Chapter 12

Cooperation

{chap:coop}

12.1 The prisoner's dilemma

{sec:IPD}

Under suspicion of breaking into the computer systems of Multi-National United, two hackers are being detained in separate cells and interrogated in turn by Agent Smith. Evidence gathered at the time of arrest being rather thin, each suspect is being told that he/she will get off with a lighter sentence if agreeing to provide additional evidence to ensure a guilty verdict for the other. For each suspect the best possible outcome arises by betraying the other, but only provided the other does not betray as well; mutual betrayal (defection) is dangerous because both suspects then end up in deeper trouble than they would have had they each opted to remain true to the other (cooperation), which is the dual move with the greatest *mutual* benefit.

Agent Smith

Danger: of mutual betra

This situation exemplifies the types of problems adressed by *game theory*. The key concept in the type of two-player game just described is the *payoff matrix*. In this matrix the element ij give the score obtained by making move i if the opponent plays move j; in terms of our two moves, Cooperate (C) and Defect (D), the payoff matrix used in all that follows is:

IPD: payoff matrix

$$\begin{array}{c} \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cc} C & D \end{array} \\ \begin{pmatrix} CC & CD \\ DC & DD \end{pmatrix} \end{array} = \begin{array}{c} \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cc} C & D \end{array} \\ \begin{pmatrix} 3 & 0 \\ 5 & 1 \end{pmatrix} \end{array} . \tag{12.1}$$

{eq:payoff}

On the basis of this specific payoff matrix, and without information regarding the other player's behavior, the best move is clearly to defect (average payoff 3, versus 1.5, if the opponent's move is equiprobably C or D). But the opponent can figure this out too, and therefore by the same logic will also defect. In that case both players obtain a score of 1, less than the equiprobable "average" payoff associated with (12.1). It is also much less than if both had cooperated (payoff 3 to each), but still more than a cooperating move met by a defection (payoff zero). What is then the best move ? Therein lies the prisoner's dilemma[1].

If the game involves only a single round, then retaliation is not a concern, and from a purely probabilist point of view the best strategy is to defect. This is no longer the case if the game is played as many consecutive rounds between the same two players. This is the *iterated prisoner's dilemma* (hereafter IPD). In such a situation the opponent's behavior in past rounds can be used to pick the "best" move for the current round. In the remainder of this chapter we consider the complex patterns of cooperation and competition that can take place between the following five classical IPD strategies:

IPD: defined

IPD: strategies

1. **ALLC**: Always cooperate; this "nice" strategy is *non-reactive*, since the current move is not influenced by the opponent's prior move(s).

[1] Numerical values other than those given in the payoff matrix (12.1) are of course allowed, but in general subject to the constraint $DC > CC > DD > CD$ for the class of games under consideration here. The dilemma materializes provided the entries in the payoff matrix satisfy the relation $2 \times CC > CD + DC$.

2. **ALLD**: Always defect; like ALLC, this "nasty" strategy is not influenced by the opponent's prior move(s).

3. **RAND**: Choose randomly but with equal probability between cooperation of defection, independently of the opponent's prior move(s) (non-reactive);

4. **TFT**: Tit-for-Tat: start off with cooperation, then simply reciprocate the opponent's latest move; TFT is an example of a reactive strategy, as it is influenced (in fact, for TFT it is completely determined) by the opponent's prior move.

5. **PAV**: The so-called "Pavlov" strategy: also a reactive strategy, which starts off with cooperation, then switches move ($C \to D$ or $D \to C$) whenever the opponent defects.

Reactive or not, these are all remain very simple strategies, that require one bit (TFT and PAV) or no bit (ALLC, ALLD and RAND) of memory to store relevant prior move information. Moreover they are all computationally quite simple; no need for a 3-pound universe of a brain to execute any of these!

The left half of Table 12.1 illustrates seven typical 20-round IPD game plays between different pairs of these five strategies. For each player in a given contest, the table lists the sequence of moves, D or C, with the corresponding score for each individual move listed above and below the sequence of moves, followed by the average score-per-round given for each player. Repeating this process for all combinations of our five strategies, including strategies playing against themselves, one can construct the following score matrix, now computed for 100 rounds per game so as to minimize the impact of the initial move on the final score[2], as opposed to 20 for the illustrative games of Table 12.1:

e matrix

{eq:ipdscore}

	ALLC	ALLD	RAND	TFT	PAV
ALLC	3.00	0.00	1.50	3.00	3.00
ALLD	5.00	1.00	3.00	1.04	3.00
RAND	4.00	0.50	2.25	2.23	2.84
TFT	3.00	0.99	2.26	3.00	3.00
PAV	3.00	0.50	2.01	3.00	3.00

(12.2)

Entries in this matrix give the average score-per-round of the strategy listed in the leftmost column playing against the strategy listed in the top line. The best payoff, 5.0 per round, is to ALLD when playing ALLC; this, of course, also yields the worst possible payoff for ALLC, namely zero. TFT loses the first round against ALLD, but then defects at all subsequent rounds (see Game 1 in Table 12.1). This leads to a small score differential in favor of ALLD, the more rounds the lesser this differential. PAV does worst than TFT against ALLD, because faced with systematic defection it alternates between C and D, for an average payoff of 0.5 (Game 2 in Table 12.1). ALLD does extract a good average score of 3 per round, playing against PAV, because the latter cooperates every second move. ALLC, TFT and/or PAV playing against one another always score 300 because they all open with cooperation, and TFT and PAV will only change to D if the opponent defects, which will not happen here (Games 4, 5, and 6). Playing against itself over many rounds, RAND simply collects the mean score of the whole payoff matrix (12.1); it obtains an almost identical score playing TFT (Game 3) since the latter, from the first defection onward, echoes its randomness; it exploits ALLC half the rounds on average, and is exploited by ALLD half the rounds as well.

Whether due to lack of sleep, illicit chemicals, or just plain bad lighting on the dance floor, in the real world we all sometimes make mistakes. This can be accomodated within the IPD by introducing a mistake probability p ($\ll 1$) that a strategy makes the "wrong" move at any given round. The right half of Table 12.1 illustrates the effect. Here in all cases player 1 is making a wrong move on round 5, while player 2 makes a wrong move on round 15 (lowercase

due to bad lighting

random

[2] In this 100-round score matrix, all scores implying the strategy RAND are averages over 10^4 statistically independent realizations of the 100-round game with the opponent.

