[0.52 0.50 0.47]	[0.52 0.51 0.47]	[0.49 0.51]	[1, *] and [*, 1]
200	[0.54 0.51 0.44]	[0.55 0.52 0.44]	[0.49 0.51]	[1, *] and [*, 1]
300	[0.56 0.51 0.40]	[0.57 0.53 0.42]	[0.50 0.57]	[1, *] and [*, 1]
…	…	…	…	…
1000	[0.66 0.53 0.25]	[0.71 0.57 0.28]	[0.49 0.56]	[1, *] and [*, 1]
	Converges to NE	Converges to NE		

Table 8 – Mutated Pizza War (with NE underlined)

		Player B strategy	
		[1, 0, 0]	[0, 0, 1]
Player A strategy	[1, 0, 0]	[0.3, 0.7]	[0.5, 0.5]
	[0, 0, 1]	<u>[0.6, 0.6]</u>	[0.7, 0.3]

A strategy	B strategy	Payoff for A and B
[1, 0, 0]	[1, 0, 0]	[0.3, 0.7]
[0, 0, 1]	[1, 0, 0]	[0.6, 0.6]
[1, 0, 0]	[0, 0, 1]	[0.5, 0.5]
[0, 0, 1]	[0, 0, 1]	[0.7, 0.3]

Experiment Result for Table 8

times	A strategy	B strategy	actual payoff for a round	desired payoff
0	[0.50 0.50 0.50]	[0.50 0.51 0.50]	[0.53 0.53]	[1, *] and [*, 1]
100	[0.46 0.50 0.54]	[0.53 0.51 0.46]	[0.53 0.54]	[1, *] and [*, 1]
200	[0.41 0.50 0.57]	[0.57 0.52 0.42]	[0.53 0.54]	[1, *] and [*, 1]
300	[0.38 0.50 0.60]	[0.60 0.53 0.38]	[0.54 0.54]	[1, *] and [*, 1]
…	…	…	…	…
1000	[0.21 0.51 0.75]	[0.75 0.58 0.22]	[0.55 0.56]	[1, *] and [*, 1]
	Converges to NE	Converges to NE		

Table 9 – Mutated War Extended (with NE underlined)

		Player B strategy		
		[1, 0, 0]	[0, 1, 0]	[0, 0, 1]
Player A strategy	[1, 0, 0]	[0.6, 0.6]	[0.3, 0.7]	[0.3, 0,3]
	[0, 1, 0]	[0.7, 0.3]	<u>[0.5, 0.5]</u>	[0.3, 0.3]
	[0, 0, 1]	[0.3, 0.3]	[0.3, 0.3]	[0.2, 0.2]

A strategy	B strategy	Payoff for A and B
[1, 0, 0]	[1, 0, 0]	[0.6, 0.6]
[0, 1, 0]	[1, 0, 0]	[0.7, 0.3]
[0, 0, 1]	[1, 0, 0]	[0.3, 0.3]
[1, 0, 0]	[0, 1, 0]	[0.3, 0.7]
[0, 1, 0]	[0, 1, 0]	[0.5, 0.5]
[0, 0, 1]	[0, 1, 0]	[0.3, 0.3]
[1, 0, 0]	[0, 0, 1]	[0.3, 0.3]
[0, 1, 0]	[0, 0, 1]	[0.3, 0.3]
[0, 0, 1]	[0, 0, 1]	[0.2, 0.2]

Experiment Result for Table 9

times	A strategy	B strategy	actual payoff for a round	desired payoff
0	[0.33 0.33 0.34]	[0.34 0.34 0.34]	[0.47 0.31]	[1, *] and [*, 1]
100	[0.33 0.36 0.29]	[0.34 0.38 0.29]	[0.49 0.33]	[1, *] and [*, 1]
200	[0.34 0.40 0.26]	[0.35 0.43 0.26]	[0.50 0.35]	[1, *] and [*, 1]
300	[0.34 0.43 0.23]	[0.36 0.48 0.22]	[0.52 0.36]	[1, *] and [*, 1]
…	…	…	…	…
1000	[0.38 0.64 0.13]	[0.40 0.73 0.12]	[0.58 0.43]	[1, *] and [*, 1]
	Converges to NE	Converges to NE		

Table 10 – Mutated Pizza War Extended (with NE underlined)

		Player B strategy		
		[1, 0, 0]	[0, 1, 0]	[0, 0, 1]
Player A strategy	[1, 0, 0]	[0.7, 0.3]	<u>[0.6, 0.6]</u>	[0.3, 0,7]
	[0, 1, 0]	[0.3, 0.3]	[0.5, 0.5]	[0.7, 0.3]
	[0, 0, 1]	[0.3, 0.3]	[0.2, 0.7]	[0.8, 0.3]

A strategy	B strategy	Payoff for A and B
[1, 0, 0]	[1, 0, 0]	[0.7, 0.3]
[0, 1, 0]	[1, 0, 0]	[0.3, 0.3]
[0, 0, 1]	[1, 0, 0]	[0.3, 0.3]
[1, 0, 0]	[0, 1, 0]	[0.6, 0.6]
[0, 1, 0]	[0, 1, 0]	[0.5, 0.5]
[0, 0, 1]	[0, 1, 0]	[0.2, 0.7]
[1, 0, 0]	[0, 0, 1]	[0.3, 0.7]
[0, 1, 0]	[0, 0, 1]	[0.7, 0.3]
[0, 0, 1]	[0, 0, 1]	[0.8, 0.3]

