Neural Networks and the Evolution of Cooperation

Lenka Pitonakova
contact@lenkaspace.net
University of Sussex 2011

Abstract

The paper investigates artificial evolution of cooperation in the Iterated Prisoner's Dilemma using a number of player implementations. Existing strategy encoding and neural network models are compared with an action-discriminating neural network created during writing of this paper. Evaluation is performed in terms of number of generations needed for reaching a desired cooperation level as well as the nature of evolved strategies. Examples when the action-discriminating model evolved the most beneficent strategies are given.

Keywords

Iterated Prisoner's Dilemma, evolution of cooperation, strategies, interaction, neural networks

1. Introduction

In the world governed by survival instincts an individual always tries to be the fittest possible by exploiting the environment and the others around. Despite of that some species evolved what we call altruism, a tendency of an individual to cooperate with its group members in order to maximise their collective fitness (Axelrod 1984). Scientists have always wondered about what drove such evolution and what makes selfish individuals sacrifice some of their reward for common long-term benefits.

Game theory models behaviour of agents in attempt to explain dynamics of their interactions. One of its model games is Prisoner's Dilemma (PD) where two or more players engage and can either cooperate with or defect the others. Rewards based on all players' actions are given after all have acted. In the 2-player PD, defection means getting a higher or the same reward as the opponent. Mutual cooperation leads to a mean reward for both, although cooperating when the other defects results in getting the lowest score. The standard payoff matrix (Axelrod 1984, Axelrod 1987, Smucker et al. 1994, Lomborg 1996) is shown in Table 1.

C , D	D , C	C , C	D , D
L=0, H=5	H=5, L=0	C=3, C=3	D=1, D=1

Table 1: payoffs received in the Prisoner's Dilemma

Iterated Prisoner's Dilemma (IPD) is PD played a number of times between the same players and the rewards are given after each round. Iterations allow for the individuals to keep track of the opponent's moves and use strategies to maximise their own average payoff. The payoff matrix shown in Table 1 is designed to satisfy the following rules:

$$L<D<C<H \quad (1)$$

$$(H+L)/2 < C \quad (2)$$

The first rule implies that defecting while other cooperates leads to the highest score H=5 for the defector and the lowest L=0 for the cooperator. Mutual defection means the second lowest score D=1 and mutual cooperation leads to the score C=3. The second rule assures that the average payoff is higher when the players mutually cooperate (i.e. demonstrate altruistic behaviour) than when they alter cooperate-defect, defect-cooperate actions (Smucker et al. 1994).

Apart from getting a long-term reward for mutual cooperation, IPD players can punish defection by defecting on the next move. Experiments with humans performed by Rapport (1966) proved that it is always the most beneficial to defect on the last move since there will be no punishment from the opponent in the future. Rapport follows this logic and states that it would be rational to defect on (N-1)th, (N-2)th,..(N-N)th moves. For mutual cooperation to emerge, players must be boundedly rational, i.e. they cannot know with 100% certainty when the game will end or what action the opponent will take.

It is important to understand that there is no single strategy that can outperform all others. IPD strategies work and score differently in different social environments. A highly defective behaviour will perform poorly amongst other defective ones but will win against cooperative players. Similarly, a minority of cooperative players will score below the average, although enough of such players can eventually manage to get the optimal average payoffs as long as defectors are punished well enough. One of the most robust strategies is Tit-For-Tat (TFT) where a player cooperates on the first move and then does whatever the opponent did on the previous move (Axelrod 1987). Wu and Axelrod (1995) provide some other quasi-robust strategies.

The IPD model simplifies real-world interaction not only between humans but other species as well (Axelrod 1984). It is believed that analysing artificial social networks can provide insights into dynamics and evolution of social networks found in the nature (Smucker 1994). This paper will comment on the strategy-encoding approach for modelling IPD agents developed by Axelrod (1987) as well as the finite-state-machine model by Ashlock (1994) and investigate the former by using a genetic algorithm. Particularly, the nature of evolved strategies will be examined. The second half of the paper will concentrate on neural network based models and compare them with the traditional ones. The questions that this research will try to answer are whether cooperation can be evolved amongst players with neural networks and whether such models can perform at least as well as the strategy-encoding one.

2. Traditional Agent Models

Axelrod (1987) pioneered the evolution of cooperation in IPD by creating a model where agents looked at their own and the opponent's three previous moves and took an action encoded in their genotype. Each gene had a value of 1 (cooperate) or 0 (defect). There are 4 possible combinations (CD, DC, CC, DD) of actions history in each time step, thus 4x4x4=64 genes were needed to encode actions for each possible history of 3 steps back. Additional 6 genes were added to give history of previous three hypothetical moves so that 1^{st} , 2^{nd} and 3^{rd} action could be looked up in the genotype. Axelrod used a genetic algorithm where crossover and low mutation produced an offspring from two parents. The parents were selected so that average individuals got one mating and individuals that were one standard deviation more effective got two. Each IPD game consisted of 151 rounds.

Axelrod conducted 40 evolutions that lasted for 30 generations and provided a list of evolved strategies that he claimed mirrored TFT: cooperate after three mutual cooperations, defect after three mutual defections, defect when the other player defects after cooperating, cooperate when mutual cooperation has been restored. These results were certainly remarkable in the 1970s when the computational power was low, although it is arguable how statistically precise they are since only 40 experiments were conducted. Nevertheless, Axelrod's work demonstrated the power of the genetic algorithm and provided a starting point for further experiments.

Another model developed by Ashlock et. al. (1994) was based on finite state machines. It was shown in their first paper that the evolution of cooperation progressed much quicker when agents were able to remember an expected payoff for each opponent and offer or refuse games based on this value. In their second paper (Smucker et al. 1994), they discussed behavioural patterns based on IPD with choice and refusal (IPD/CR) and developed a Significant Play Graph that showed 'friendships' between agents.

3. The Strategy-Encoding Model

The first model implemented for the purposes of this paper mirrored the one of Axelrod in the genotype-phenotype encoding, since it focused directly on evolution of strategies. TFT-like strategies should theoretically only need to evaluate the previous time step, thus the memory length was relaxed to two. The genotype had length 4x4 + 4 = 20.

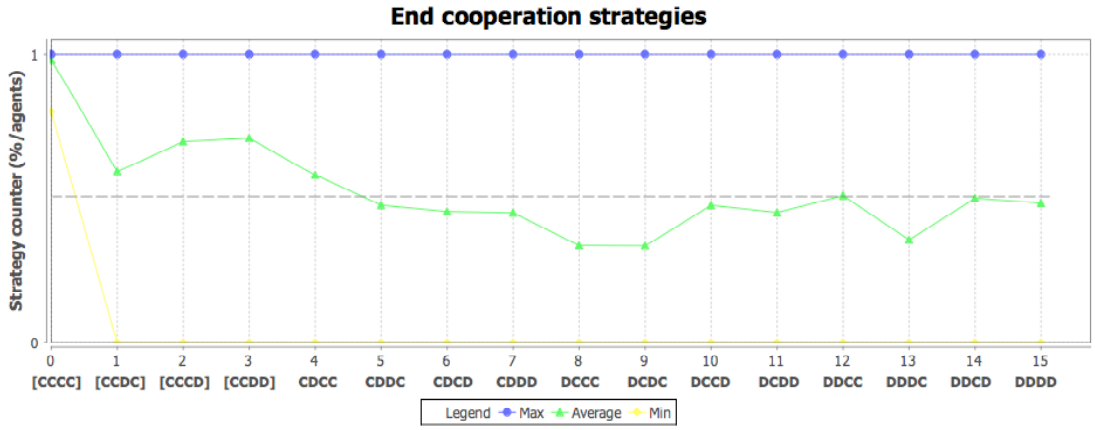
A deviation of the microbial genetic algorithm (MGA) was used in order to provide good enough diversion in the population and a suitable evolution speed. A standard MGA invented by Harvey (1996) selects two agents each round, compares their payoffs and transfers genes from a winner to a loser with a constant recombination probability RP for each gene. Genes of the loser are then mutated with a mutation probability MP. The GA used for this paper had a number of parents PAR at the end of each generation and PAR winner-loser comparisons were made after a number of IPD tournaments. Setting PAR to more than 1 effectively speeded up the evolutionary process. It is important to note that the winner's genes were not passed but the loser's genes were still mutated when both had the same fitness. This rule was added so that fit genotypes did not get lost in the new generation but the mutation was uniformly applied to all agents competing for parenthood.

The experiments were performed 1000 times each with a population of 10 (POP=10). Increasing the population size led to very long evolutions and sometimes made evolving cooperation impossible. An evolution ended when 90% cooperations out of all actions taken in a generation were recorded. 5 IPD tournaments with unique pairs were played during each generation, meaning that everybody played everybody and each evolved strategy had the same chance in the generation. To maximise the speed of the evolution and satisfy the condition of bounded rationality, the IPD tournaments had 200-210 PD rounds and the exact number was random for each tournament. The optimal mutation probability was found to be 0.1, meaning that an average of 10% of genes mutated each mating. The number of parents (PAR) was set to 20.

The genotype-phenotype encoding was tested by observing memory contents, genotypes and actions taken by agents.

3.1. Evolved strategies

Picture 1 shows maximum, average and minimum percentages of agents that would have cooperated if they recorded a particular moves history at the end of each evolution. The histories are denoted as following: opponent's action at t-1, own action at t-1, opponent's action at t-2 and own action at t-2, giving 4 letters for each possible history. Averages in the graph are based on 1000 evolutions. Correctness of the values was confirmed by running a number of single evolutions and manually counting the number of cooperations for histories that occurred in them.



Picture 1: Maximum, average and minimum representation percentages of cooperative strategies with POP=10, PAR=20. Each dot on the graph represents an evolved cooperation strategy and its value equals to a percentage of agents that would cooperate given a history pattern. See further explanation in text.

The minimum and maximum found percentages shown on Pictures 1 are always 0 and 1 respectively. This means that there was always at least one occasion when a strategy did not evolve at all or everybody evolved it. This result seemed to be a side effect of the low population number. The minimum and maximum values resembled the averages more closely in greater populations.

The strategies recorded after individual evolutions were often different than the shown averages. Cooperation after two mutual cooperations always evolved strongly, although so did cooperation after two mutual defections in some cases (examples are given in Appendix A). However, averages over 1000 generations provided satisfying results and should a final set up be taken from these experiments, it would have to be an average of the genotypes.

The average evolved strategies were very likely to cooperate when the opponent previously cooperated and most likely to do so when the opponent cooperated in both previous time steps. The defection was most likely if the opponent defected during one or two attempts to cooperate.

The presented results resemble Axelrod's, although it is not possible to precisely compare the numbers due to different memory lengths and the small amount of experiments conducted by Axelrod. The evolved agents cooperated after 2 mutual cooperations (CCCC) or less often after cooperation has been restored (CCDC, CCCD, CCDD). Also, they often defected after one or two attempts were made to cooperate while the other defected (DCCC, DCDC). Defection after two mutual defections did not evolve as strongly as Axelrod reported.

4. Existing Artificial Neural Network Models

There were several attempts to solve the IPD problem using neural networks. Most of them did not use evolution but network training and compared effectivity of the learned behaviours with known fixed strategies. (Sandholm and Crites 1995, O'Madadhain 2004). These approaches seemed to deliver good solutions, although they did not address evolution dynamics of neural network models.

Research about evolution of neural networks in IPD was performed by Harrald and Fogel (1996). The network used memory with length of 3 and the actions were encoded as continuous values between -1 and 1 where 1 meant complete cooperation. 6 input nodes represented 3 previous steps of the player and the opponent. There was a hidden layer of N fully connected nodes and an output node that produced values from the range <-1;1>. The payoffs were calculated according to the formula

$$\text{Payoff} = -0.75\alpha + 1.75\beta + 2.25 \quad (3)$$

where α was an output of own and β of the opponent's network. The equation satisfied the constraints of a traditional IPD model given in formulas (1) and (2).

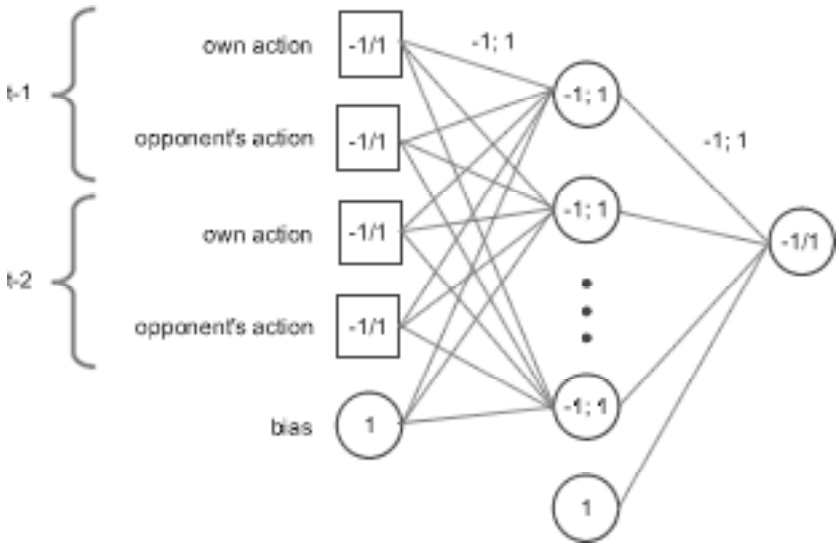
Fogel found that the most effective network had topology 6-20-1 (6 inputs, 20 hidden layer nodes, 1 output) and that at least a population of 20 needed to be used to achieve a reasonable cooperation level where 80% of 20 test cases generated cooperative behaviour after 10th generation.

4.1. Fogel-inspired Model

A Fogel-inspired artificial neural network (ANN) was implemented with two differences: a) the input and output values were discrete (-1 for defection or 1 for cooperation), thus the reward points were given according to the standard table used for the strategy-encoding model (Table 1), b) Population of 10 with PAR=20 was used. Both changes were made to achieve a closer match with the strategy-encoding model. Picture 3 shows architecture of the ANN.

The same genetic algorithm as for the strategy-encoding model was used. Each loser's gene was mutated by a small random number from a Gaussian distribution with variance VAR=0.1 and mean in the gene value, since evolution had to deal with real numbers (Harrald and Fogel 1996). Values of weights and nodes were limited to the range <-1; 1>. Correct input recording, genotype-phenotype encoding as well as resulting actions were tested by stepping through the program, observing and manually calculating nodes values for each of the tested network topologies. Table 2 summarizes how different topologies denoted by using Harrald and Fogel's method performed.

The number of generations in each evolution was higher that Fogel reported most probably because a) the network only remembered two time steps back, b) the evolution did not finish until the 90% cooperation target was reached and c) the population was lower*. It is rather difficult to compare the quality of evolved cooperation with Harrald and Fogel since they do not provide sufficient description of the 'cooperative behaviour' that they evolved. However, the average fitness recorded throughout evolutions was similar to one they reported - it neared the average payoff 3.