Table 12.1: 20-rounds IPD faceoffs with and without noise{tab:ipd}

Game	Player	No noise	Score	With noise	Score
		51111111111111111111	1.2	51110511111111511111	1.55
1	ALLD	DDDDDDDDDDDDDDDDDDDD		DDDDcDDDDDDDDDDDDDD	
	TFT	CDDDDDDDDDDDDDDDDDDD		CDDDDCDDDDDDDDcDDDDD	
		01111111111111111111	0.95	01115011111111011111	1.05
		51515151515151515151	3.0	51513515151515515151	3.1
2	ALLD	DDDDDDDDDDDDDDDDDDDD		DDDDcDDDDDDDDDDDDD	
	PAV	CDCDCDCDCDCDCDCDCDCD		CDCDCCDCDCDCDCcDCDCD	
		01010101010101010101	0.5	01013010101010010101	0.6
		50505103333511051050	2.3	50503503333511351050	2.55
3	RAND	DCDCDDCCCCCDDDCDDCDC		DCDCcDCCCCCDDDCDDCDC	
	TFT	CDCDCDDCCCCCDDDCDDCD		CDCDCCDCCCCCDDcCDDCD	
		05050153333011501505	2.3	05053053333011301505	2.3
		33333333333333333333	3.0	33335015015015333333	2.65
4	TFT	CCCCCCCCCCCCCCCCCCCC		CCCCdCDDCDDCDDCCCCCC	
	PAV	CCCCCCCCCCCCCCCCCCCC		CCCCCDDCDDCDDCcCCCCC	
		33333333333333333333	3.0	33330510510510333333	2.4
		33333333333333333333	3.0	33335050505050111111	2.15
5	TFT	CCCCCCCCCCCCCCCCCCCC		CCCCdCDCDCDCDCDDDDDD	
	TFT	CCCCCCCCCCCCCCCCCCCC		CCCCCDCDCDCDCDdDDDDD	
		33333333333333333333	3.0	33330505050505111111	2.15
		33333333333333333333	3.0	33335133333333013333	2.75
6	PAV	CCCCCCCCCCCCCCCCCCCC		CCCCdDCCCCCCCCCDCCCC	
	PAV	CCCCCCCCCCCCCCCCCCCC		CCCCCDCCCCCCCCdDCCCC	
		33333333333333333333	3.0	33330133333333513333	2.75

"c" and "d"). Comparing the games (and scores) on the left and right halves of Table 12.1, one soon realizes that even an occasional wrong move can sometimes have a lasting and devastating impact. This is particularly striking with TFT playing against itself, where a single wrong move by one player will turn was was up to then an unbroken sequence of cooperation move into an alternance of Cooperation/Defection, leading to a significant drop in the scores for both players (see Game 5). PAV, on the other hand, shows error tolerance when playing against itself (Game 6), in that it can recover to mutual cooperation two rounds after any mistaken move. ALLC and ALLD are in some sense error tolerant by design, since their moves are not influenced by that of the opponent.

Figure 12.1 gives the listing of a C-function that returns the score obtained by one strategy (variable `p1`) playing another (variable `p2`), over a set 100 rounds. In the absence of mistakes (input variable `prob`= 0), a simple pre-computed look-up table for the scores is used, while a true 100-round IPD game is played when mistakes are allowed to occur ($0 <$ `prob` < 1).

```
int ipd( int p1, int p2, float prob )
{
/* Declarations/initialisations ------------------------------------------------ */
  int mvp1n, mvp1, mvp2n, mvp2, k ; float scorep1 ;
  int mv0[5]={1,0,0,0,1} ;                              /* initial move          */
  float payoff[2][2]={{3.,0.},{5.,1.}} ;                /* payoff matrix Eq (12.1)*/
  float score[5][5] ={ {300.,  0.,150.,300.,300.},      /* score  matrix Eq (12.2)*/
                       {500.,100.,300.,104.,300.},
                       {400., 50.,225.,223.,284.},
                       {300., 99.,226.,300.,300.},      /* NOTE: pre-computed for */
                       {300., 50.,201.,300.,300.} } ;   /* a 100-round IPD !      */
/* Executable ------------------------------------------------------------------ */
  if ( prob == 0 ) {                                    /* no mistakes           */
    scorep1=score[p1][p2] ;                             /* just use score matrix  */
  } else {                                              /* mistakes: play IPD     */
    mvp1=mv0[p1] ; mvp2=mv0[p2] ;                       /* make initial move      */
    if ( p1 == 2 ) { mvp1=floor(1.*rand()/RAND_MAX) }   /* special case: RAND     */
    if ( p2 == 2 ) { mvp2=floor(1.*rand()/RAND_MAX) }   /* special case: RAND     */
    scorep1=0 ;
    for (k=0 ; k < 100 ; k++) {                         /* play for 100 rounds    */
      switch (p1) {                                                 /* player 1        */
        case 0 : mvp1n=1                          ; break ;    /* ALLC            */
        case 1 : mvp1n=0                          ; break ;    /* ALLD            */
        case 2 : mvp1n=floor(2.*rand()/RAND_MAX) ; break ;    /* RAND            */
        case 3 : mvp1n=mvp2                       ; break ;    /* TFT             */
        case 4 : mvp1n=pow(mvp1,mvp2)             ; break ;    /* PAV             */
      }
      switch (p2) {                                                 /* player 2        */
        case 0 : mvp2n=1                          ; break ;    /* ALLC            */
        case 1 : mvp2n=0                          ; break ;    /* ALLD            */
        case 2 : mvp2n=floor(2.*rand()/RAND_MAX) ; break ;    /* RAND            */
        case 3 : mvp2n=mvp1                       ; break ;    /* TFT             */
        case 4 : mvp2n=pow(mvp2,mvp1)             ; break ;    /* PAV             */
      }
      if (1.*rand()/RAND_MAX < prob) { mvp1n=1-mvp1n } /* player 1 makes mistake */
      if (1.*rand()/RAND_MAX < prob) { mvp2n=1-mvp2n } /* player 2 makes mistake */
      scorep1+=payoff(mvp1n,mvp2n) ;                    /* cumulative payoff      */
      mvp1=mvp1n ; mvp2=mvp2n ;                         /* update for next round  */
      }                                                 /* end of this round      */
  }
  return scorep1 ;                                      /* score of player 1      */
}
```