Experiment Result for Table 10

times	A strategy	B strategy	actual payoff for a round	desired payoff
0	[0.49 0.49 0.50]	[0.50 0.51 0.51]	[0.37 0.52]	[1, *] and [*, 1]
100	[0.52 0.49 0.47]	[0.45 0.56 0.51]	[0.38 0.54]	[1, *] and [*, 1]
200	[0.54 0.49 0.44]	[0.40 0.61 0.51]	[0.38 0.56]	[1, *] and [*, 1]
300	[0.57 0.49 0.41]	[0.36 0.65 0.50]	[0.39 0.58]	[1, *] and [*, 1]
…	…	…	…	…
1000	[0.70 0.48 0.26]	[0.21 0.79 0.50]	[0.41 0.65]	[1, *] and [*, 1]
	Converges to NE	Converges to NE		

Table 11 – Mutated Pizza War Extended (with NE underlined)

		Player B strategy		
		[1, 0, 0]	[0, 1, 0]	[0, 0, 1]
Player A strategy	[1, 0, 0]	[<u>0.7</u>, 0.3]	[<u>0.6</u>, 0.0]	[0.3, <u>0.7</u>]
	[0, 1, 0]	[0.3, 0.3]	[0.5, <u>0.5</u>]	[0.7, 0.3]
	[0, 0, 1]	[0.3, 0.3]	[0.2, 0.7]	<u>[0.8, 0.9]</u>

A strategy	B strategy	Payoff for A and B
[1, 0, 0]	[1, 0, 0]	[0.7, 0.3]
[0, 1, 0]	[1, 0, 0]	[0.3, 0.3]
[0, 0, 1]	[1, 0, 0]	[0.3, 0.3]
[1, 0, 0]	[0, 1, 0]	[0.6, 0.0]
[0, 1, 0]	[0, 1, 0]	[0.5, 0.5]
[0, 0, 1]	[0, 1, 0]	[0.2, 0.7]
[1, 0, 0]	[0, 0, 1]	[0.3, 0.7]
[0, 1, 0]	[0, 0, 1]	[0.7, 0.3]
[0, 0, 1]	[0, 0, 1]	[0.8, 0.9]

Experiment Result for Table 11

times	A strategy	B strategy	actual payoff for a round	desired payoff
0	[0.49 0.49 0.50]	[0.50 0.51 0.51]	[0.39 0.49]	[1, *] and [*, 1]
100	[0.51 0.49 0.47]	[0.44 0.50 0.58]	[0.41 0.51]	[1, *] and [*, 1]
200	[0.54 0.49 0.44]	[0.38 0.49 0.65]	[0.43 0.53]	[1, *] and [*, 1]
300	[0.56 0.49 0.42]	[0.32 0.49 0.70]	[0.44 0.55]	[1, *] and [*, 1]
…	…	…	…	…
1000	[0.66 0.48 0.28]	[0.16 0.44 0.86]	[0.50 0.58]	[1, *] and [*, 1]
	Not Converging to NE	Converges to NE		

*****This is an example of failure at epochs of 1000.**

However, the machine converging to NE at epochs of 10000.

The reason for this failure is that, **the trajectory of player A's strategies and the trajectory of player B's strategies intersect at two points (with one at [0.8, 0.9], and the other near [0.3, 0.7]),** which renders the

machine confused at lower epochs of training iteration and stuck at [0.3, 0,7]. And it is a valuable example of failure. For a vivid explanation, imagine these two ants A and B climbs in two more rugged mountain sides (since the epochs in the training or learning phase is rather low, and the mountain side is more rugged and less smooth as a result), and they get stuck in local maximum more easily***

Table 12 – Rock Paper Scissor (No Discrete NE Exists)

		Player B strategy		
		[1, 0, 0]	[0, 1, 0]	[0, 0, 1]
Player A strategy	[1, 0, 0]	[0, 0]	[0, 1]	[1, 0]
	[0, 1, 0]	[1, 0]	[0, 0]	[0, 1]
	[0, 0, 1]	[0, 1]	[1, 0]	[0, 0]

A strategy	B strategy	Payoff for A and B
[1, 0, 0]	[1, 0, 0]	[0, 0]
[0, 1, 0]	[1, 0, 0]	[1, 0]
[0, 0, 1]	[1, 0, 0]	[0, 1]
[1, 0, 0]	[0, 1, 0]	[0, 1]
[0, 1, 0]	[0, 1, 0]	[0, 0]
[0, 0, 1]	[0, 1, 0]	[1, 0]
[1, 0, 0]	[0, 0, 1]	[1, 0]
[0, 1, 0]	[0, 0, 1]	[0, 1]
[0, 0, 1]	[0, 0, 1]	[0, 0]

Experiment Result for Table 12

times	A strategy	B strategy	actual payoff for a round	desired payoff
0	[0.33 0.33 0.34]	[0.34 0.34 0.34]	[0.58 0.14]	[1, *] and [*, 1]
100	[0.33 0.33 0.33]	[0.33 0.34 0.34]	[0.58 0.14]	[1, *] and [*, 1]
200	[0.34 0.33 0.32]	[0.33 0.35 0.34]	[0.58 0.14]	[1, *] and [*, 1]
300	[0.35 0.33 0.31]	[0.32 0.35 0.35]	[0.58 0.14]	[1, *] and [*, 1]
…	…	…	…	…
1000	[0.38 0.33 0.27]	[0.29 0.37 0.36]	[0.58 0.14]	[1, *] and [*, 1]
	Less Divergent due to no discrete NE exists	Less Divergent due to no discrete NE exists		