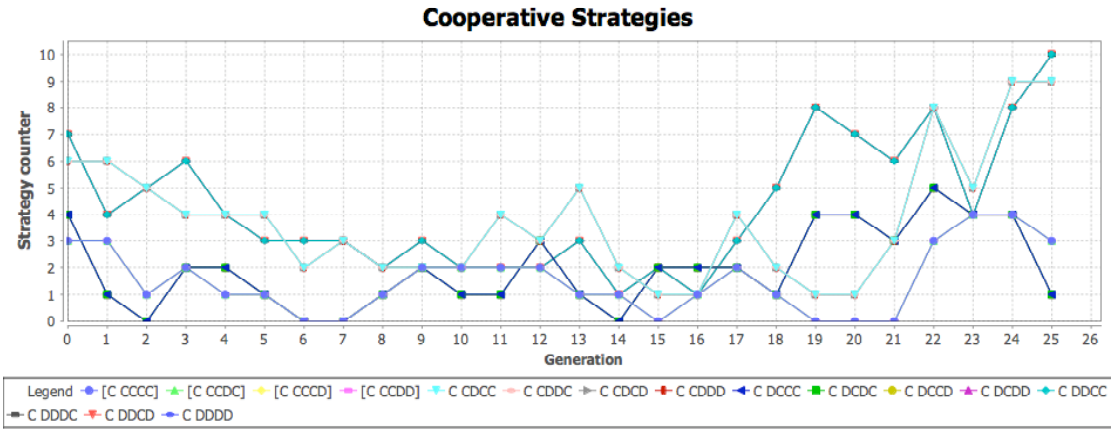
Picture 3: topology of the Fogel-inspired neural network

* *Experiments with population 20 and PAR=40 with the 4-8-1 network showed faster evolution*

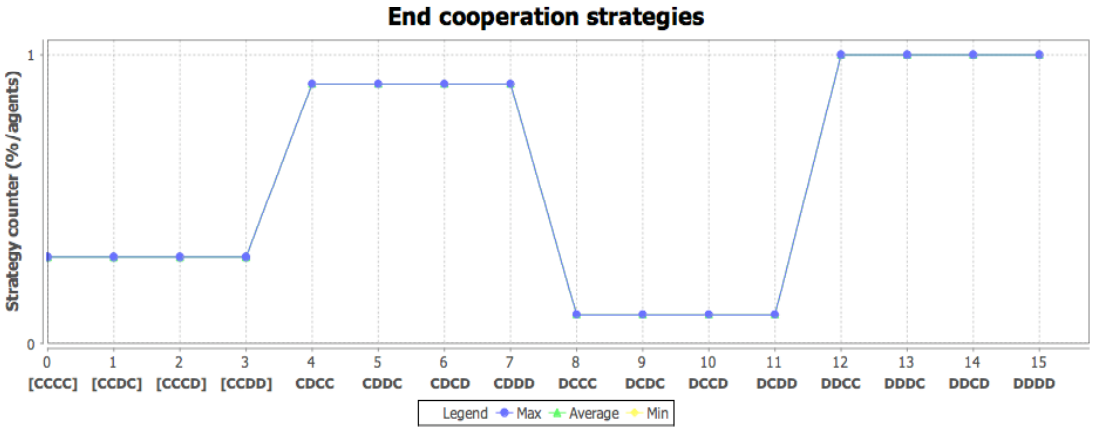
No.	Setup	Generations	Average fitness
1	4-2-1	1057; 138.569; 3	3.000; 2.965; 2.802
2	4-8-1	792; 98.445; 2	3.000; 2.960; 2.801
3	4-20-1	560; 79.510; 1	3.000; 2.960; 2.800
4	4-8-8-1	1227; 170.002; 3	3.000; 2.962; 2.801

Table 2: performance of different ANNs with the Fogel-inspired actions encoding. The ANNs are denoted by a number of nodes in each layer, starting with the input and ending with the output layer. The given values are presented as maximum found; average; minimum found.

Even thought cooperation did eventually evolve, there are a number of problems with the Fogel-inspired approach. Firstly, Harrald and Fogel themselves reported that
behaviour always declined towards defection when the evolution continued and that a recovery was very rare. Secondly, defection and cooperation for one actor at a certain time step shared the same node and consequently the same weight. This resulted in groups of strategies evolving together. Picture 4 shows evolution of cooperative strategies in an exemplary case with the 4-8-1 ANN. There are clearly four groups where strategies have the same values throughout the evolution. Picture 5 shows how many agents used each cooperative strategy at the end of the exemplary evolution and confirms the connectivity of strategies that share the same first two letters. Other examples for the 4-8-1 setup are presented in Appendix B. Similar results were found with all network topologies that used the Fogel-inspired action representation, even when the population was higher.

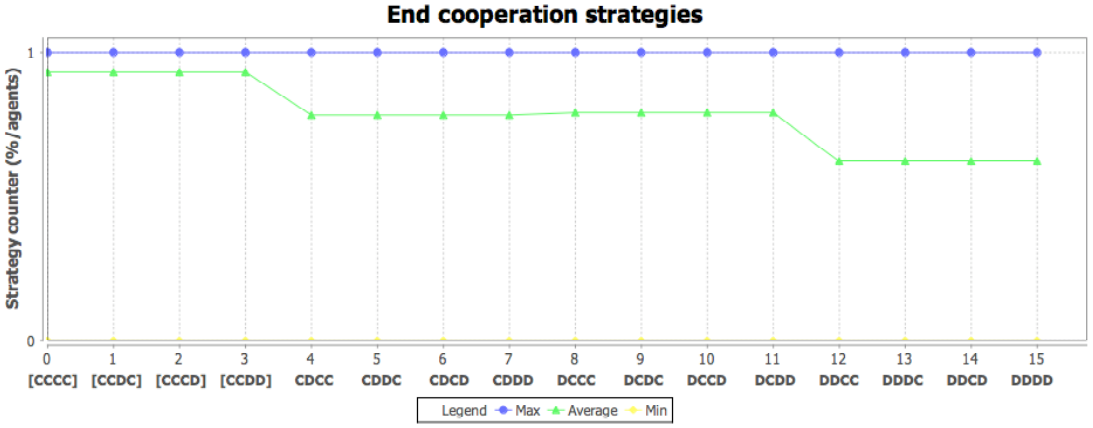


Picture 4: an example of evolution of cooperative strategies using the Fogel-inspired action representation with the 4-8-1 NN. Each graph line represents a possible history. The values show how many of 10 agents would cooperate given a specific history pattern at the end of each generation.



Picture 5: exemplary representation of evolved cooperative strategies after 25 generations using the Fogel-inspired action representation with the 4-8-1 NN. .

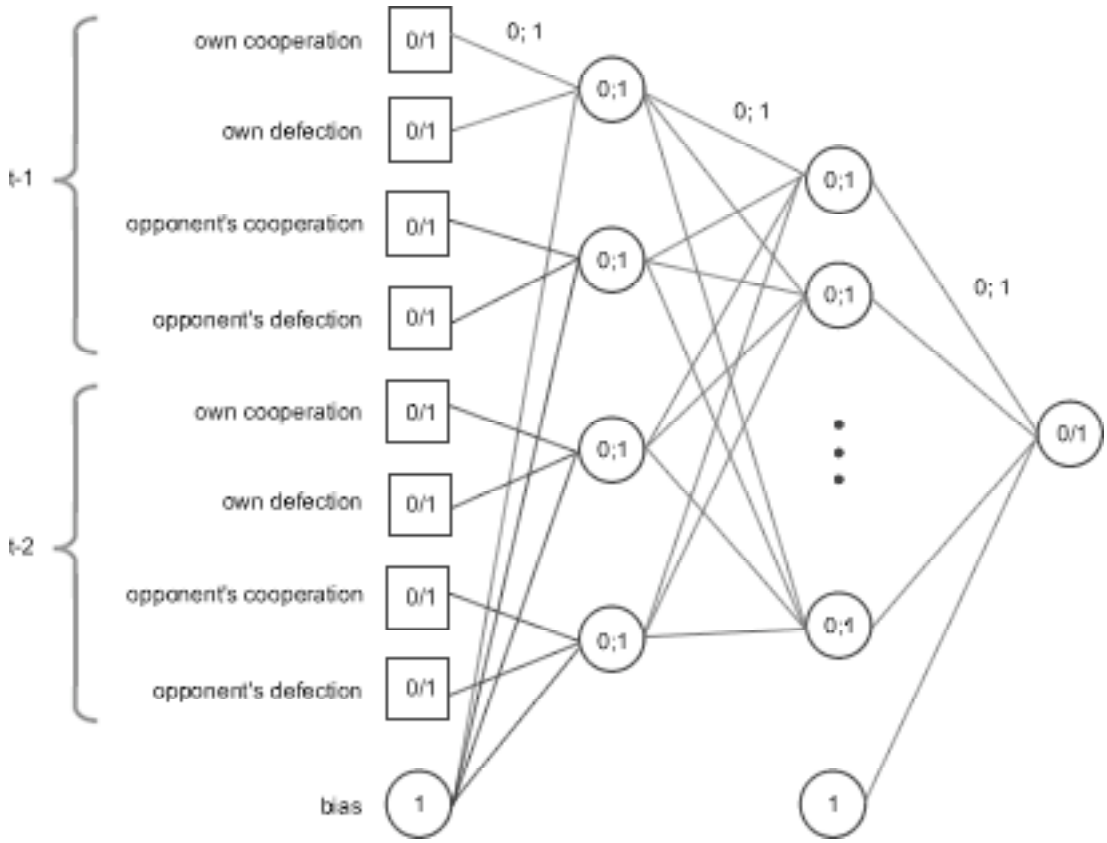
Picture 6 shows average, minimum and maximum values for each cooperative strategy based on 1000 evolutions. There is a lower tendency towards cooperating when the other defected on the previous turns. While this is a partially satisfying result, the problem with grouped strategies remains.



Picture 6: maximum, average and minimum representation percentagesof evolved cooperative strategies in the Fogel-inspired model with POP=10, PAR=20 based on 1000 evolutions

5. Action-Discriminating Artificial Neural Network

Action-discriminating artificial neural network (ADANN) was implemented in an attempt to solve the strategies grouping problem of the Fogel-inspired model. Two nodes were used for an opponent's move in (t-1), one for cooperation and one for defection, both of which had values either 0 or 1. The values of the nodes were set to 1 for cooperation, 0 for defection if the opponent's move was to cooperate and vice versa. Similarly, there were two nodes for own action in (t-1) and additional four nodes for actions in (t-2).



Picture 7: topology of the action-discriminating artificial neural network with 1 T-layer and an additional hidden layer. See explanation in the text.

Implementing separate nodes for the actions allowed for each to affect the network output separately. The evolution could therefore distinguish between their importance and strategies with one or more opposite letters (e.g. CCCC and CCCD) were not dependant on each other.

Initially, a network with one 'translation' layer (T-layer) was implemented. The layer had four nodes and its role was to evaluate the individual actions. Each node was only connected to two inputs of the same actor in the same time step. The layer was then fully connected to another hidden or output layer of the network. Picture 7 shows architecture of a network with a T-layer and an additional hidden layer. The output produced cooperation if its value was above 0.5. Output of all transfer functions and the node values were restricted to the range <0;1>. MV was kept as 0.1, RP as 1 and the population was set to 10 with PAR=20. Correctness of the node values and how they affected the output action was confirmed by the same method as with the Fogel-inspired networks.

The first network denoted 8-4*-1 for 8 inputs, 4 T-layer nodes and 1 output did not produce cooperation and evolved strategies could not be evaluated easily. Additional experiments were performed with networks that had no T-layer and a fully connected hidden layer (8-2-1, 8-4-1, 8-10-1, 8-20-1), 1 T-layer and 1 hidden layer (8-4*-2-1), as well as 2 hidden layers (8-4-4-1, 8-8-4-1, 8-8-8-1). Only a few of the networks produced reasonable cooperation levels between generations 100 and 400*. The best-performing network had topology 8-2*-1, where the two nodes of the T-layer connected actions of both actors in 1 time step (i.e. each T-layer node had 4 inputs).

**The upper limit 400 was selected because agents that used strategy encoding usually reached the 90% cooperation target after around 250-350 generations. The lower limit 100 was selected to give the evolutions enough time to reach a potential stability in terms of the strategies.*

5.1. The Cooperation Constraint

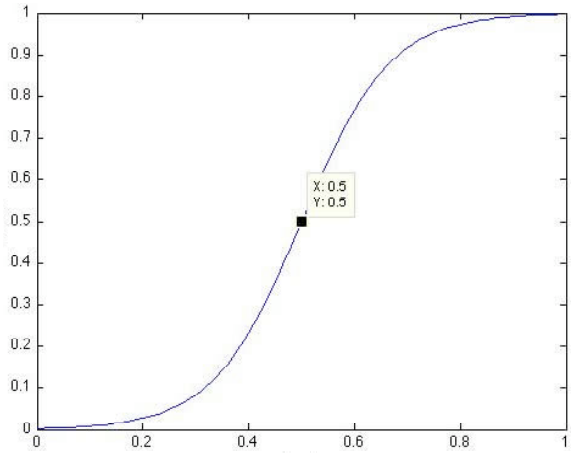
Cooperation was found to be very limited during the first experiments with the ADANN due to its restricted conditions. The output transfer function dictated that more than a half of the last hidden layer nodes had to produce values above 0.5 in order for the output to produce a value above 0.5. Metaphorically speaking, if the N last hidden layer nodes represented N different strategies then the cooperation would have to evolve for at least a half of them so that network cooperated in 1 particular situation. However, a particular situation should theoretically have been represented by only one of the strategies. A more realistic expectation was thus to cooperate if at least one of the nodes demanded cooperation. Therefore, the shape of the sigmoid transfer function of the output was adjusted based on the number of nodes in the last layer:

$$\text{SigmoidMiddle} = 1/\text{lastLayerSize} * 5/6 \quad (4)$$

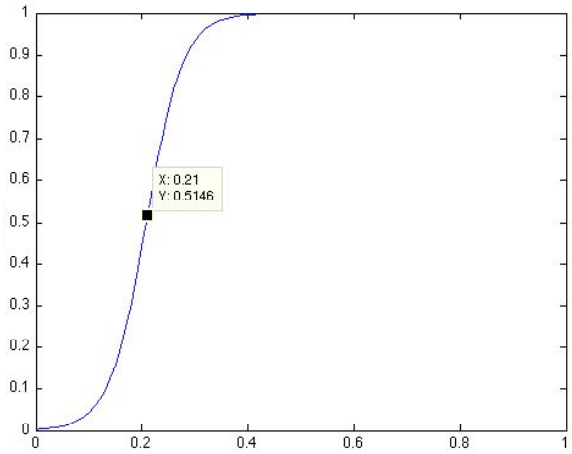
$$\text{Temp} = 12*0.5/\text{SigmoidMiddle}$$

$$\text{Out} = (1/ (1 + e^{(-(-6 + \text{temp}*\text{weightedSumOfInputs}))})) \quad (5)$$

The value of Out (output after activation) was above 0.5 if at least one node was very close to completely suggesting cooperation. Pictures 8 and 9 compare shapes of sigmoid functions with the middle in 0.5 and in the adjusted SigmoidMiddle for 4 previous-layer nodes respectively.



Picture 8: sigmoid transfer function with SigmoidMiddle=0.5



*Picture 9: sigmoid with SigmoidMiddle calculated based on equations (4) and (5). The output value is above 0.5 if at least 1 node of 4 has value 5/6, i.e. value before activation is 1/4 * 5/6 = 0.208.*

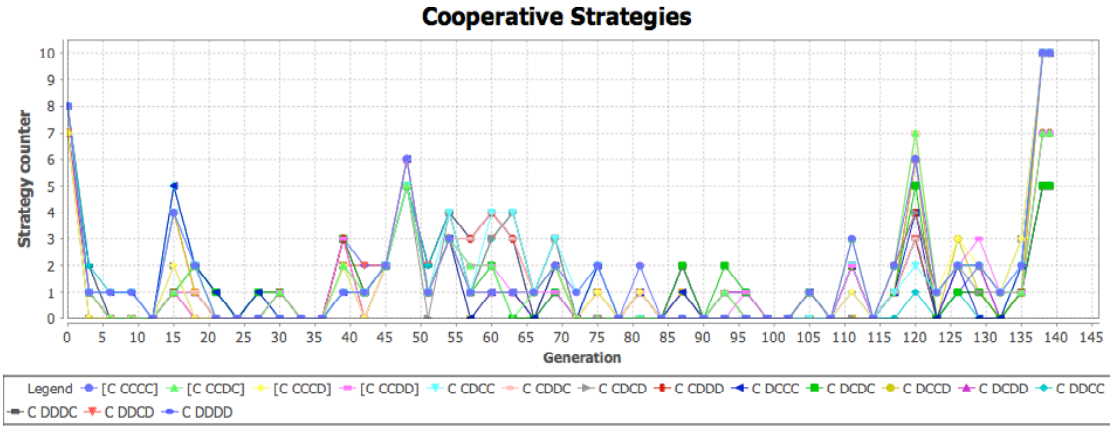
Networks with the new constraint produced more cooperation and it was possible to run 1000 evolutions to test what strategies evolved when using different topologies. The evolution characteristics are shown in Table 3.