Figure 12.1: Source code in the C-programming language for a function computing the score of one strategy (`p1`) playing another (`p2`). Strategies are identified as integers, here going from 0 to 4: $0 \equiv$ ALLC, $1 \equiv$ ALLD, $2 \equiv$ RAND, $3 \equiv$ TFT, and $4 \equiv$ PAV, with a `switch case` construct selecting the corresponding strategy. Internally the function uses "0" to indicate a defection, and '1" for cooperation; this allows (among other things) a compact implementation of PAV: its move (if player 1) is given by `mvp2` to the power `mvp1`. The effect of mistakes is to turn a C into a D, or vice versa, which here means turning a "1" into a "0" or vice versa; this is readily implemented by computing one minus the noise-free move. {`code:IPD`}

12.2 The fully-mixed IPD

{sec:ESS}

Things start becoming really interesting when groups of strategies interact in an evolutionary context. We consider a heterogeneous population of fixed size N, with the number N_i of each distinct strategies present in the population at generational iteration n defining their frequency; restricting ourselves to the five strategies introduced above, we have:

$$f_i^n = \frac{N_i}{N} \ , \qquad \sum_{i=1}^{5} f_i^n = 1 \ . \tag{12.3}$$

{???}

At each temporal iteration of the evolutionary simulation, each population member plays the IPD with every other member. From an ecological point of view, this is a *fully mixed* model. The collective total score S_i^n of each strategy at iteration n is then used to compute new frequencies for the next "generation", according to

IPD: fully mixed

$$f_i^{n+1} = N \times S_i^n \times \left(\sum_{i=1}^{5} S_i^n \right)^{-1} \ , \tag{12.4}$$

{eq:reprodrule}

where the index n measures the generational (temporal) iteration. Note that this "reproduction rule" implies that the frequency of a given strategy at generation $n+1$ is not just proportional to its mean score, but also to its frequency at generation n; in other words, the *reproduction rate* of a strategy is determined by its mean score, and that rate times its current frequency in the population sets its subsequent frequency in the next generation. Such a simulation is readily set up using as its operating core the IPD C-function listed in Fig. 12.1. Figure 12.2 shows a possible implementation in the C language.

C Code: for fully mixed

Figure 12.3 shows the result of two such fully-mixed evolutionary simulations, the first noise-free (top panel) and the second introducing a mistake probability $p = 10^{-2}$ (bottom panel). This noise level implies that over the 100 rounds of each game played, each player will make on average one mistake per game. In both cases the evolution starts off with an equal mixture of all 5 strategies ($f_i^0 = 0.2$, $i = 1...5$).

With the score-proportional reproduction rule given by eq. (12.4) and an initially equal mixture of strategies, the average values of each rows in (12.2) indicate which strategies will at first prosper and which will flounder. These mean values are the following: ALLC= 2.10, ALLD= 2.608, RAND= 2.364, TFT= 2.45, and PAV= 2.302, so we would expect ALLD to prosper and ALLC to decline. This is indeed what is observed during the first few iterations on the top panel of Fig. 12.3, but as the strategy frequencies change the mean values of the score matrix (12.2) in themselves are no longer a good evolutionary predictor. In fact these initial evolutionary trends change dramatically once ALLC has been nearly decimated by ALLD, by the tenth iteration or so. Consider that the mean scores for an equal mixture of the four remaining strategies would then be: ALLD= 1.608, RAND= 1.955, TFT= 2.313, and PAV= 2.128. All the mean scores have gone down, but ALLD is particularly hard hit, having lost its primary "prey". This favors TFT, which rises to eventually dominate the population. PAV finishes slightly above its initial frequency, with RAND and ALLC surviving at very low levels and ALLD exterminated. It is remarkable that ALLD, the strategy that appeared unbeatable on the basis of the IPD payoff matrix (12.1), is the only one that becomes extinct in this simulation.

In the presence of low-level noise (mistake probability $p = 10^{-2}$, bottom panel on Fig. 12.3), the evolution begins basically the same way, but once ALLD becomes extinct and ALLC and RAND fall to low frequencies, the error-correcting capability of PAV gives it an edge over TFT, and over the next 100 generation PAV slowly takes over the population, passing TFT at generation 45 here. Which strategy ends up dominating the final population in the presence of occasional mistakes can be quite sensitive to noise level. Starting always from an equal mixture of strategies, TFT dominates the population up to about $p = 8 \times 10^{-3}$, with PAV abruptly taking over at larger p values. An equally abrupt transition occurs at $p \simeq 7 \times 10^{-2}$, with TFT

```
#include <stdio.h>
#include <stdlib.h>
#define NT 100     /* number of generational iterations */
#define NP 100     /* target size of population  */
#define PM 0.0     /* mistake probability */
int main(void)
{
/* Declarations/initialisations ------------------------------------------ */
  int ipd( int, int, float ) ;                  /* IPD function (Fig 12.1)   */
  int makepop( float[5], float [NP+5]) ;
  int iter, k, ipop, iopp, np, pop[NP] ;
  float scores[5], f[5], score1, totalfit ;
/* Executable ------------------------------------------------------------ */
  for ( k=0 ; k<5 ; k++ ) { f[k]=0.2 ; }        /* Equal initial mixture     */
  np=makepop(frac,pop) ;                        /* make population array     */
  for ( iter=0 ; iter<NT ; iter++ ) {           /* generational iteration    */
    for ( k=0 ; k<5 ; k++ ) { fitstrag[k]=0. ; }
    totalfit=0. ;
    for ( ipop=0 ; ipop<np ; ipop++ ) {         /* Loop over population      */
      score1=0. ;
      for ( iopp=0 ; iopp<np ; iopp++ ) {       /* Loop over opponents       */
        if ( iopp != ipop ) {                   /* no playing with yourself  */
          score1+=ipd(pop(ipop),pop(iopp),PM) ; /* Play game                 */
        }
        scores(pop(ipop))+=score1/(np-1) ;      /* Average score of ipop     */
        totalfit+=scores(pop(ipop))             /* total score on the fly    */
      }                                         /* End loop over opponents   */
    }                                           /* End population loop       */
    for ( k=0 ; k<5 ; k++ ) { f[k]=NP*scores(k)/totalfit ;    /* Eq (12.4)   */
    np=makepop(f,pop) ;                         /* new population array      */
    printf ("Frequencies: %d,%f,%f,%f,%f,%f\n",iter,f[0],f[1],f[2],f[3],f[4]) ;
  }                                             /* End generational loop     */
}
int makepop( float frac[], float pop[] )        /* Builds population array   */
{
   int i, k, tot, ns[5] ;
   tot=0 ;
   for (k=0 ; k<5 ; k++) {
      ns[k]    = floor( NP*frac[k]+0.25 ) ;           /* Eq. (12.3)          */
      for (i=tot ; i<tot+ns[k] ; i++ ) { pop[i]=k ; } /* Fill pop array      */
      tot+= ns[k] ;
   }
   return tot ;                                      /* Population size     */
}
```

Figure 12.2: Source code in the C-programming language for a fully-mixed evolutionary simulation of a IPD-playing population of strategies. The C-function `makepop` computes the number of strategies associated with the core-based fractions, as per eq. (12.4 and fills the population array `pop` accordingly. Because of truncation happening when turning real-valued frequencies into integer-valued numbers of startegy, the total population size `np` sometimes deviates from the set target `NP`, but very seldom by more than ± 1 already for `NP`$= 100$. {`code:mixedIPD`}

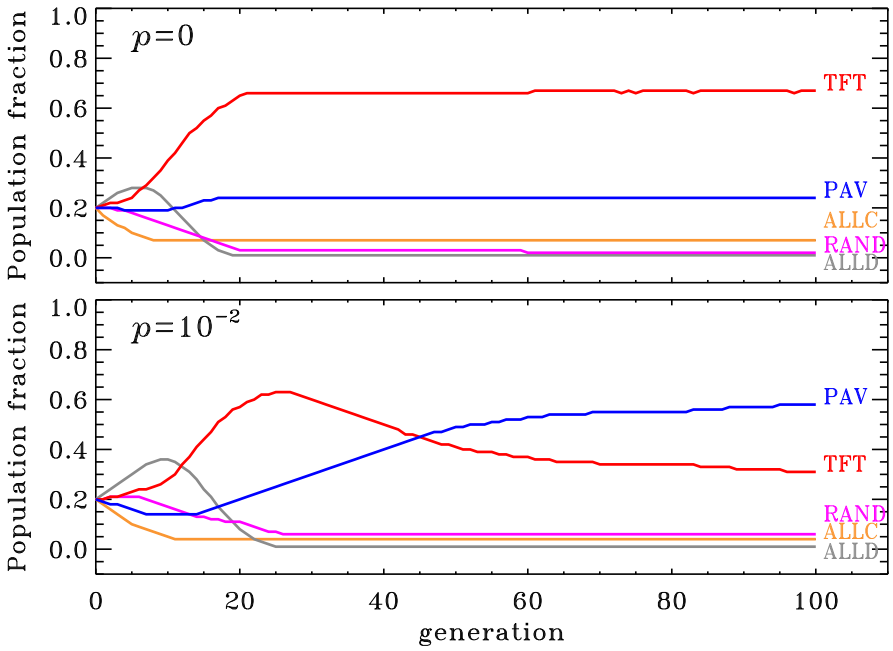

Figure 12.3: Variations of strategy frequencies with time, in two fully-mixed 100-round IPD simulations. In both cases a population of 100 players plays all other players in the population once at each temporal iteration. The top plot shows results for a noiseless reference simulation, while in the simulation of the bottom panel playing "mistakes" occur with probability $p = 10^{-2}$. {fig:ipd2ts}

taking over again from PAV. Increasing the noise level just a bit more, to $p \simeq 10^{-1}$, triggers yet another abrupt transition with ALLD now completely dominating the population. In a noisy environment ALLD does well playing against itself, because it scores 5 every time its ALLD opponent cooperates by mistake, and zero when it cooperates by mistake, for a mean score of 2.5 per mistake-ridden round, much better than its score of 1.0 in the absence of noise, and already as good as TFT playing against itself at very low noise levels. As p approaches a value of one half, all strategies start behaving effectively randomly, and converge again to $f_i = 0.2$.

In most of these fully-mixed IPD simulations, the frequencies of all strategies eventually stabilize, sometimes after a few tens of generational iterations, other times after a few hundreds. An interesting question is the stability (or lack thereof) of the global strategy frequencies. An *Evolutionary Stable Strategies* (ESS) is a strategy which, when dominating in given "environment", cannot be destabilized by externally-driven perturbations in the population frequencies, and can resist invasion by new or formerly extinct strategies introduced in the stabilized population[3]. One of the computational exercise at the end of this chapters leads you into an exploration of invasion and ESS in the fully-mixed IPD evolutionary model.

12.3 The spatially extended IPD {sec:spatialIPD}

From the point of view of population dynamics, for most living organisms the fully mixed model of the preceding section is ecologically and socially unrealistic. Even in our age of planet-wide communication and (relatively) easy plane travel, as a resident of Montréal I remain far more

[3]In the biological evolutionary context the primary such perturbations are the appearance of mutants within the existing population, or migration of new competing species into the local ecosystem.

likely to interact with an American from Vermont than with a Russian resident of Novosibirsk.

The spatial dimension can be introduced by placing strategies on the nodes of a lattice, and having them play only nearest-neighbour strategies. From one generational iteration to the next, each node simply adopts the strategy of its highest scoring neighbour, provided it exceeds its own score. This defines the *spatial IPD*, and the code listed on Fig. 12.4 illustrates one possible numerical implementation, based on 8 nearest-neighbours (top/down/right/left/diagonals). Note how the scores are first calculated for all nodes, and the choice of replacement strategies for each node is carried out subsequently in a second set of loops over the lattice dimensions. The lattice update is then carried out in a final set of loops. This is yet another instance of synchronous updating, necessary to avoid introducing a spatial evolutionary bias to the simulations.

This reproduction rule implies that a strategy entirely surrounded by its kin cannot change. As a consequence, once a compact group of any one strategy forms, it can only evolve at its perimeter. This turns out to be a key determinant of the evolution in these spatially-extended IPD simulations, as it can allow survival of a strategy in a hostile playing environment. This is illustrated on Figure 12.5, for the specific case of a group of ALLCs holding out despite being surrounded by ALLDs. The ALLC at the middle of the block scores $8 \times 3 = 24$ per round, while the best performing ALLD scores 20 per round ($3 \times 5 + 5 \times 1$). As consequence, the other 8 ALLCs will resist invasion by ALLD because their highest-scoring neighbour is always the central ALLC.