No.	Setup	Generations	Average fitness
1	8-4*-1	3693; 431.144; 1	3.000; 2.957; 2.801
2	8-4-1	5138; 530.783; 1	3.000; 2.957; 2.822
3	8-8-1	2601; 7.785; 1	3.000; 2.993; 2.878
4	8-8-1 (SM2)	3512; 438.471; 1	3.000; 2.954; 2.801
5	8-20-1 (SM2)	1913; 216.176; 1	3.000; 2.954; 2.820

6	8-20-20-1	5233; 659.513; 1	3.000; 2.957; 2.802
7	8-4*-4-1	N/A - not all evolutions took less than MAX_GENERATIONS	
8	8-4*-8-1 (SM2)	N/A - not all evolutions took less than MAX_GENERATIONS	
9	8-4*-8-1	6592; 918.067; 1	3.000; 2.957; 2.820
10	8-4*-20-1	1618; 154.446; 1	3.000; 2.957; 2.801
12	8-4-8-1	6232; 740.663; 1	3.000; 2.957; 2.801
13	8-4-8-1 (SM2)	N/A - not all evolutions took less than MAX_GENERATIONS	
14	8-4-20-1	992; 88.729; 1	3.000; 2.955; 2.822
15	8-4-20-1 (SM2)	N/A - not all evolutions took less than MAX_GENERATIONS	

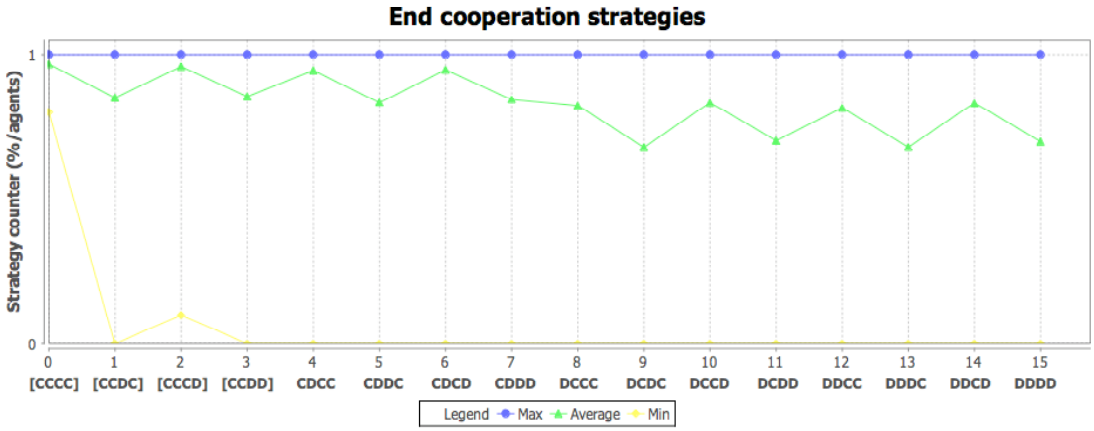
Table 3: number of generations and average fitness for evolutions with different ADANN topologies. The given values are presented as maximum found; average; minimum found. Networks denoted (SM2) used a higher cooperation constrain discussed in Section 5.2.

Similarly to the Fogel-inspired network, adding nodes to the last hidden layer decreased the number of generations needed to reach the 90% cooperations target (compare rows 2 and 3 in Table 3). The resulting strategies were more diverse than before, although they did not match the ones evolved with the strategy-encoding model. However, Picture 10 shows that they evolved separately unlike in the Fogel-inspired model. Evolutions with all ADANN topologies expressed this characteristic.



Picture 10: an example of evolution of cooperative strategies with the 8-4-20-1 ADANN*

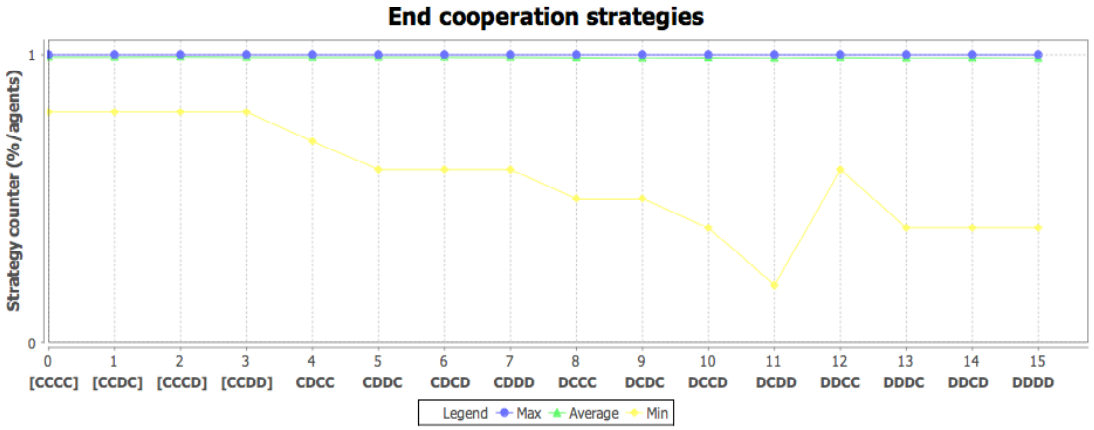
The steps visible in the graph of average evolved strategies (Picture 11) might imply that strategies were dependant on each other, although the example of a single evolution on Picture 10 does not confirm this assumption. While the strategies did evolve independently, the outcomes of evolutions progressively produced the shown average values. This effect was directly observed by looking at changes in the averages graph after each new evolution. The behaviour was significantly different from the one of the Fogel-inspired network where connections between strategies were obvious even when looking at individual evolutions (compare Appendices B and C).



Picture 11: representation percentages of evolved cooperation strategies based on 1000 evolutions with the 8-4-20-1 ADANN*

5.2. The Effect of Low Cooperation Constraint and Evolution Length

Relaxing the cooperation constraint significantly by adding many nodes to the hidden layer (the result in equation (4) was too low) or by directly changing equation (4) to produce lower values caused agents to cooperate very often (e.g. rows 3 and 14 in Table 3). Picture 12 shows that the average strategies evolved with the 8-8-1 network (Table 3, row 3) were almost at their maximum values, although the graph of minimum values (in yellow) vaguely resembled averages in the strategy-encoding model (compare pictures 1,2 and 12).

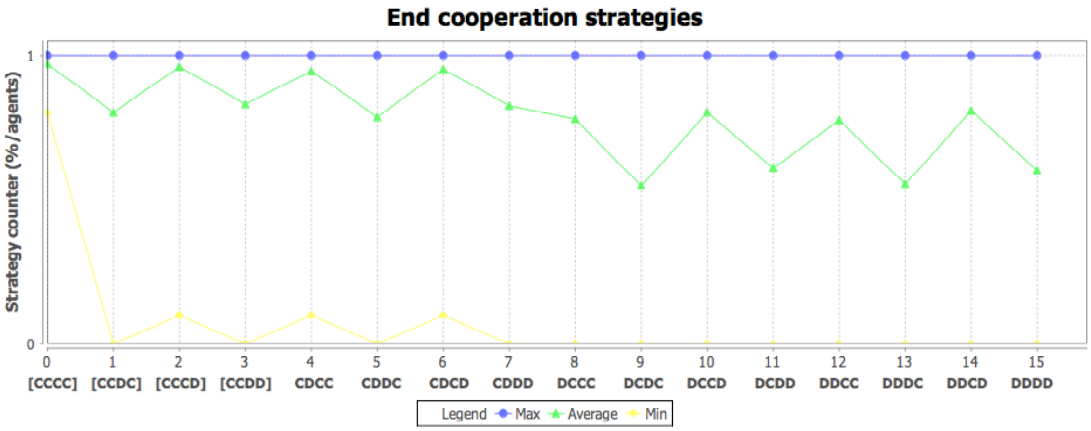


Picture 12: cooperative strategies representation after 1000 evolutions using the 8-8-1 ADANN.

Changing the equation (4) to

SigmoidMiddle2 = SM2 = 1/4 * 5/6 (6)

fixed this problem by constraining cooperation. The evolution was prolonged (compare rows 3 and 4 in Table 3) and the graph of average cooperation strategies changed (compare Pictures 12 and 13). However, cooperation became too constrained for some more complex hidden layers (Table 3, rows 8,13 and 15).



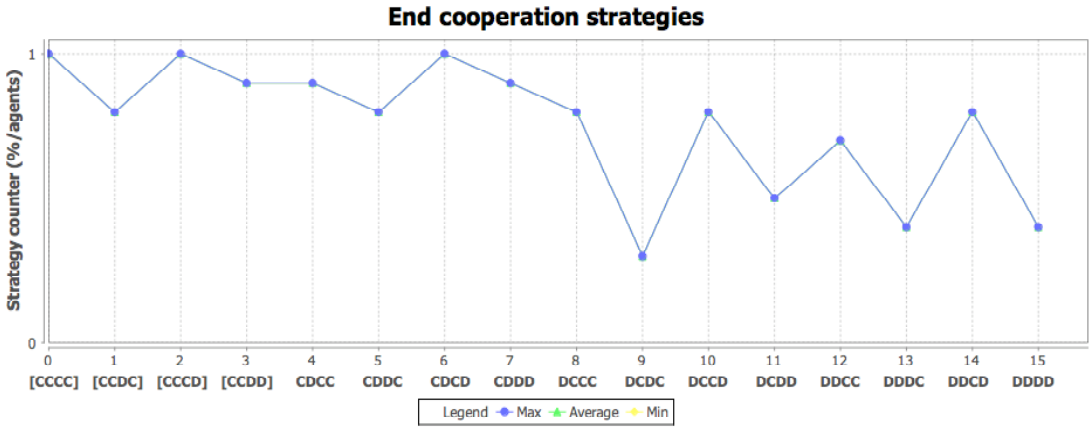
Picture 13: cooperative strategies representation after 1000 evolutions using the 8-8-1 ADANN with SigmoidMiddle2.

To conclude, longer evolutions caused by specific network topologies or more strict transfer functions caused cooperative and defensive strategies to be expressed more notably.

5.3. ADANN Network Structure

Networks with a single fully connected hidden layer produced too much cooperation. SM2 from equation (6) had to be used to cope with very fast evolutions when the number of the last hidden layer nodes was greater than 4. Increasing complexity in the SM2 networks speeded up the evolution further and smoothened the averages graph.

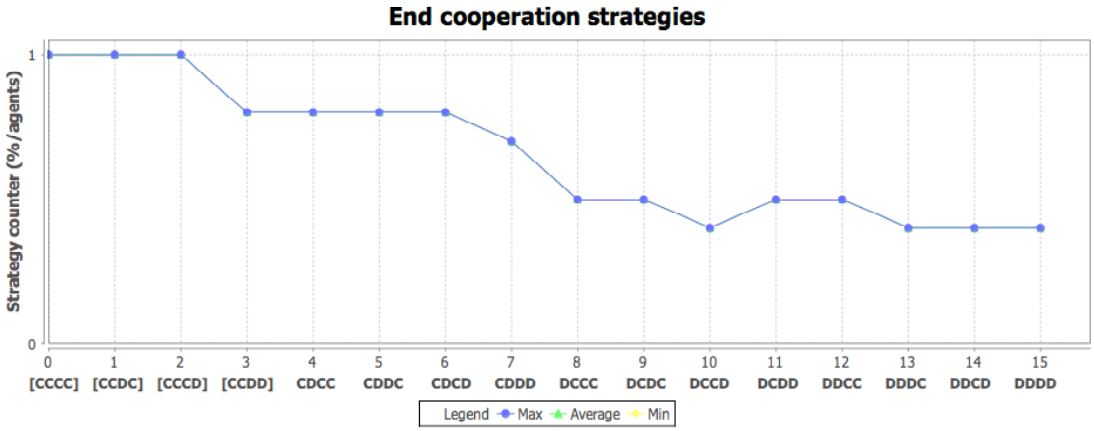
Decreasing the number of nodes to 1 restricted the cooperation drastically and it was impossible to reach its level above 20%. A network with 2 hidden layer nodes was however complex enough to produce cooperation and cooperative as well as some defensive strategies (Picture 14). However, Appendix C shows that there were great differences between evolved strategies after individual evolutions and that the desired result of evolving cooperative and defensive strategies was rarely achieved.



Picture 14: exemplary representation of evolved cooperative strategies in a population after 1672 generations with the 8-2-1 ADANN

Substituting the first fully connected hidden layer of 4 nodes with a T-layer speeded up the evolution process (compare rows 1 and 2 in Table 3). The same effect of smoothening the averages was observed.

Adding a second layer to networks with a T-layer increased the evolution time rapidly providing that the 2nd hidden layer was complex enough (compare rows 1, 7 and 9). The number of generations kept decreasing with increasing complexity of the 2nd hidden layer (row 10), although the computational speed was much lower. The 8-4*-20-1 network exhibited the most desirable behaviour. The cooperative and defensive strategies evolved more often than before, even when compared to the strategy-encoding model. Examples are provided on Picture 15 and in Appendix D. Examples for the strategy-encoding model can be found in Appendix A.



Picture 15: exemplary representation of evolved cooperative strategies in a population after 239 generations with the 8-2-1 ADANN

The resulting strategies were less desirable with networks with two fully connected hidden layers. Appendix E shows examples of substituting the T-layer for a fully connected layer. The cooperative and defensive strategies often seen with the 8-4*-20-1 network were slightly less frequent when using the 8-4-20-1 topology. Adding more complexity to the first layer (8-20-20-1 network) worsened the results. Not only did evolutions take longer on average (Table 3, row 6), but also cooperation under all circumstances was more frequent, especially when the evolution length was high. (Appendix F)*.

Adding a third layer to the ADANNs did not significantly affect the resulting strategies. The evolution lasted longer, possibly because of a greater search space created by a longer genotype. Different transformation functions in different layers were also tested, including linear and step functions. None of them made the network perform better than when the sigmoid functions were used.

5.4. Discussion of Results

A number of neural network architectures were discussed throughout the paper. With most of the NN setups, it was not possible to find one that would perform better or as well as the strategy-encoding model. This could have a number of reasons:

a) The role of mutation: the optimal mutation variance was found to be 0.1. Increasing the value led to increasingly random search and prolonged the evolution while a decreased mutation variance lost its effect completely. Nevertheless, mutation of one gene had only a partial and indirect effect on the network's output. On the other hand, mutation of a single gene in the strategy-encoding model meant a certain change of a response action.

**The examples provided in Appendices D-F are selected examples of analysis of 20 individual evolutions with each set up. Even though the results varied by evolution, the general characteristic mentioned in the text remained unchanged.*

b) Dynamics of the evolution: it was discovered by examining individual agent actions and change of their genotypes throughout the evolution that cooperative individuals were less frequent when the NN models were used. It was harder for the cooperators to achieve fitness comparable with the others since defectors often dominated populations. Climbing towards the 90% cooperations target was generally smoother with the strategy-encoding model. The NN models often decayed to total defection, although the recovery usually took only a few generations.

Another difference between the strategy-encoding and the ADANN models was that while the first model produced sensible averages, averages of the latter exhibited too much cooperation. However, individual cases of evolution with the 8-4*-20-1 model often produced better solutions than individual strategy-encoding evolutions. More importantly, while longer evolutions generally produced more cooperative strategies in the NN models, the effect was not visible with strategy encoding. Examples can be found in Appendices A and D.

A common feature of the strategy-encoding and ADANN models was in the effect of the population size. Using 20 agents with PAR=40 always resulted in longer evolutions, although the evolved strategies did not change significantly. This is in contrast with Harrald's and Fogel's report (1996) where they implied that at least a population of 20 needed to be used in order to evolve cooperation.

The paper considered evolution of cooperation with memory of 2 steps back. It would be interesting to see whether the reported evolution dynamics and strategy results would persists if longer memory was implemented or if more network parameters like number of nodes or shape of the transfer functions were subject to the evolutionary process. Finally, using automatic image- or data-processing methods could have provided a more precise analysis of individual evolutions.

Apart from indications of better results after individual evolutions with ADANNs, neural networks generally provide a better scalability, even though they very often require a longer genotype. Scalability can be useful when adding more memory or other input types to the individuals. Furthermore, switching from discrete to continuous inputs has almost no implementation cost in the NN models, while the strategy-encoding model would have to change fundamentally.

6. Conclusion

This paper examined the evolution of cooperation in the Iterated Prisoner's Dilemma with a memory of length 2. An altered microbial genetic algorithm was used to guide the evolution without the intervention of individual learning.

Axelrod's strategy-encoding model was implemented and the evolved strategies were discussed. The model was then compared with a number of artificial neural networks where weights were encoded in the genotype. A neural network inspired by the work of Harrald and Fogel (1996) produced poor results in terms of evolved strategies due to the fact that cooperation and defection actions shared the same nodes and groups of strategies could not evolve separately. Better results were achieved by using the action-discriminating network where each action had a separate node and therefore a separate effect on the output. It was shown that given a certain setup, this network had stronger tendencies towards evolving cooperative and defensive strategies during individual evolutions, even when compared with the strategy-encoding model.