Figure 12.6 shows a simulation where a lattice populated by ALLC is contaminated with 5% of randomly distributed ALLDs. Within a few iterations ALLD has taken over much of the lattice, but unlike what would happen in the fully-mixed case, here ALLC manages to survive in invasion-resistant clusters, as on Fig. 12.5, jointly accounting for a $\simeq 10\%$ fraction of the population. Here the spatial pattern stops evolving after the tenth iteration, so that the bottom right panel on Fig. 12.6 representing the final, "frozen" state of the simulation[4].

Figure 12.7 shows the first five generational iteration of another spatial IPD noise-free simulation starting from a random distribution of equiprobable strategies, this time using all five strategies considered previously. This is the spatially-extended counterpart of the fully mixed simulation of Fig. 12.3 (top panel). In the first few generational iterations the evolution starts off as in the fully mixed model, with the ALLD population growing quickly at the expense of ALLC. While an isolated TFT loses to surrounding ALLD, a compact group of TFT can resist invasion by ALLD at its boundaries. This is because their small score deficit playing a neighbouring ALLD is more than offset by the 3-per-round score obtained playing against each other. The same is true of PAV, but to a lesser extent because it scores an average 0.5 per round against ALLD, half the score obtained by TFT against ALLD. As a consequence, after a few iterations TFT groups start expanding rapidly at the expense of ALLD, with a slower expansion of PAV groups. Compact groups of PAV and the few small remaining groupings of ALLC both stabilize when surrounded by TFT, as per our spatial reproduction rule. Already by the third iteration TFT dominates the lattice, with the few, rapidly shrinking clusters of ALLD managing to survive only on the perimeter of ALLC or RAND clusters, against which they secure scores equal to (against RAND) or higher (against ALLC) than TFT playing against itself. RAND is the big loser here, becoming extinct already at the tenth iteration, over two times faster than in the fully-mixed noise-free evolutionary simulations of §12.2.

The introduction of a finite mistake probability in the spatial IPD does not change the fact that compact groups of strategies can only evolve at their boundaries. But perturbations of these boundary interactions by noise leads to widely varying evolving spatial patterns and dominant strategies. This is illustrated on Figure 12.8, showing the spatial distribution of strategies on a 100×100 lattice after 250 generational iterations, for varying mistake probabil-

[4]For the payoff matrix (12.1), an ALLC playing 8 surrounding ALLCs scores 24 per round, while an ALLD with 4 ALLDs and 4 ALLCs as neighbours also scores 24 per round. This leads to an undesirable situation whereby the order in which the game is played with the 8 neighbours can influence the change (or lack thereof) in strategy at the node, which in turn can introduce a systematic spatial bias in the replacement of strategies. Throughout this section, this problem is summarily bypassed by using a payoff matrix entry $DD = 0.999$ instead of 1.0.

```
#include <stdio.h>
#include <stdlib.h>
#define NT 25      /* number of generational iterations */
#define N  100     /* lattice size */
#define PM 0.0     /* mistake probability */
int main(void)
{
/* Declarations/initialisations ------------------------------------------- */
  int ipd( int, int, float ) ;                  /* IPD function (Fig 2.1)     */
  void periodic( float[N+2][N+2] ) ;
  int iter, j, k, jb, kb, ip ;
  float pop[N+2][N+2], fit[N+2][N+2], popnew[N+2][N+2], score, best, prob ;
  int dx[8]={-1,0,1,1,1,0,-1} ; dy[8]={-1,-1,-1,0,1,1,1,0} ;
/* Executable ------------------------------------------------------------- */
  for ( j=1 ; j<N+1 ; j++ ) {                   /* random initial condition   */
      for ( k=1 ; k<N+1 ; k++ ) { pop[j][k]=floor(5*rand()/RAND_MAX) ; } }
  for ( iter=0 ; iter<NT ; iter++ ) {           /* generational iteration     */
    for ( j=1 ; j<N+1 ; j++ ) {                 /* first loops over lattice   */
      for ( k=1 ; k<N+1 ; k++ ) {
        fit[j][k]=0. ;
        for ( ip=0 ; ip<8 ; ip++ ) {            /* Loop over neighbours       */
          fit[j][k]+=ipd(pop[j][k],pop[j+dx[ip]][k+dy[ip]],PM) ; } /* Play IPD*/
      }
    }                                           /* end first lattice loops    */
    periodic( fit ) ;                           /* enforce periodicity in x,y */
    for ( j=1 ; j<N+1 ; j++ ) {                 /* second lattice loops       */
      for ( k=1 ; k<N+1 ; k++ ) {
        jb=j ; kb=k ; best=fit[j][k] ;          /* current best: self!        */
        for ( ip=0 ; ip<8 ; ip++ ) {            /* Loop over neighbours       */
          if ( fit[j+dx[ip]][k+dy[ip]] > best) {/* found a new best           */
             jb=j+dx(ip) ; kb=k+dy(ip) ; best=fit[jb][jk] ; } }
        popnew[j][k]=pop[jb][kb] ;              /* adopt strategy of best     */
      }
    }                                           /* End second lattice loops   */
    for ( j=1 ; j<N+1 ; j++ ) {                 /* replace population         */
      for ( k=1 ; k<N+1 ; k++ ) { pop[j][k]=popnew[j][k] ; } }
    periodic( pop ) ;                           /* enforce periodicity in x,y */
  }                                             /* End generational loop      */
}
void periodic( float latt[][] )                 /* Enforces periodic B.Cs.    */
{
  int j ;
  for ( j=1 ; j<N+1 ; j++ ) {
    latt[j][0]=latt[j][N] ; latt[j][N+1]=latt[j][1] ;     /* periodic in x    */
    latt[0][j]=latt[N][j] ; latt[N+1][j]=latt[1][j] ;     /* periodic in y    */
  }
  latt[0][0]    =latt[N][N] ; latt[N+1][N+1]=latt[1][1] ; /* the 4 corners    */
  latt[0][N+1]  =latt[N][1] ; latt[N+1][0]  =latt[1][N] ;
}
```

Figure 12.4: Source code in the C-programming language for the spatially-extended IPD. This code uses the same IPD C-function (listed on Fig. 12.1) as the fully-mixed model of Fig. 12.2. {code:spatialIPD}

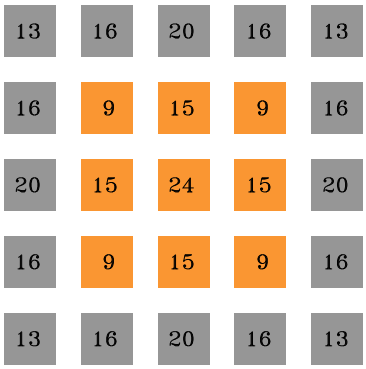

Figure 12.5: Survival of a 3×3 block of ALLC (orange) surrounded by ALLDs (gray). The score of each strategy playing against its 8 neighbours is indicated. The outer layer of ALLD is assumed to be surrounded by even more ALLD. These scores are based on the payoff matrix (12.1). {fig:gangup}

ities p, as labeled. All simulations begin from the same random initial conditions and use the same sequence of random numbers when playing RAND, so that the differences truly reflect the system's evolution, rather than idiosyncracies of the initial conditions or of RAND's behavior. The $p = 0$ snapshots represents the spatially frozen end state of the simulation of Fig. 12.7, persisting unchanged since the eleventh iteration for this simulation.

Already at $p = 10^{-3}$, where mistakes happen on average once every 10 games, ALLC benefits from TFT's inability to correct mistakes when playing against itself; in a 100-round game, its score can range from 1.02 to 3.02, depending on when the mistake happens. ALLC being blindly forgiving, at low noise levels it does quite well playing against itself; if each player makes only one mistake in a 100-round game, they each collect a score of 2.99 instead of 3.0. This is even better than PAV playing against itself (2.94 instead of 3.0). In such an environment, an apparently suicidal strategy like ALLC ends up doing well when it can cluster; there can indeed be safety in numbers! The larger population of ALLC also allows survival of ALLD at much higher levels ($f \simeq 0.15$) than in the absence of noise ($f = 1.3 \times 10^{-4}$).

numbers

At high noise levels ($p > 3 \times 10^{-2}$) PAV's error-correction capabilities give it an unbeatable edge over TFT. At $p = 0.1$ (on average ten mistakes per player in a 100-round game), the simulation is dominated by PAV ($f = 0.84$) already after 100 generational iterations, with remaining isolated compact blocks of ALLC each surrounded by a thin shell of parasitic ALLD. ALLD also survives in linear structures cutting paths through the dominant PAV background. TFT, extinguished here after 30 iteration, cannot compete with PAV at this noise level, due to its inability to correct mistakes. The spatial pattern is still evolving slowly after 250 iterations, with PAV occasionally taking over a node occupied by an ALLD following a favorable sequence of playing mistakes.

At intermediate noise levels ($p = 10^{-2}$), RAND and PAV again become extinct, after 50 and 130 iterations respectively, leaving an approximately equal mixture of ALLC, ALLD et TFT. Their spatial distribution (bottom left panel on Fig. 12.7 roughly resembles that characterizing lower noise levels (rop right panel), but the evolution shows a new and interesting temporal behavior, namely a three-species predator-prey cycle between ALLC, ALLD, and TFT. This is depicted on Figure 12.9, showing a 50 generation-long segment of the frequency time series in the $p = 10^{-2}$ simulation of Fig. 12.7. The cycle is intimately tied to the spatial distribution of the three strategies. ALLD cannot invade compact clusters of TFT, but it can eat into the layers of ALLC, until it hits the underlying "core" of TFT. At this point the growth of ALLD ceases, and soon TFT rises again, working jointly with ALLC to recover ground lost to ALLD. Because ALLC is more noise tolerant than TFT, it eventually gains ground against TFT as

Prey

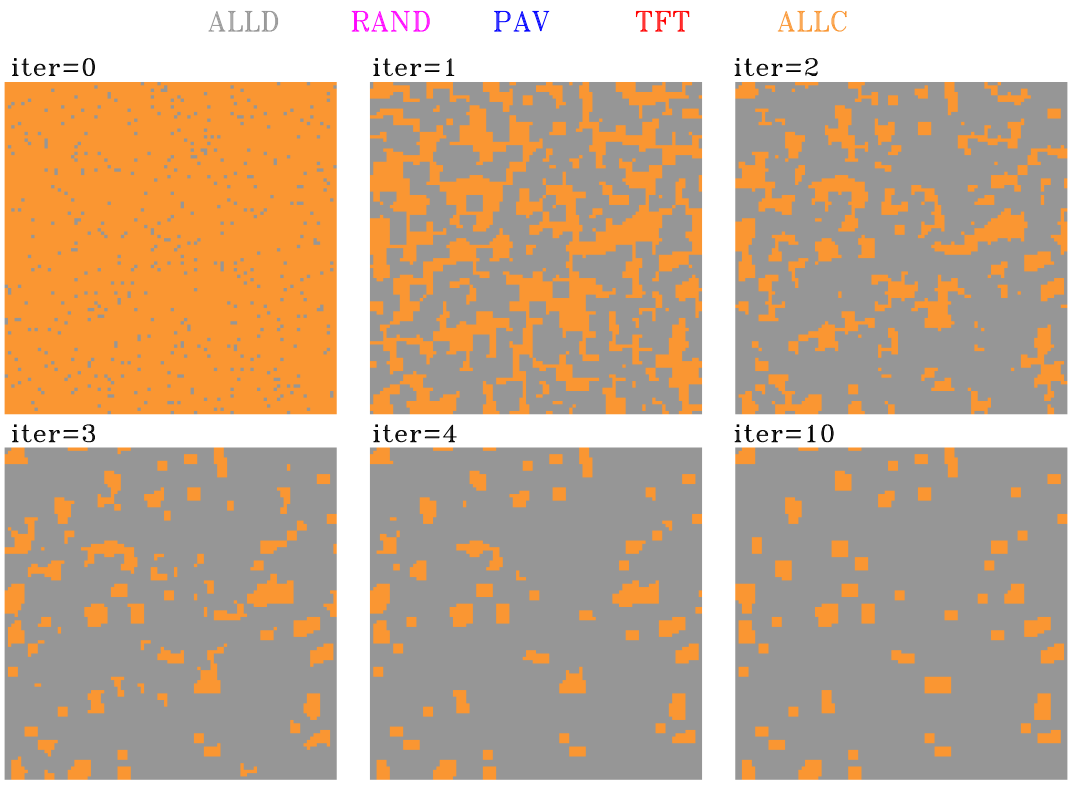

Figure 12.6: Invasion of an ALLC population (orange) by a small number of randomly distributed ALLD (gray). ALLD dominates the lattice within a few iterations, but small clusters of ALLC manage to resist invasion and persist into the final, frozen state of the simulation (bottom right). {fig:invasion}

well, with the latter starting to decline while the former keeps rising. The stage is now set for the onset of the next cycle.