It was also found that dynamics of the evolution in the strategy-encoding and neural network models were notably different. This could serve as a warning for the study of communication and interaction in general. All our observations and theories depend on the model we choose and are not necessarily universal or fully applicable to other models of the same phenomena. More importantly, understanding the evolution dynamics and interactions quality in existing models does not automatically mean understanding of social interactions in the nature. It is perhaps time to move away from simple and already understood models and implement more nature-inspired ones. Artificial neural networks may

provide one way of doing so, although care needs to be taken when choosing a level of abstraction. Evolution and emergence of sensible altruistic behaviours under more complicated circumstances would be a much greater achievement altogether.

Project code source

Appendix G provides Java code of the two main classes, World and Agent.
The full project is open-source and downloadable from
http://lenkaspace.net/downloads/neuralNetworksEvolutionOfCooperation.zip.
An applet where all the implemented models can be tested can be found on
http://lenkaspace.net/previews/neuralNetworksEvolutionOfCooperation.

Acknowledgements

I would like to thank Steffen Fiedler for his help with converting my Java code into a runnable web Applet.

References

Ashlock D. et al. (1996) 'Preferential Partner Selection in an Evolutionary Study of Prisoner's Dilemma'. *Biosystems* 37: 99-125

Axelrod R. (1984) *The Evolution of Cooperation*, Basic Books

Axelrod R. (1987) 'The evolution of Strategies in the Iterated Prisoner's Dilemma' In: *Genetic Algorithms and Simulated Annealing*, Davis L. (ed.), Kaufmann M.

Harrald P.G. and Fogel D.B. (1996) 'Evolving continuous behaviours in the Iterated Prisoner's Dilemma' *Biosystems*, 37:135-145

Harvey I. (1996) 'The Microbial Genetic Algorithm'. Unpublished manuscript

Lomborg B. (1996) 'Nucleus and Shield: The Evolution of Social Structure in the Iterated Prisoner's Dilemma'. *American Sociological Review*, 61:04:278-307

O'Madadhain J. (2004) 'Neural Network-Based Strategies for the Iterated Prisoner's Dilemma'. Unpublished work

Rapport A. and Dale P. S. (1966) 'The "End" and "Start" Effects in Iterated Prisoner's Dilemma'. The *Journal of Conflict Resolution*, 10:363-366

Sandholm T. W. and Crites R. H (1995) 'Multiagent Reinforcement Learning in the Iterated Prisoner's Dilemma' *Biosystems*, 37:147-166

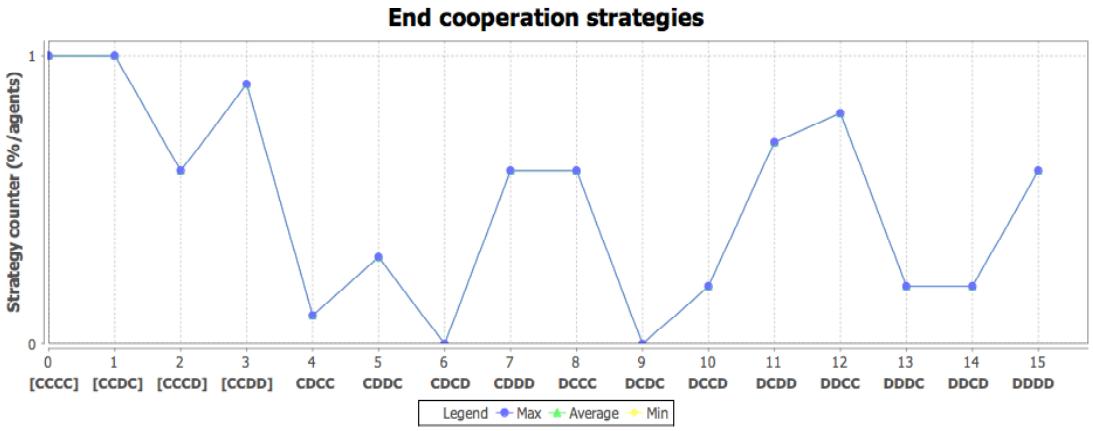
Smucker M. D., Stanley E. A. and Ashlock D. (1994) 'Analyzing Social Network Structures in the Iterated Prisoner's Dilemma with Choice and Refusal'. Department of Computer Sciences Technical Report CS-TR-94-1259, UW-Madison

Spector L. and Klein J. (2005) 'Trivial Geography in Genetic Programming'. In: *Genetic Programming Theory and Practice III*, Yu T., Riolo R.L. and Worzel B. (eds.), Kluwer Academic Publishers, Boston, MA, 109–124

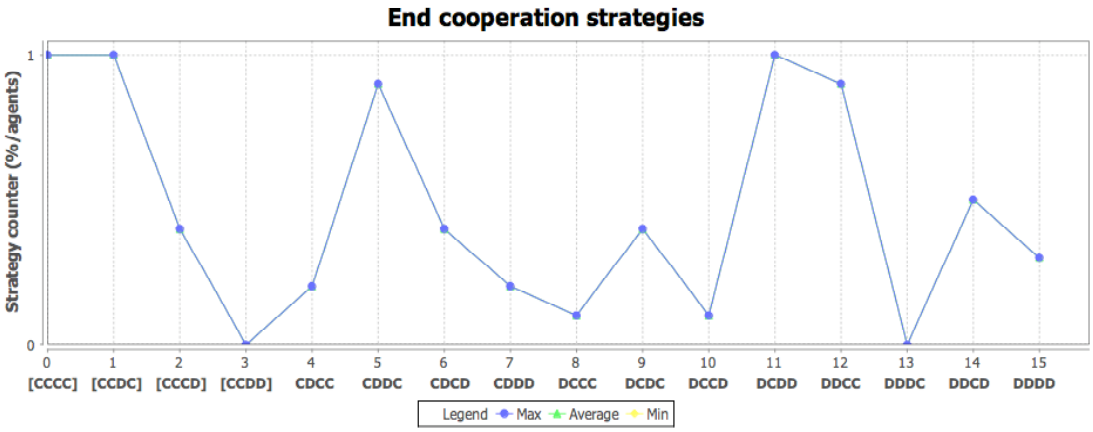
Wu J. and Axelrod R. (1995) 'How to Cope with Noise in the Iterated Prisoner's Dilemma'. *Journal of Conflict Resolution*, 39:183-189

Appendix A

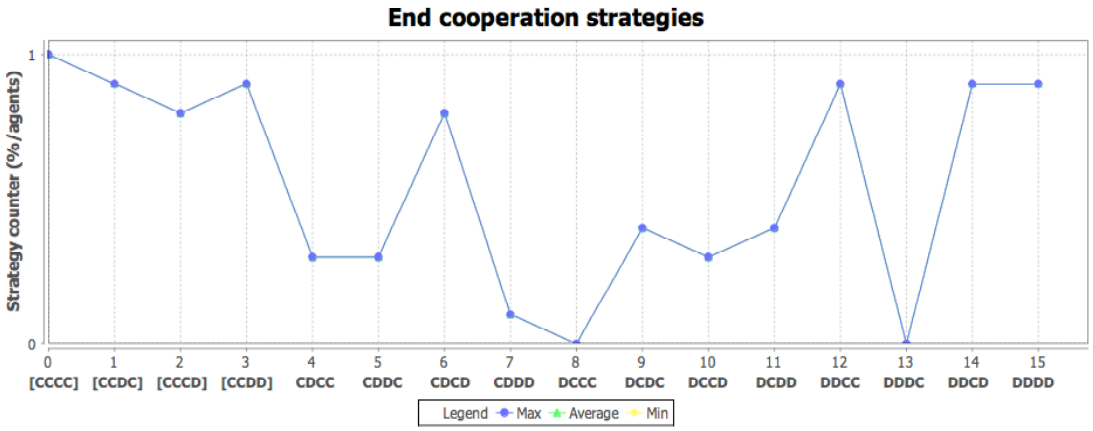
Exemplary representations of evolved strategies in individual evolutions with the strategy-encoding model and POP=10, PAR=20.
Explanation of graphs can be found in section 3.1.



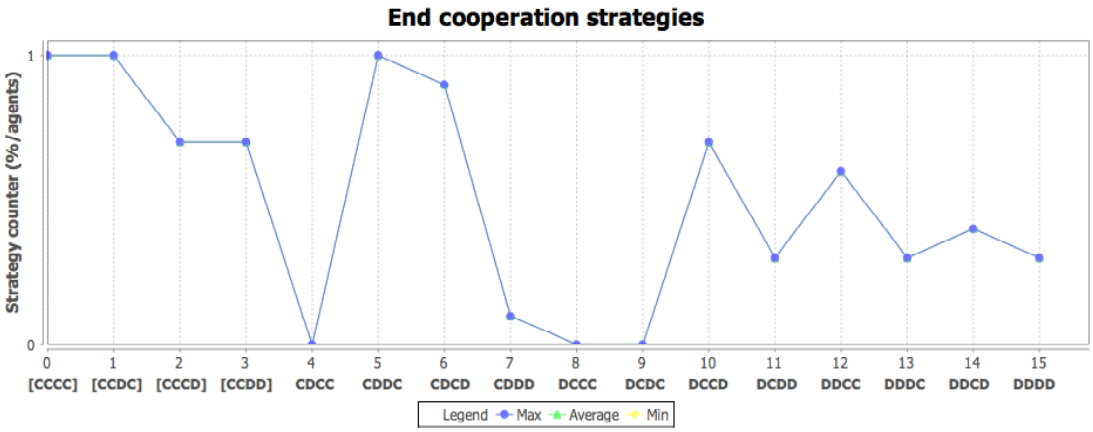
Picture 1: Evolution ended after 8 generations



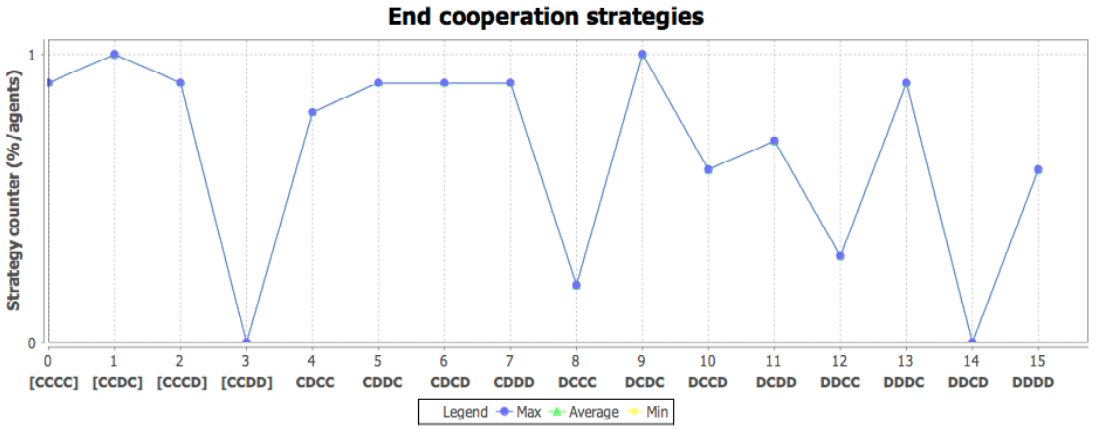
Picture 2: Evolution ended after 14 generations



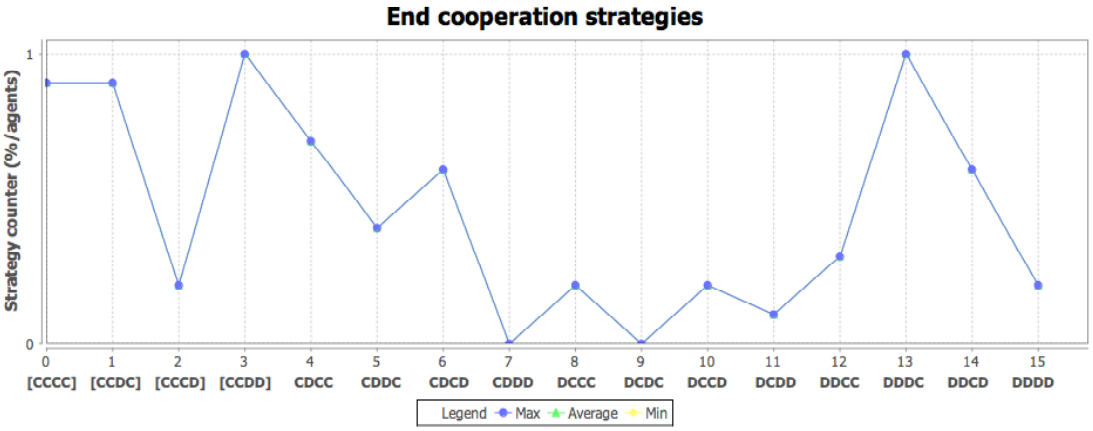
Picture 3: Evolution ended after 135 generations



Picture 4: Evolution ended after 147 generations



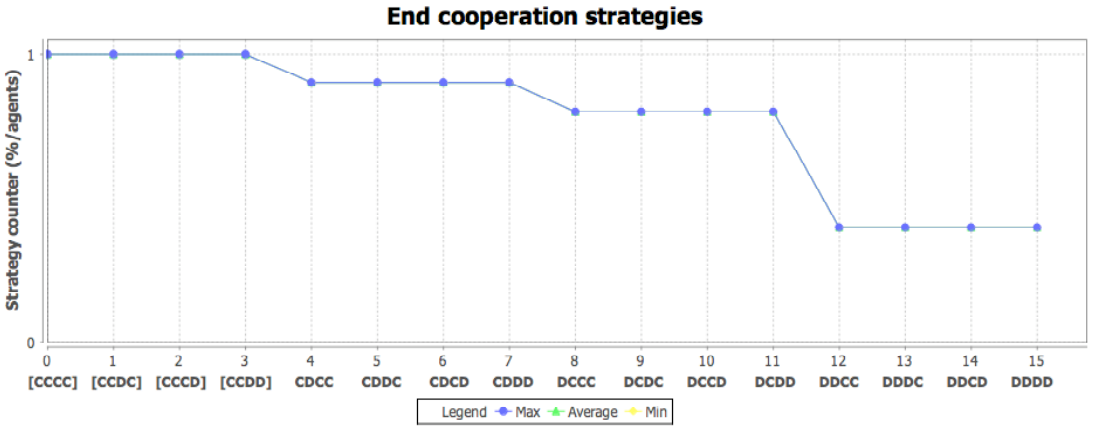
Picture 5: Evolution ended after 875 generations



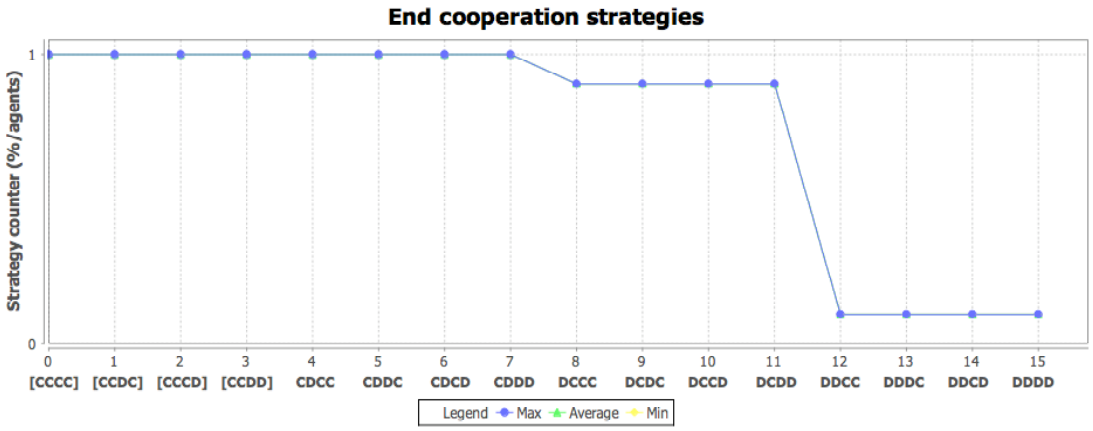
Picture 6: Evolution ended after 2572 generations

Appendix B

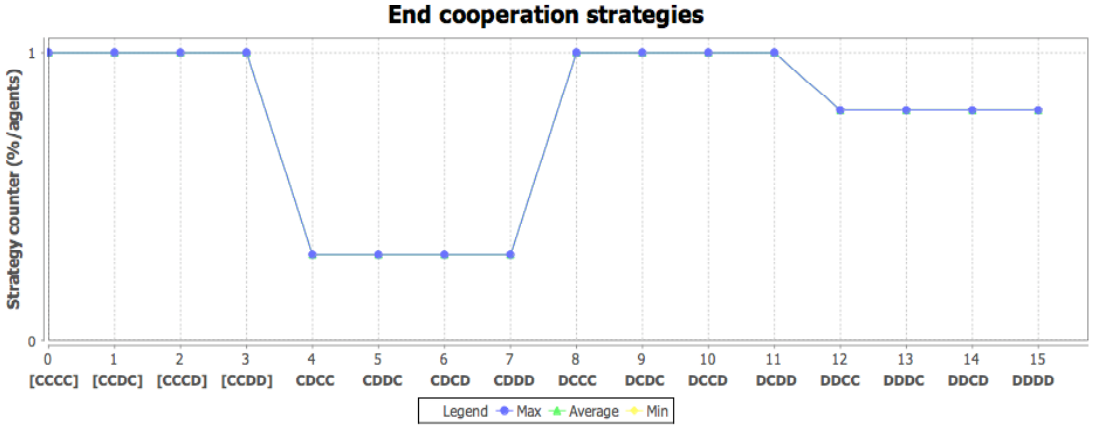
Exemplary representations of evolved strategies in individual evolutions with the 4-8-1 Fogel-inspired ANN



Picture 1: Evolution ended after 35 generations



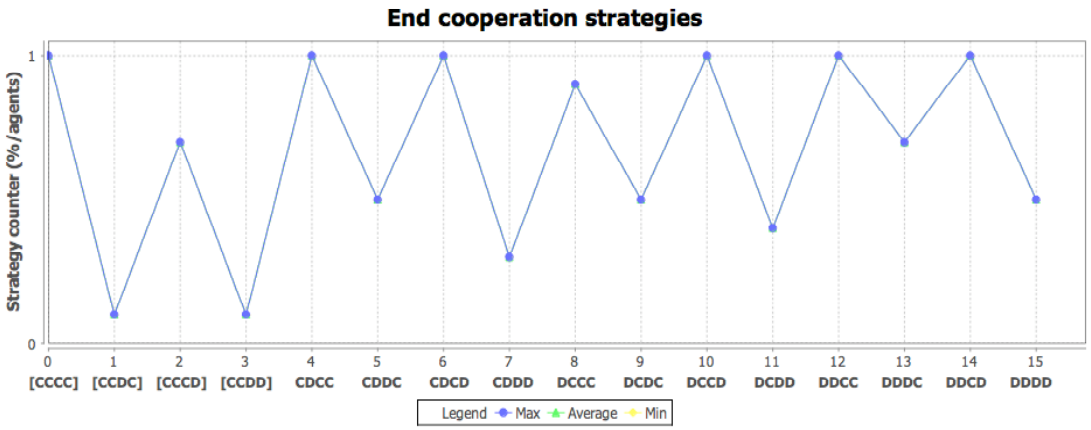
Picture 2: Evolution ended after 55 generations



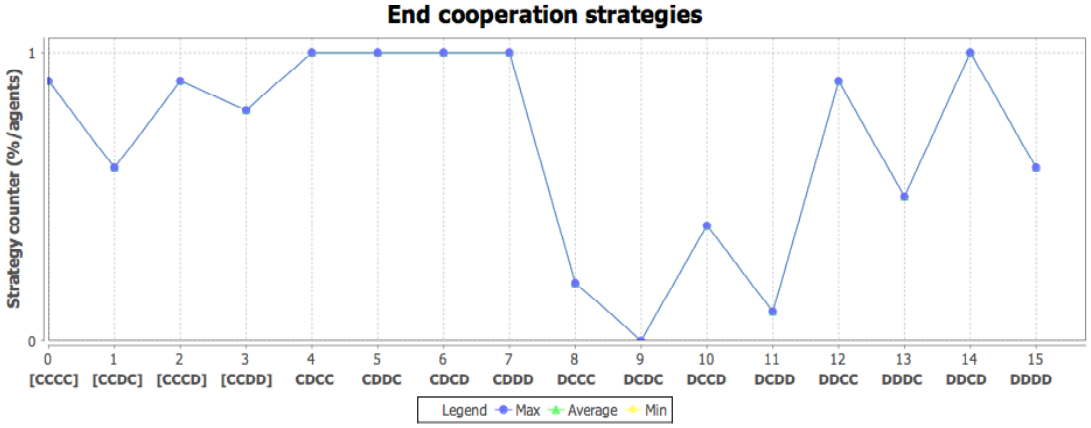
Picture 3: Evolution ended after 158 generations

Appendix C

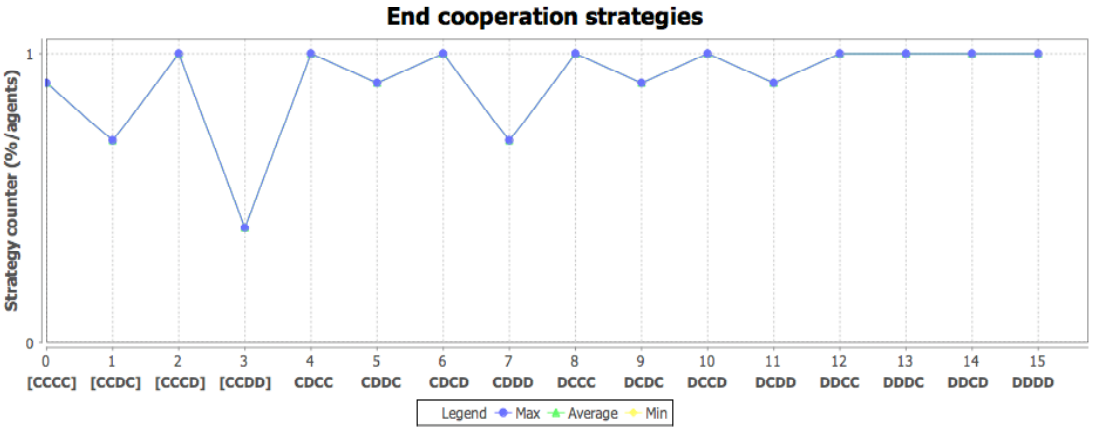
Exemplary representations of evolved strategies in individual evolutions with the 8-2-1 ADANN



Picture 1: Evolution ended after 735 generations



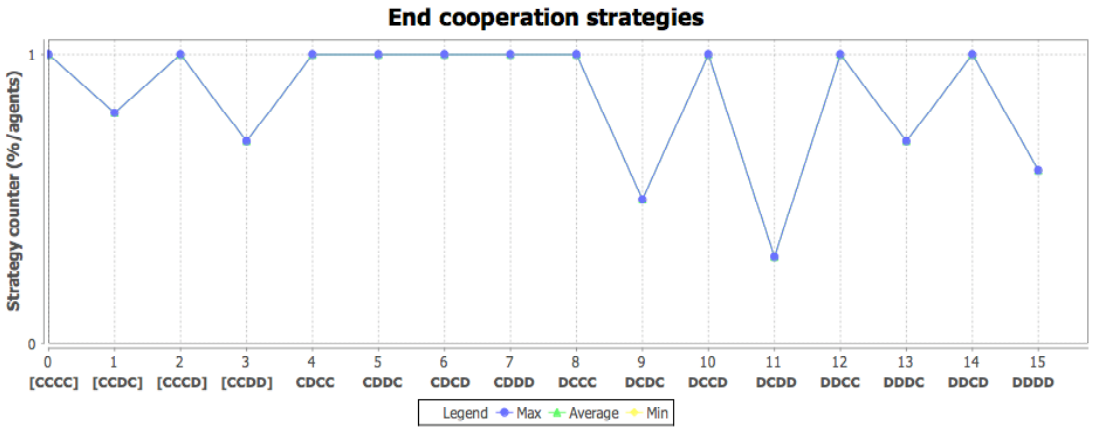
Picture 2: Evolution ended after 789 generations



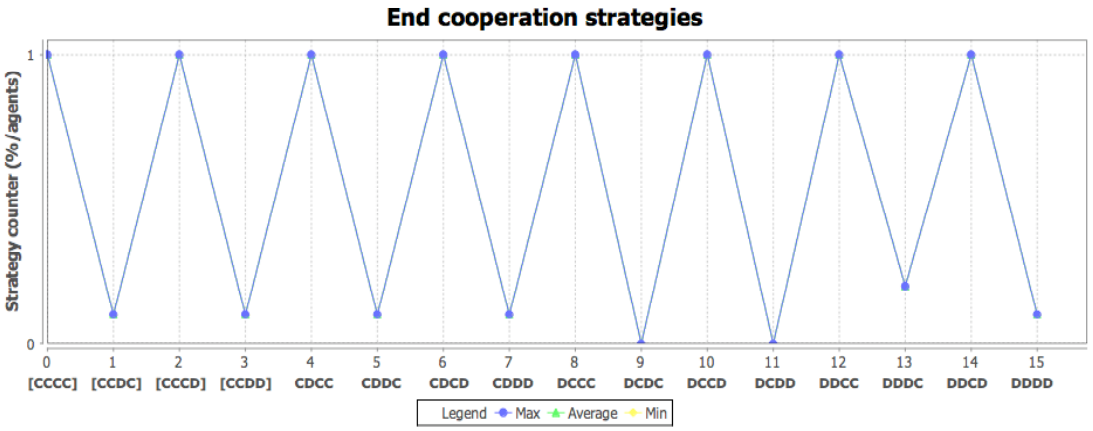
Picture 3: Evolution ended after 4157 generations

Appendix D

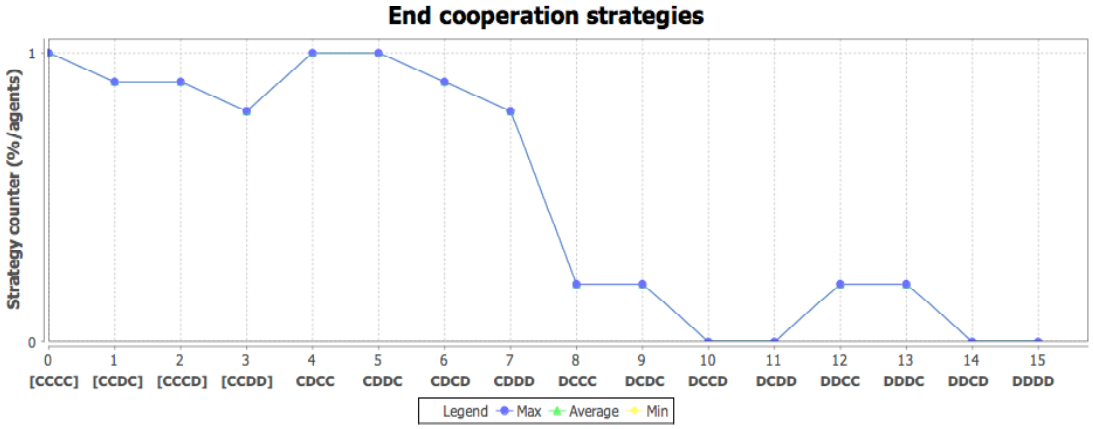
Exemplary representations of evolved strategies in individual evolutions with the 8-4*-20-1 ADANN



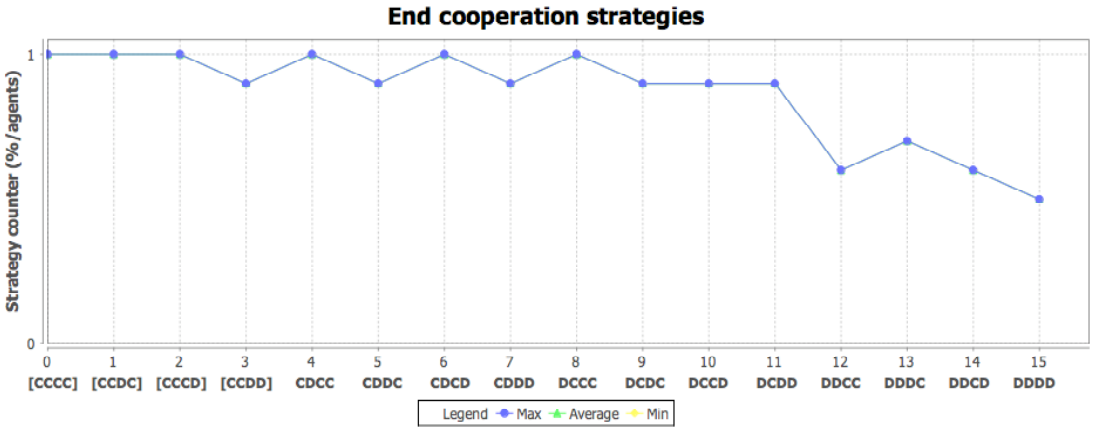
Picture 1: Evolution ended after 76 generations



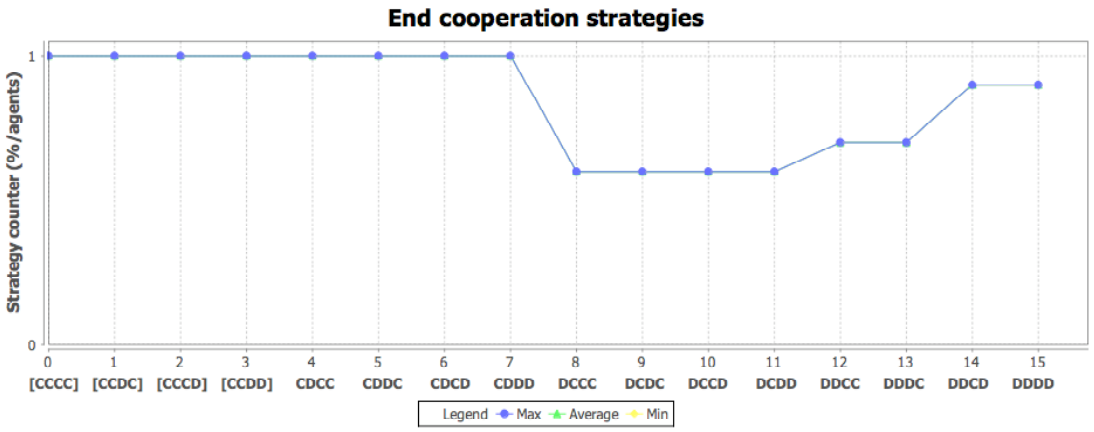
Picture 2: Evolution ended after 112 generations



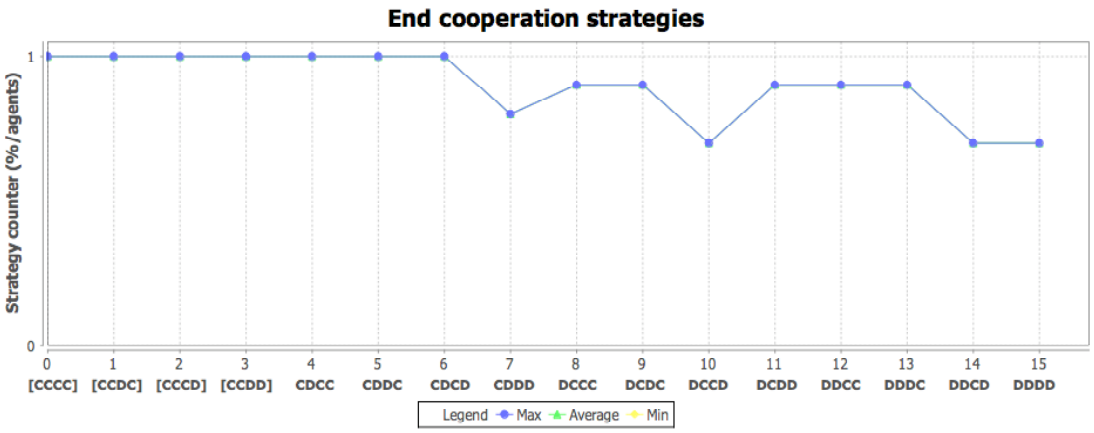
Picture 3: Evolution ended after 118 generations