Recurrence cycle

12.4 To cooperate or to defect ?

Throughout this chapter we only "tested" a very small set of five strategies, but we can nonetheless extract from our results some valid general conclusions: blind cooperation (ALLC) is not a good strategy; neither is systematic defection (ALLD), except if an unlimited supply of suckers (ALLC) is available for repeated abuse, an unlikely situation in any realistic evolutionary context. TFT represents a form of compromise that works extremely well as long as the player's actions are "rational", in the sense of not being subjected to too many random mistakes. But surely strategies even better than TFT must exist ?

Danger: of blind cooper

In the late 1970s Robert Axelrod organized an IPD tournament in which participants could enter a computer program encoding their favorite strategy. Axelrod received 15 valid entries, including both deterministic and probabilistic strategies, some quite simple and other very intricate. In this tournament, the simplest program won: it was TFT (also the shortest entry, in terms of lines of code). A second tournament held a few years afterwards attracted over 60 competing strategies, many designed with the knowledge that TFT had won the first tournament. TFT won again! As far as I know, no strategy has yet been found that can manage to

Axelrod, Robert
IPD: tournament

Tit-For-Tat

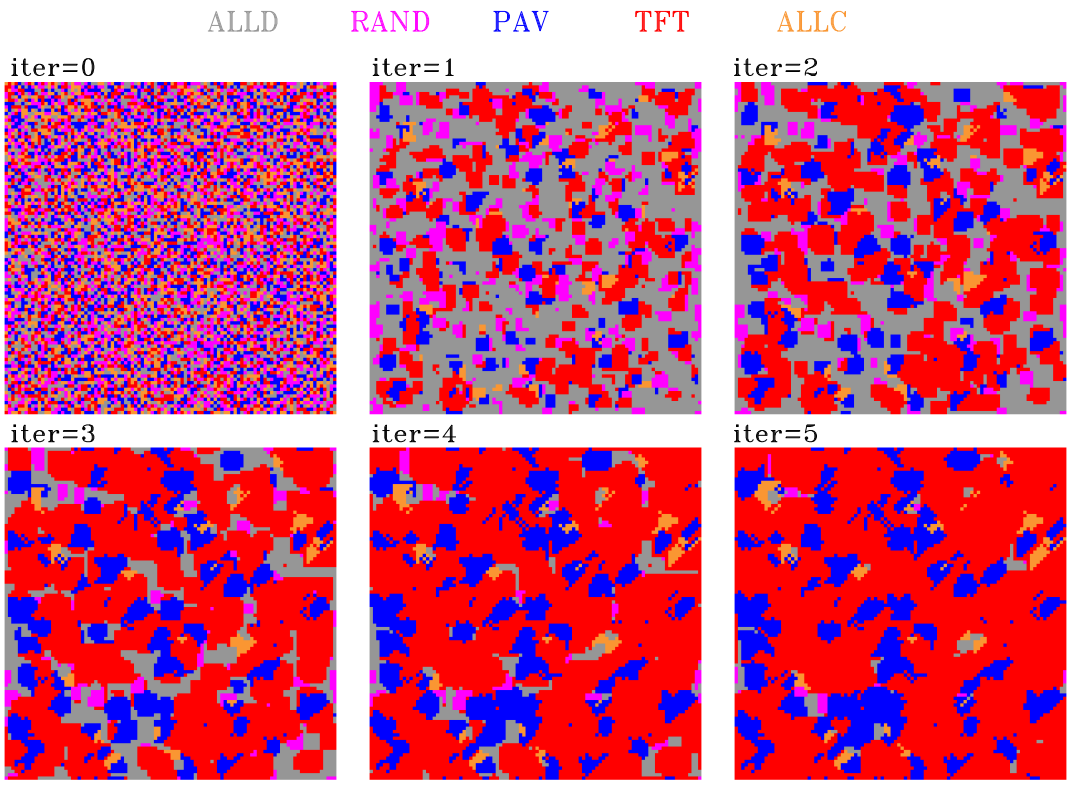

Figure 12.7: Evolving spatial distribution of strategies in the spatial version of the IPD, using 8 nearest-neighbour tournament. The initial condition (top left) is a random mixture of strategies. For the specific initial condition used here, by the eleventh iteration the evolution has ceased and the spatial pattern remains frozen thereafter. {fig:6snapshots}

beat TFT in such a noise-free, multi-strategy tournament environment[5].

It is interesting to reflect upon the fact that as an ethical guide for social interations, TFT embodies only three simple behavioral principles: (1) Start off nice; (2) retaliate ruthlessly; (3) don't hold a grudge. Axelrod's tournaments, and in a more limited way this chapter, demonstrate that there is empirical support —if not grandeur— in this view of life.

12.5 Exercices and further computational explorations

1. Using the fully-mixed model of §12.2, introduce a fraction f of TFT in an ALLD population; at what value of f can TFT invade ALLD ? Repeat the exercise, this time with PAV trying to invade ALLD.

2. Use the IPD function of Fig. 12.1 to compute score matrices equivalent to eq. (12.2), but with noise probabilities 10^{-3}, 10^{-2} and 10^{-1}. Can you infer from these which strategy is likely to dominate as the noise level varies, in the context of a fully-mixed model ? And, can you now figure out why at moderate noise levels, "lines" of ALLD can survive within a PAV-dominated environment (see Fig. 12.8, bottom right panel).

[5] A modified version of TFT which systematically defects on the very last round of each game would clearly gain a small edge over classical TFT; however, beating TFT in this manner is only possible if the number of rounds to be played is known a priori.