Picture 4: Evolution ended after 202 generations



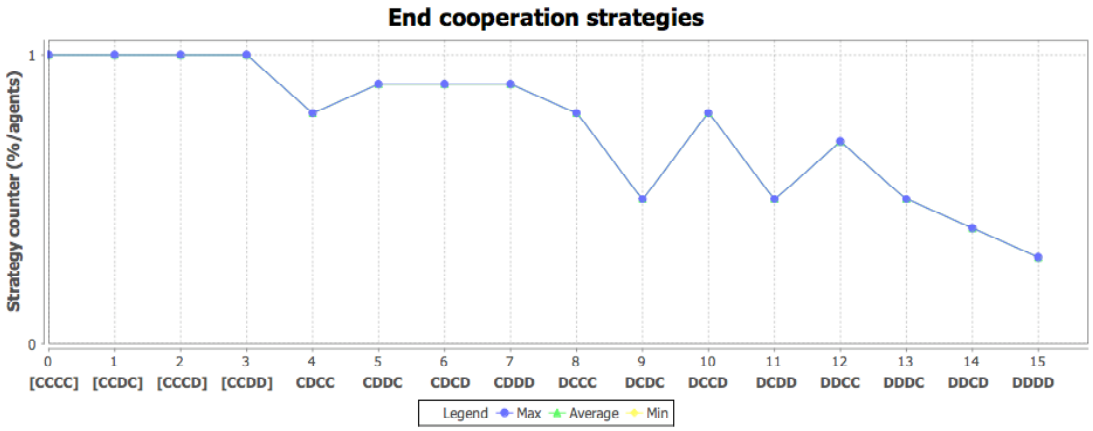
Picture 5: Evolution ended after 245 generations



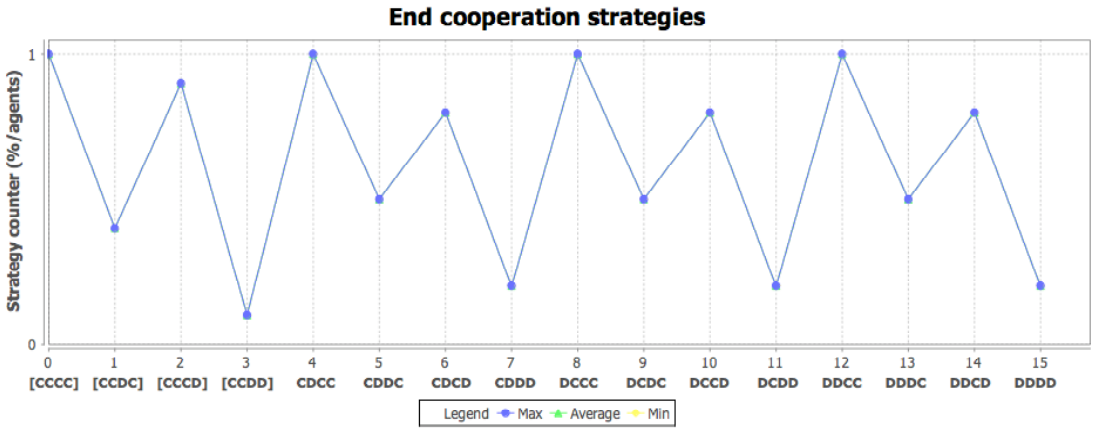
Picture 6: Evolution ended after 457 generations

Appendix E

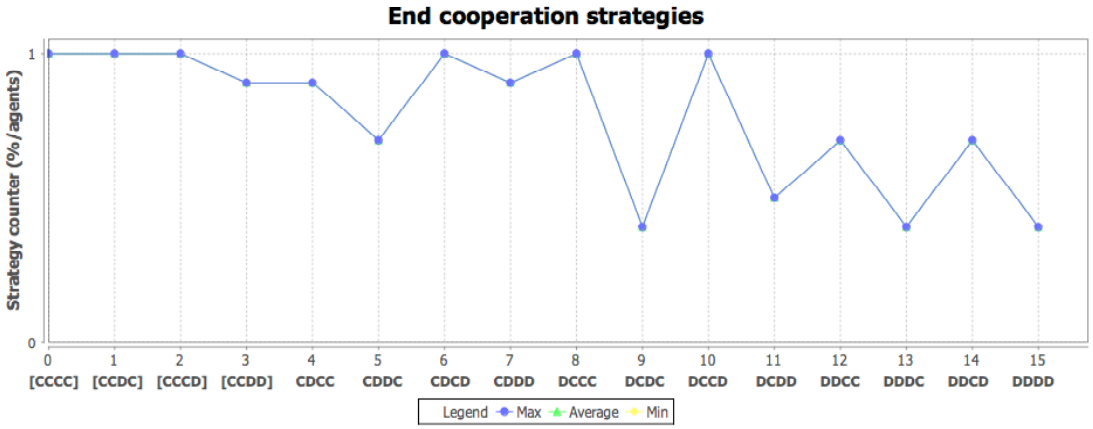
Exemplary representations of evolved strategies in individual evolutions with the 8-4-20-1 ADANN



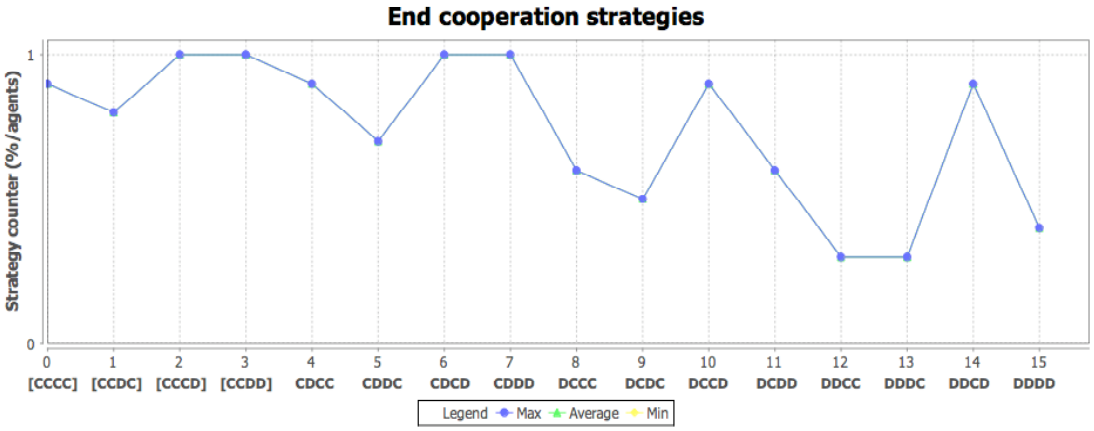
Picture 1: Evolution ended after 44 generations



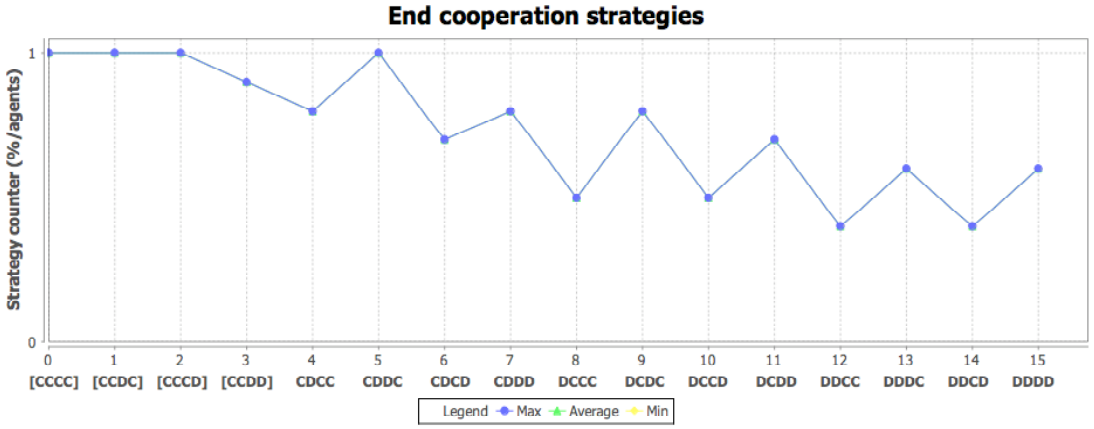
Picture 2: Evolution ended after 176 generations



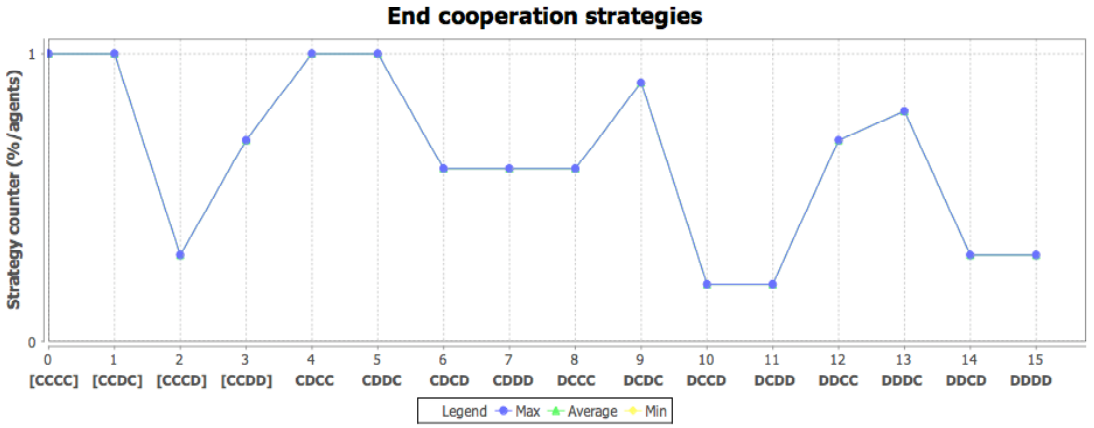
Picture 3: Evolution ended after 189 generations



Picture 4: Evolution ended after 227 generations



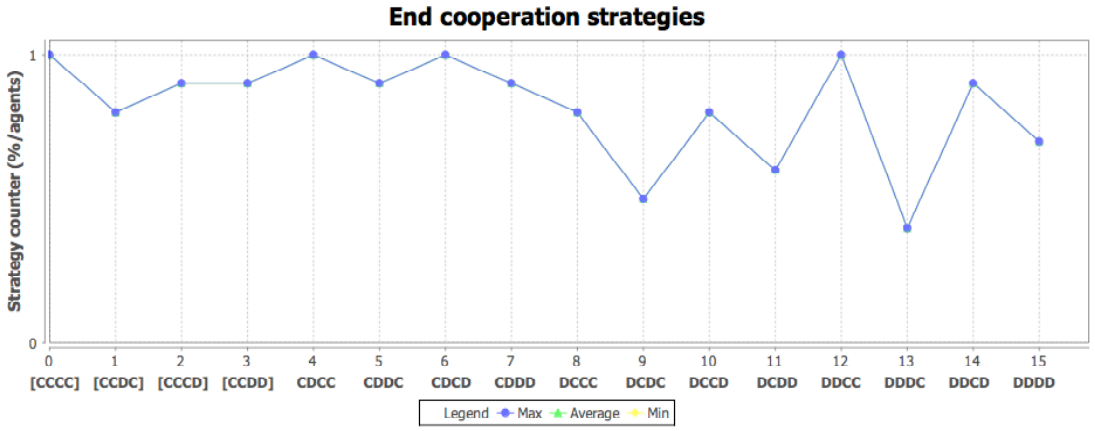
Picture 5: Evolution ended after 328 generations



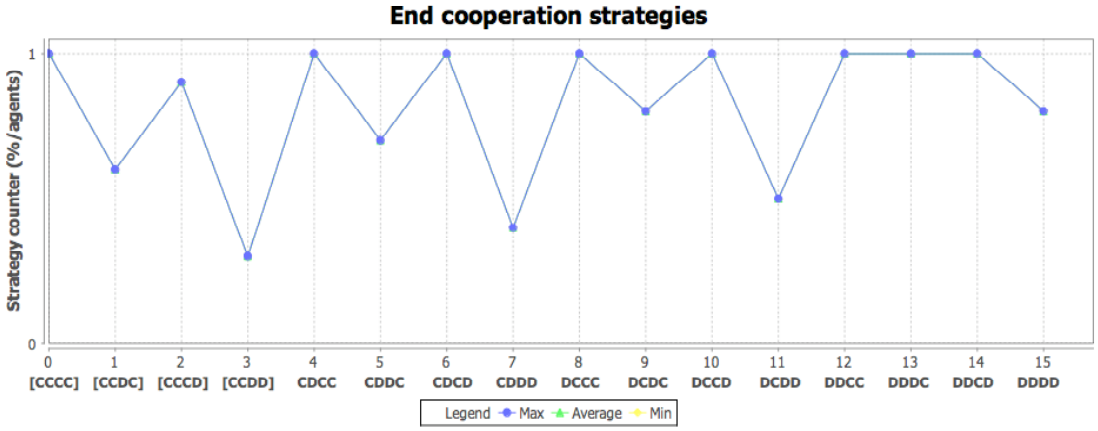
Picture 6: Evolution ended after 378 generations

Appendix F

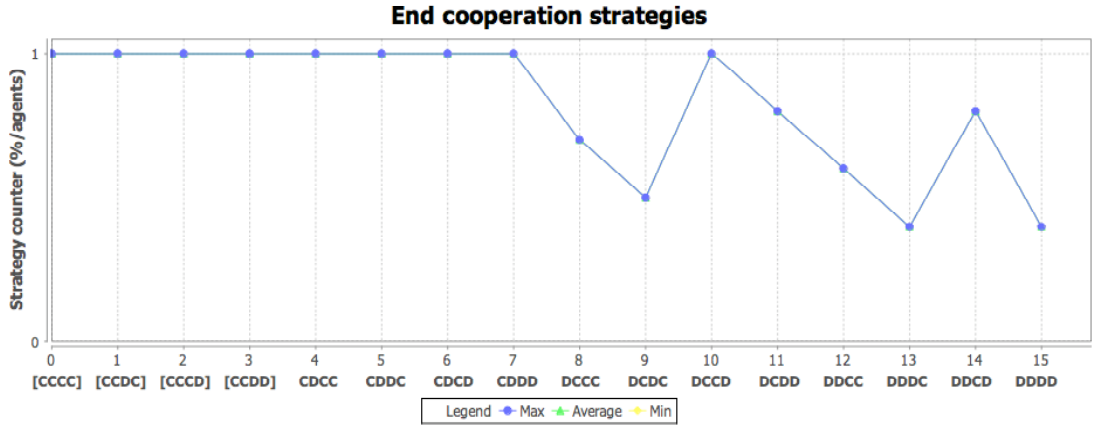
Exemplary representations of evolved strategies in individual evolutions with the 8-20-20-1 ADANN



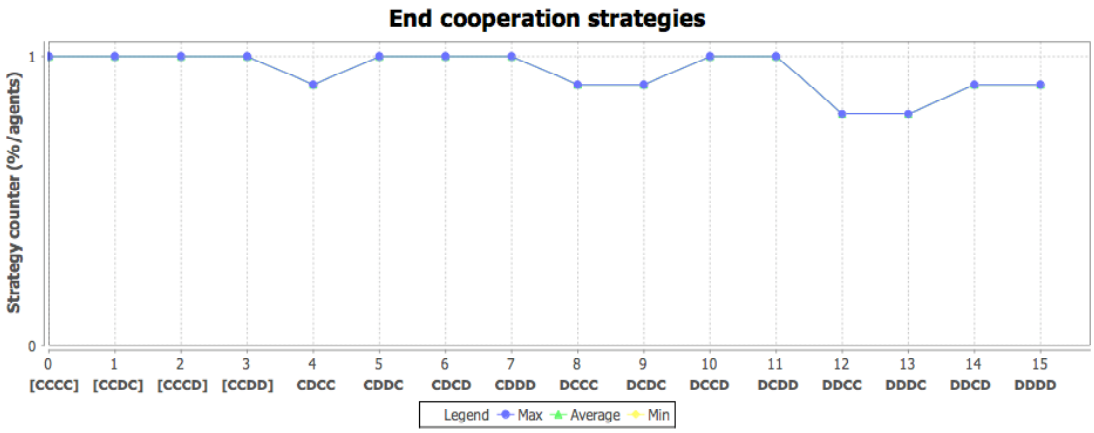
Picture 1: Evolution ended after 279 generations



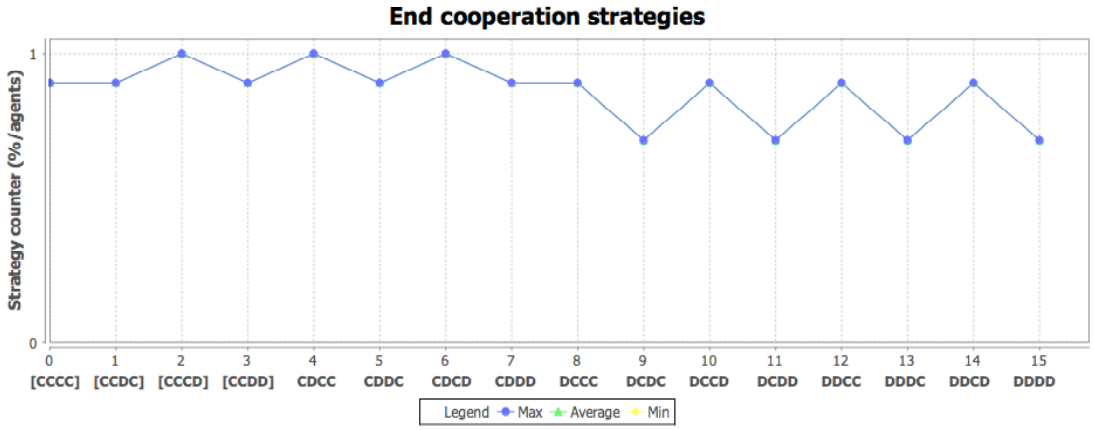
Picture 2: Evolution ended after 289 generations



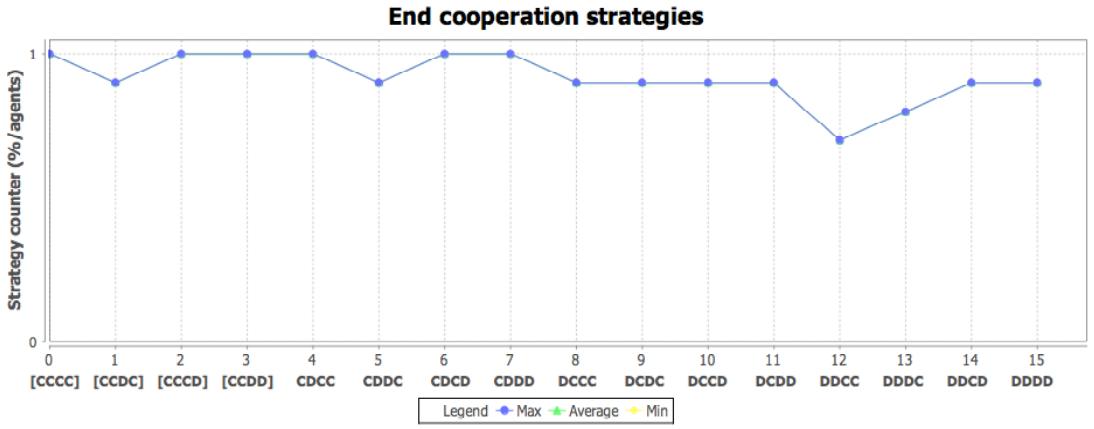
Picture 3: Evolution ended after 330 generations



Picture 4: Evolution ended after 634 generations



Picture 5: Evolution ended after 976 generations



Picture 6: Evolution ended after 1200 generations

Appendix G

A-life-relevant program code

```
/*
 * World class
 * - contains agents and implements the genetic algorithm
 */

public class World  {
    //---- constats
    …
    //----- objects
    private ArrayList <Agent> agentList;
            …
    //--- update loop
    private Object[] tournamentPairs;

    public World(PrisonersNoRecognition owner_) {
        …
        tournamentPairs = null;
        runningSimulation = true;
        //reset overall stats:
        totalGenerations = 0;
        minGenerations = 999999999;
        maxGenerations = 0;

        totalAverageFitness = 0;
        minAverageFitness = 999999999;
        maxAverageFitness = 0;
        …
        this.initEvolution();

    }

    public void initEvolution() {
        agentList = new ArrayList <Agent>();
        for (int i=0; i<NUM_AGENTS; i++) {
            agentList.add(new Agent(i, this));
        }
        …
        generation = 0;
        initGeneration();
    }

    public void endEvolution() {
        evolution++;
        //-------- end-of-evol stats:
        …
        totalGenerations += generation;
        if (generation < minGenerations) {
            minGenerations = generation;
        }
        if (generation > maxGenerations) {
            maxGenerations = generation;
        }

        double totalFitn = 0;
        …
```

```
        for (int i=0; i<NUM_AGENTS; i++) {
            totalFitn += this.agentList.get(i).getFitness();
                                    …
        }
        …
        //------ fitness stats
        …
        double averageFitn = (double)totalFitn / (double) NUM_AGENTS;
        totalAverageFitness += averageFitn;
        if (averageFitn < minAverageFitness) {
            minAverageFitness = averageFitn;
        }
        if (averageFitn > maxAverageFitness) {
            maxAverageFitness = averageFitn;
        }

        //---------
        if (evolution < NUM_EVOLUTIONS) {
            this.initEvolution();
        }
    }

    public void initGeneration() {
        //reset counters:
        cooperationsCount = 0;
        actionsCount = 0;
        mutCooperationsCount = 0;
        tournamentNo = 0;
        …
        //reset agents:
        for (int i=0; i< NUM_AGENTS; i++) {
            this.agentList.get(i).initState();
        }
    }

    /*
     * update()
     * - called in each frame of the progam
     */
    public void update() {
        if (evolution < NUM_EVOLUTIONS) {
            if (tournamentNo < TOURNAMENTS_PER_GENERATION) {
                //------------------- randomly generate pairs of agents that will play:
                tournamentPairs = generatePairs(0,agentList.size());
                //------------------- make them play Nth  tournament in M rounds
                for (int i=0; i<tournamentPairs.length; i+=2) {
                    Agent player1 = agentList.get((Integer)tournamentPairs[i]);
                    Agent player2 = agentList.get((Integer)tournamentPairs[i+1]);
                    player1.initGameState();
                    player2.initGameState();
                    int rnds = Maths.getSingletonObject().getRandomInteger(150,160);
                    for (int tournamentRound=0; tournamentRound<rnds; tournamentRound++) {
                        …
                        Agent.Action pl1action = player1.doAction();
                        Agent.Action pl2action = player2.doAction();
                        player1.registerActions(pl2action, pl1action);
                        player2.registerActions(pl1action, pl2action);
                        player1.addPlayedWith(player2.getId());// getAlreadyPlayedWithList().add();
                        player2.addPlayedWith(player1.getId());
```

```
                        if (pl1action == Agent.Action.COOPERATE) {
                            cooperationsCount++;
                        }
                        if (pl2action == Agent.Action.COOPERATE) {
                            cooperationsCount++;
                        }
                        if (pl2action == Agent.Action.COOPERATE && pl1action == Agent.Action.COOPERATE) {
                            mutCooperationsCount+=2;
                        }
                        actionsCount += 2;
                        //---------compute payoffs and give them to players:
                        byte[] payoffs = this.calculatePayoffs(pl1action, pl2action);
                        player1.setPayoff(player1.getPayoff() + payoffs[0]);
                        player2.setPayoff(player2.getPayoff() + payoffs[1]);
                        //---------display output:
                        ...
                        }
                    // end of game
                    }
                }
                // end of tournament
                tournamentNo++;
            } else {
                //================== all tournaments complete, END OF GENERATION ============
                //percentage of cooperations out of total games played each generation
                double cooperationsPercent = (double)cooperationsCount / (double)actionsCount;
                double mutCooperationsPercent = (double)mutCooperationsCount / (double)actionsCount;
                boolean endEvolution = (cooperationsPercent >= TARGET_COOPERATION_PERCENT) || (generation >=
MAX_GENERATIONS);
                //------------------------------------------- end-of-generation statistics
                ...
                //------------------- end condition of evolution:
                if (endEvolution) {
                    this.endEvolution();
                } else {
                    //------------------------------------------- create new generation:
                    for (int i = 0; i<NUM_REPLICATED_AGENTS; i++) {
                        //====  USING GEOGRAPHY: second parent from a deme where genot1 is in the middle:
                        int agentNo1 = Maths.getSingletonObject().getRandomInteger(0, NUM_AGENTS-1);
                        int agentNo2 = (int)(agentNo1 - (double)DEME_SIZE/2 +
Maths.getSingletonObject().getRandomInteger(0,DEME_SIZE));
                        //wrap-around:
                        if (agentNo2 >= NUM_AGENTS) {
                            agentNo2 = agentNo2 - NUM_AGENTS;
                        } else if (agentNo2 < 0) {
                            agentNo2 = NUM_AGENTS + agentNo2;
                        }
                        Agent agent1 = agentList.get(agentNo1);
                        Agent agent2 = agentList.get(agentNo2);

                        Agent winner, looser;
                        if (agent1.getFitness() > agent2.getFitness()) {
                            winner = agent1;
                            looser = agent2;
                        } else {
                            looser = agent1;
                            winner = agent2;
                        }

                        // compare the genotypes to determine which one will be passing genes,
```

```
                        // if both the same fitness keep them both:
                        if (agent1.getFitness() != agent2.getFitness()) {
                            //change the genotype of looser according to winner:
                            looser.changeGenotype(winner);
                        } else {
                            //make sure looser at least mutates:
                            looser.changeGenotype(looser);
                        }
                    }
                    generation++;
                    initGeneration();
                }
            }
        } else if (runningSimulation){
            this.endSimulation();
        }

    }


    public void endSimulation() {
        …
        double averGenerations = (double) totalGenerations / (double) NUM_EVOLUTIONS;
        …
        double averAverFitness = (double) totalAverageFitness / (double) NUM_EVOLUTIONS;
        …
        this.runningSimulation = false;
    }


    //=====================================================================
    private byte[] calculatePayoffs(Agent.Action pl1action, Agent.Action pl2action) {
        byte[] payoffs = new byte[2];
        if (pl1action == Agent.Action.COOPERATE) {
            if (pl2action == Agent.Action.COOPERATE) {
                //both cooperate, both receive 3:
                payoffs[0] = 3;
                payoffs[1] = 3;
            } else {
                //pl1 cooperates, pl2 defects -> pl2 receives 5, other 0:
                payoffs[0] = 0;
                payoffs[1] = 5;
            }
        } else {
            if (pl2action == Agent.Action.COOPERATE) {
                //pl2 cooperates, pl1 defects -> pl1 receives 5, other 0:
                payoffs[0] = 5;
                payoffs[1] = 0;
            } else {
                //both defect, both receive 1
                payoffs[0] = 1;
                payoffs[1] = 1;
            }
        }
        return payoffs;
    }

    private Object[] generatePairs(int startNum, int endNum) {
        ArrayList<Integer> pairs = new ArrayList<Integer>();
        while (pairs.size() < NUM_AGENTS) {
```

```
            Integer nextInt = Integer.valueOf(Maths.getSingletonObject().getRandomInteger(startNum, endNum-1) +
startNum);
            if (!pairs.contains(nextInt)) {
                boolean canAdd = true;
                if (pairs.size() %2 == 1 ) {
                    Integer a = pairs.get(pairs.size()-1);
                    Agent player1 = agentList.get(a);
                    //make sure this pair did not play in this generation:
                    if (player1.alreadyPlayedWith(nextInt)) {
                        canAdd = false;
                        counter++;
                        if (counter == 10) {
                            pairs = new ArrayList<Integer>(); //try other combinations...
                            canAdd = true;
                        }
                    }
                }
                if (canAdd)  {
                    counter = 0;
                    pairs.add(nextInt);
                }
            }
        }
        return (Object[]) pairs.toArray();
    }

    ...

}



/*
 * Agent class
 * - translates a genotype into an action, engages with other agents
 */
public class Agent extends BaseModel implements Comparable {
    ...
    private ArrayList<Integer> alreadyPlayedWithList;
    private double[] genotype;
    private double[] inputs;
    private double[] nodes;
    private int payoff;
    private int roundsPlayed;
    private int agentNo;

    private String strategiesRecord;
    private double[][]inputCombinationsForNN;

    ArrayList <String> possibleScenarios;

    public enum Action {
       COOPERATE, DEFECT
    }

    public Agent(int id_, World world_)  {
        super(id_);

        inputCombinationsForNN = new double [16] [Agent.INPUTS_LENGTH];
```

```
        double[]opt0 = { 1,0, 1,0, 1,0, 1,0}; inputCombinationsForNN[0] = opt0;
        double[]opt1 = { 1,0, 1,0, 0,1, 1,0}; inputCombinationsForNN[1] = opt1;
        double[]opt2 = { 1,0, 1,0, 1,0, 0,1}; inputCombinationsForNN[2] = opt2;
        double[]opt3 = { 1,0, 1,0, 0,1, 0,1}; inputCombinationsForNN[3] = opt3;

        double[]opt4 = { 1,0, 0,1, 1,0, 1,0}; inputCombinationsForNN[4] = opt4;
        double[]opt5 = { 1,0, 0,1, 0,1, 1,0}; inputCombinationsForNN[5] = opt5;
        double[]opt6 = { 1,0, 0,1, 1,0, 0,1}; inputCombinationsForNN[6] = opt6;
        double[]opt7 = { 1,0, 0,1, 0,1, 0,1}; inputCombinationsForNN[7] = opt7;

        double[]opt8 = { 0,1, 1,0, 1,0, 1,0}; inputCombinationsForNN[8] = opt8;
        double[]opt9 = { 0,1, 1,0, 0,1, 1,0}; inputCombinationsForNN[9] = opt9;
        double[]opt10 = { 0,1, 1,0, 1,0, 0,1}; inputCombinationsForNN[10] = opt10;
        double[]opt11 = { 0,1, 1,0, 0,1, 0,1}; inputCombinationsForNN[11] = opt11;

        double[]opt12 = { 0,1, 0,1, 1,0, 1,0}; inputCombinationsForNN[12] = opt12;
        double[]opt13 = { 0,1, 0,1, 0,1, 1,0}; inputCombinationsForNN[13] = opt13;
        double[]opt14 = { 0,1, 0,1, 1,0, 0,1}; inputCombinationsForNN[14] = opt14;
        double[]opt15 = { 0,1, 0,1, 0,1, 0,1}; inputCombinationsForNN[15] = opt15;

        possibleScenarios = new ArrayList<String>();

        possibleScenarios.add("1111"); //RR
        possibleScenarios.add("1101"); //RS
        possibleScenarios.add("1110"); //RT
        possibleScenarios.add("1100"); //RP

        possibleScenarios.add("1011"); //TR
        possibleScenarios.add("1001"); //TS
        possibleScenarios.add("1010"); //TT
        possibleScenarios.add("1000"); //TP

        possibleScenarios.add("0111"); //SR
        possibleScenarios.add("0101"); //SS
        possibleScenarios.add("0110"); //ST
        possibleScenarios.add("0100"); //SP

        possibleScenarios.add("0011"); //PR
        possibleScenarios.add("0001"); //PS
        possibleScenarios.add("0010"); //PT
        possibleScenarios.add("0000"); //PP

        this.initState();
    //create a random genotype:
        this.genotype = new double [GENOTYPE_LENGTH];
            if (USING_NN) {
            //generate initial weights and inputs as double numbers, inputs will be later translated to 0s and1s
            if (!USING_FOGEL_INPUTS) {
                //numbers between 0 and 1
                for (int i=0; i<GENOTYPE_LENGTH; i++) {
                    Double gene = Maths.getSingletonObject().getRandomDouble(true);
                    this.genotype[i] = gene;
                }
            } else {
                //numbers between -1 and 1
                for (int i=0; i<GENOTYPE_LENGTH; i++) {
                    //weigths
                    Double gene = (double)Maths.getSingletonObject().getRandomInteger(-100, 100)/(double)100;
                    this.genotype[i] = gene;
                }
```

```
            }
        } else {
            for (int i=0; i<GENOTYPE_LENGTH; i++) {
                Double rand = Maths.getSingletonObject().getRandomDouble(true);
                if (rand > 0.5) {
                    this.genotype[i] = 1;
                } else {
                    this.genotype[i] = 0;
                }
            }
        }
        this.initGameState();
    }

    public void initState() {
        if (USING_NN) {
            int INPUTS_LENGTH_IN_GENOTYPE = 0;
            if (!USING_FOGEL_INPUTS) {
                INPUTS_LENGTH = 8;
                INPUTS_LENGTH_IN_GENOTYPE = 4;
            } else {
                INPUTS_LENGTH = 4;
                INPUTS_LENGTH_IN_GENOTYPE = 4;
            }
            if (USING_NN_TYPE == Agent.NN_H1_TRANS) {
                GENOTYPE_LENGTH = 21; //8+4 + 4(bias) +8/2 + 1 //I, HL1 with 4 nodes,translation, O
            } else if (USING_NN_TYPE == Agent.NN_H1_TRANS_H2) {
                GENOTYPE_LENGTH = INPUTS_LENGTH + H1_NODES + (H1_NODES+1)*H2_NODES +
(H2_NODES+1)*1 + INPUTS_LENGTH_IN_GENOTYPE;
            } else if (USING_NN_TYPE == Agent.NN_H1) {
                GENOTYPE_LENGTH = (INPUTS_LENGTH + 1)*H1_NODES + (H1_NODES + 1)*1 +
INPUTS_LENGTH_IN_GENOTYPE;
            } else if (USING_NN_TYPE == Agent.NN_H1_H2) {
                GENOTYPE_LENGTH = (INPUTS_LENGTH + 1)*H1_NODES + (H1_NODES+1)*H2_NODES +
(H2_NODES + 1)*1 + INPUTS_LENGTH_IN_GENOTYPE;
            } else if (USING_NN_TYPE == Agent.NN_H1_H2_H3) {
                GENOTYPE_LENGTH = (INPUTS_LENGTH + 1)*H1_NODES + (H1_NODES+1)*H2_NODES +
(H2_NODES+1)*H3_NODES + (H3_NODES + 1)*1 + INPUTS_LENGTH_IN_GENOTYPE;
            }
        } else {
            GENOTYPE_LENGTH = 20; //4*4+4
            INPUTS_LENGTH = 4;
        }
        this.payoff = 0;
        this.roundsPlayed = 0;
        this.inputs = new double[INPUTS_LENGTH];
        ...
        this.alreadyPlayedWithList = new ArrayList <Integer>();
        this.initGameState();
    }

    public void initGameState() {
        //create initial hypothetical inputs based on last N genes:
        if (this.genotype != null) {
            if (USING_NN) {
                if (!USING_FOGEL_INPUTS) {
                    int counter = 0;
                    for (int i=0; i<INPUTS_LENGTH/2; i++) {
                        double geneVal = this.genotype[i + (GENOTYPE_LENGTH - INPUTS_LENGTH/2)];
                        if (geneVal > 0.5) {
```

```
                        this.inputs[counter++] = 1;
                        this.inputs[counter++] = 0;
                    } else {
                        this.inputs[counter++] = 0;
                        this.inputs[counter++] = 1;
                    }
                }
            } else {
                for (int i=0; i<INPUTS_LENGTH; i++) {
                    double geneVal = this.genotype[i + (GENOTYPE_LENGTH - INPUTS_LENGTH)];
                    if (geneVal > 0) {
                        this.inputs[i] = 1;
                    } else {
                        this.inputs[i] = -1;
                    }
                }
            }
        }
        else {
            for (int i=0; i<INPUTS_LENGTH; i++) {
                inputs[i] = this.genotype[GENOTYPE_LENGTH - INPUTS_LENGTH+i];
            }
        }
    }
    ...
}

//============================================== PRISONERS DILEMMMA
public Agent.Action doAction() {
    //compute cooperation probability:
    double cooperationProb = genotype[0];
    double randNum = Maths.getSingletonObject().getRandomDouble(true);

    if (USING_NN) {
        cooperationProb = this.calculateCooperationProbWithNN(this.inputs);
    } else {
        String str = "";
        for (int i=0; i<INPUTS_LENGTH; i++) {
            str += (int)this.inputs[i];
        }
        int whatHappenedIndex = possibleScenarios.indexOf(str);
        cooperationProb = this.genotype[whatHappenedIndex];
    }
    // decide if C or D:
    if (randNum <= cooperationProb && cooperationProb != 0) {
        return Agent.Action.COOPERATE;
    } else {
        return Agent.Action.DEFECT;
    }
}

private double calculateCooperationProbWithNN(double[] inputLayer) {
    double cooperationProb = 0;
    int weightsOffset = 0;
    int hiddenLayer1Size = H1_NODES; //set by UI
    double []hiddenLayer1 = new double[hiddenLayer1Size];


    //calculate hidden layer 1
    if (USING_NN_TYPE == Agent.NN_H1_TRANS || USING_NN_TYPE == Agent.NN_H1_TRANS_H2 ) {
```

```
            //=============== 'translation' hidden layer
            for (int i=0; i<hiddenLayer1Size; i++) {
                /*========= all actions from 1 timestep, 1 actor to 1 node (H1=4) ========= */
                double a = this.genotype[i*3] * inputLayer[i*2];
                double b = this.genotype[i*3+1]*inputLayer[i*2+1];
                double c = this.genotype[i*3+2]*1; //bias
                hiddenLayer1[i] = (double)1/2* (a+b) + (double)1/2*c; //max val: 1
                                                    // values multiplied by 0.5, becayse max for (a+b) = 1
                                                    // max for bias = 1, so we effectively have as if 2 inputs
            }
            //activate hidden layer 1:
            for (int i=0; i<hiddenLayer1Size; i++) {
                hiddenLayer1[i] = this.activate(hiddenLayer1[i], Agent.ACTIVATION_SIGMOID, 0.5); //middle is 0.5, had only
2 nodes attached to each
            }
            weightsOffset = INPUTS_LENGTH+hiddenLayer1Size; //inputs + number of biases
        } else {
            //======== fully connected first layer:
            for (int i=0; i<hiddenLayer1Size; i++) {
                double weightOfConnection = 0;
                if (!USING_FOGEL_INPUTS) {
                    weightOfConnection = (double)1/(INPUTS_LENGTH/2+1); // only INPUTS_LENGTH/2,
                                                                        // because half of the inputs
                                                                        // will have value 0
                } else {
                    weightOfConnection = (double)1/(INPUTS_LENGTH+1);
                }
                for (int j=0; j<INPUTS_LENGTH; j++) {
                    int where = i*(INPUTS_LENGTH+1) + j;
                    hiddenLayer1[i] +=  weightOfConnection * (this.genotype[where] * inputLayer[j]); //max val would be 4
                }
                int where = i*(INPUTS_LENGTH+1) + INPUTS_LENGTH;
                hiddenLayer1[i] += weightOfConnection * this.genotype[where]; // add bias
            }
            double resultMiddle;
            if (!USING_FOGEL_INPUTS) {
                resultMiddle = 0.5 ;
            } else {
                resultMiddle = 0.5;
            }
            //activate hidden layer 1:
            for (int i=0; i<hiddenLayer1Size; i++) {
                hiddenLayer1[i] = this.activate(hiddenLayer1[i], Agent.ACTIVATION_SIGMOID, resultMiddle);
            }
            weightsOffset = hiddenLayer1Size*(INPUTS_LENGTH+1);
        }

        //============== second layer:
        int hiddenLayer2Size = H2_NODES;
        double []hiddenLayer2 = new double[hiddenLayer2Size];

        if (USING_NN_TYPE == Agent.NN_H1_TRANS_H2 || USING_NN_TYPE >= Agent.NN_H1_H2 ) {
            for (int i=0; i<hiddenLayer2Size; i++) {
                for (int j=0; j<hiddenLayer1Size; j++) {
                    int where = weightsOffset + i*(hiddenLayer1Size+1) + j;
                    hiddenLayer2[i] += (double)1/(hiddenLayer1Size+1) * (this.genotype[where] * hiddenLayer1[j]);
                }
                int where2 = weightsOffset + i*(hiddenLayer1Size+1) + hiddenLayer1Size;
                hiddenLayer2[i] += (double)1/(hiddenLayer1Size+1) * this.genotype[where2]; // add bias
            }
```

```
        double resultMiddle = 0.5;
        //activate hidden layer 2:
        for (int i=0; i<hiddenLayer2Size; i++) {
            hiddenLayer2[i] = this.activate(hiddenLayer2[i], Agent.ACTIVATION_SIGMOID, resultMiddle);
        }
        weightsOffset = weightsOffset + hiddenLayer2Size*(hiddenLayer1Size+1);
    }

    //============== third layer:
    int hiddenLayer3Size = H3_NODES;
    double []hiddenLayer3 = new double[hiddenLayer3Size];

    if (USING_NN_TYPE >= Agent.NN_H1_H2_H3) {
        for (int i=0; i<hiddenLayer3Size; i++) {
            for (int j=0; j<hiddenLayer2Size; j++) {
                int where = weightsOffset + i*(hiddenLayer2Size+1) + j;
                hiddenLayer3[i] += (double)1/(hiddenLayer2Size+1) * (this.genotype[where] * hiddenLayer2[j]);
            }
            int where2 = weightsOffset + i*(hiddenLayer2Size+1) + hiddenLayer2Size;
            hiddenLayer3[i] += (double)1/(hiddenLayer2Size+1) * this.genotype[where2]; // add bias
        }
        double resultMiddle = 0.5;
        //activate hidden layer 3:
        for (int i=0; i<hiddenLayer3Size; i++) {
            hiddenLayer3[i] = this.activate(hiddenLayer3[i], Agent.ACTIVATION_SIGMOID, resultMiddle);
        }
        weightsOffset = weightsOffset + hiddenLayer3Size*(hiddenLayer2Size+1);
    }

    //=========== output
    int lastLayerSize = 0;
    double []lastLayer = null;
    if (USING_NN_TYPE == Agent.NN_H1_TRANS || USING_NN_TYPE == Agent.NN_H1) {
        lastLayerSize = hiddenLayer1Size;
        lastLayer = hiddenLayer1;
    } else if (USING_NN_TYPE < NN_H1_H2_H3){
        lastLayerSize = hiddenLayer2Size;
        lastLayer = hiddenLayer2;
    } else {
        lastLayerSize = hiddenLayer3Size;
        lastLayer = hiddenLayer3;
    }

    //=========== calculate output based on last layer:
    double output = 0;
    if (USING_SM2) {
        lastLayerSize = 4;
    }
    double resultMiddle = (double)1/lastLayerSize * (double)5/6;
    for (int i=0; i<lastLayerSize; i++) {
        double a = this.genotype[weightsOffset+i]*(double)lastLayer[i];
        output += (double)1/(lastLayerSize+1) * a ; //maxVal:1
    }
    int where = weightsOffset + lastLayerSize;
    output += (double)1/ (lastLayerSize+1) * this.genotype[where]; //add bias

    //activate output:
    output = activate(output, Agent.ACTIVATION_SIGMOID, resultMiddle);

    //decide if C or D, set cooperation probability to 1 or 0 depending on value of output:
```

```
        if (!USING_FOGEL_INPUTS) {
            cooperationProb = activate(output, Agent.ACTIVATION_STEP, 0.5);
        } else {
            cooperationProb = activate(output, Agent.ACTIVATION_STEP, 0.0); //all above 0 cooperate
        }
        return cooperationProb;
    }

    private double activate(double valueIn, int activationType, double middle) {
        double result = 0;
        if (activationType == Agent.ACTIVATION_LINEAR) {
            result = valueIn;
        } else if (activationType == Agent.ACTIVATION_STEP) {
            if (valueIn > middle) {
                result = 1;
            } else {
                result = 0;
            }
        } else if (activationType == Agent.ACTIVATION_SIGMOID) {
            double temp = (double)(12* (0.5 / middle)); // will move middle of sigmoid curve to the middle specified as
param of function
            if (USING_NN) {
                if (!USING_FOGEL_INPUTS) {
                    result = (1/ (1 + Math.exp(-(-6 + temp*valueIn)))); //sigmoids have different middles but return val always
from <0;1>
                } else {
                    result = -1 + 2*(1/ (1 + Math.exp(-(-0 + 7*valueIn)))); //sigmoids have different middles but return val
always from <-1;1>
                    //result = -1 + 2*(1/ (1 + Math.exp(-(-7 + 14*valueIn))));

                }
            } else {
                result = (1/ (1 + Math.exp(-(-48 + 96*valueIn))));
            }
        }
        return result;
    }


    public void registerActions(Agent.Action opponentAction, Agent.Action selfAction) {
        //push array of memory:
        if (USING_NN) {
            if (!USING_FOGEL_INPUTS) {
                //push self action
                inputs[6] = inputs[2];
                inputs[7] = inputs[3];
                //push other action
                inputs[4] = inputs[0];
                inputs[5] = inputs[1];
                if (opponentAction == Agent.Action.COOPERATE) {
                    inputs[0] = 1.0;
                    inputs[1] = 0.0;
                } else {
                    inputs[0] = 0.0;
                    inputs[1] = 1.0;
                }
                if (selfAction == Agent.Action.COOPERATE) {
                    inputs[2] = 1.0;
                    inputs[3] = 0.0;
                } else {
```

```
                    inputs[2] = 0.0;
                    inputs[3] = 1.0;
                }
            } else {
                //push self action
                inputs[3] = inputs[1];
                //push other action
                inputs[2] = inputs[0];
                if (opponentAction == Agent.Action.COOPERATE) {
                    inputs[0] = 1.0;
                } else {
                    inputs[0] = -1.0;
                }
                if (selfAction == Agent.Action.COOPERATE) {
                    inputs[1] = 1.0;
                } else {
                    inputs[1] = -1.0;
                }
            }
        } else {
            //push memory:
            this.inputs[2] = this.inputs[0];
            this.inputs[3] = this.inputs[1];
            if (opponentAction == Agent.Action.COOPERATE) {
                this.inputs[0] = 1;
            } else {
                this.inputs[0] = 0;
            }
            if (selfAction == Agent.Action.COOPERATE) {
                this.inputs[1] = 1;
            } else {
                this.inputs[1] = 0;
            }
        }
    }


    //====================================== GENOTYPE

    public void changeGenotype(Agent other) {
        //go through each gene of winner, copy it onto the looser with some probability and mutate with some probability:
        for (int g=0; g<GENOTYPE_LENGTH; g++ ) {
            if (Maths.getSingletonObject().getRandomDouble(true) < RECOMBINATION_PROBABILITY) {
                this.genotype[g] = other.getGenotype()[g];
            }

            if (USING_NN) {
                this.genotype[g] = Maths.getSingletonObject().getRandomGaussian(this.genotype[g],
MUTATION_VARIANCE);
                if (!USING_FOGEL_INPUTS) {
                    //make sure gene value is between 0 and 1:
                    if (this.genotype[g] < 0) {
                        this.genotype[g] = 0.0;
                    } else if (this.genotype[g] > 1) {
                        this.genotype[g] = 1.0;
                    }
                } else {
                    //make sure gene value is between -1 and 1:
                    if (this.genotype[g] < -1) {
                        this.genotype[g] = -1.0;
```

```
                    } else if (this.genotype[g] > 1) {
                        this.genotype[g] = 1.0;
                    }
                }
            } else {
                if (Maths.getSingletonObject().getRandomDouble(true) < MUTATION_PROBABILITY) {
                    if (this.genotype[g] == 1) {
                        this.genotype[g] = 0.0;
                    } else {
                        this.genotype[g] = 1.0;
                    }
                }
            }
        }
    }

    public double[] getGenotypeAsStratFromNN() {
        //hard-code inputs and test action, put into genotype[16], to get the same output as agent.getGenotype() for evolution with strategies:
        double [] genotype = new double[ReportController.NUM_STRATEGIES];
        for (int i=0; i<ReportController.NUM_STRATEGIES; i++) {
            double randNum = Maths.getSingletonObject().getRandomDouble(true);
            double myWouldBeCoopProb = this.calculateCooperationProbWithNN(inputCombinationsForNN[i]);
            // decide if C or D:
            if (randNum <= myWouldBeCoopProb && myWouldBeCoopProb != 0) { //  //cooperationProb >0.5
                genotype[i] = 1;
            } else {
                genotype[i] = 0;
            }
        }
        return genotype;
    }


    //=========================================================

    public void addPlayedWith(int agentNo) {
        if (!this.alreadyPlayedWithList.contains(agentNo)) {
            this.alreadyPlayedWithList.add(agentNo);
        }
    }

    public int compareTo(Object o) {
        Agent other =  (Agent)o;
        if (other.getFitness() < this.getFitness()) {
            return -1;
        } else if (other.getFitness() > this.getFitness()) {
            return 1;
        } else {
            //both same fitness, decide randomly:
            double randNum = Maths.getSingletonObject().getRandomDouble(true);
            if (randNum <= 0.5) {
                return 1;
            } else {
                return -1;
            }
        }

    }
```

```
//======================================================== GETTERS / SETTERS
...
public void setPayoff(int payoff) {
    this.payoff = payoff;
    this.roundsPlayed++;
}

public double getFitness() {
    return (double)this.payoff / (double)this.roundsPlayed;
}

...

public boolean alreadyPlayedWith(Integer nextInt) {
    if (this.alreadyPlayedWithList.contains(nextInt)) {
        return true;
    }
    return false;
}
```