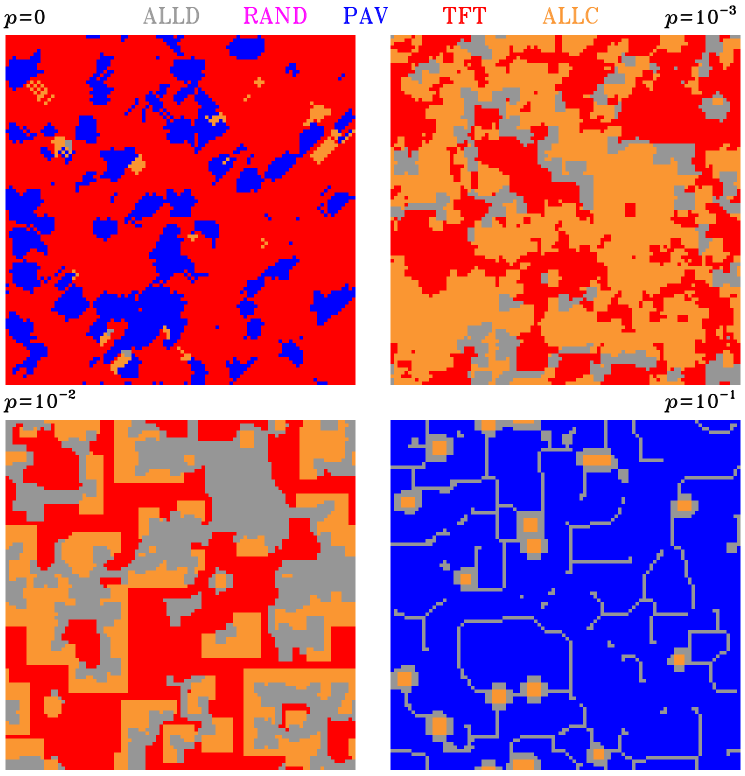

Figure 12.8: Similar in format to Fig. 8.6, but showing now the distribution of strategies after 250 generational iterations, in a series of spatial IPD simulations with increasing mistake probability p, as labeled. All four simulations started from the same random initial distribution of strategies. The top right panel is the final "frozen" state of the noise-free simulation of Fig. 12.7. {fig:4snapshots-noise}

3. Consider the following two new strategies: GRIM is an unforgiving version of ALLC, in that it starts off nice (move C), but as soon as facing defection (D) from an opponent it switches to D to the end of the game, no matter what the opponent does afterwards. "Generous Tit-for-Tat" (GTFT) operates like TFT, except that it can play a move C with some probability p (= 0.3, say) even if the opponent's previous move was D, which can serve as an error correction mechanism. Set up a fully-mixed evolutionary simulations using these two new strategies in addition to the five considered in this chapter, and examine which reach dominance under various noise conditions (including no noise). IPD: strategies

4. Construct diagrams similar to Figure 12.5 and determine (1) what is the smallest-size block of TFT or PAV than can survive in an ALLD environment; (2) what is the smallest-size block of ALLD that can survive in either an TFT or PAV-dominated environment.

5. Introducing noise in the IPD can be viewed as changing the values of the score matrix's entries (if you're not convinced, do exercise #2 above!). Another way to change the score matrix is to change the payoff matrix (12.1). Consider a payoff matrix of the form: IPD: payoff matrix

$$\begin{array}{c} \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cc} C & D \end{array} \\ \begin{pmatrix} 1 & 0 \\ b & 10^{-3} \end{pmatrix} \end{array}$$

Run noise-free spatial IPD simulations starting with an equal mixture of randomly distributed ALLC and ALLD, using different values of b in the range $1.1 \leq b \leq 1.7$. Reflect

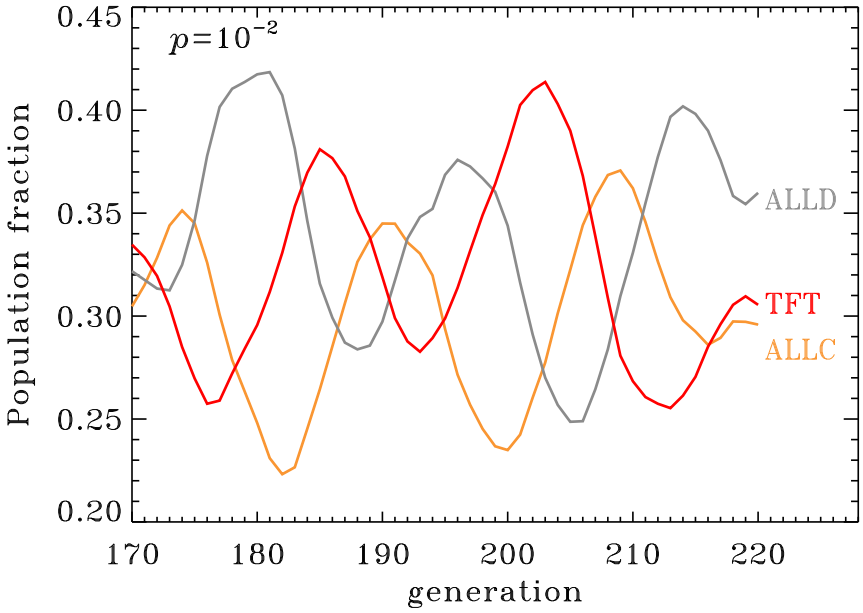

Figure 12.9: Three-species Predator-Prey cycles in a spatial IPD simulation with mistake probability $p = 10^{-2}$ (bottom left panel on Fig. 12.8). {fig:ts-cycle}

upon the wide variety and complexity of the spatiotemporal evolutionary patterns produced... and on their resemblance to some of the patterns produced by the hodgepodge machine back in the preceding chapter... as well as way back in chapter 2!

6. And now for the Grand Challenge: adding a random walk to the spatial IPD. At each generational iteration, each player makes a random walk move to one of its four immediate nearest-neighbour (top/down/right/left), and then plays a 100-round IPD game with any other player located within a 3×3 stencil to compute its own score. At the next generational iteration, the player simply adopts the highest-ranking strategy among those played against, just as in the standard spatial IPD. You may of course allow two or more walkers to occupy the same node (if needed, see the epidemic propagation code of Fig. 9.1 to get started). Start with a randomly distributed equal mixture of ALLC and ALLD, and then consider a random mixture of the five strategies considered in this chapter (and add GRIM and GTFT from exercise #3, if you want to go all the way...). Is mobility favoring or hindering the dominance of some strategies over others ?

12.6 Further readings

This chapter is strongly inspired by

Flake, G.W., *The computational beauty of Nature*, MIT Press, chap. 17 (1998),
Nowak, M.A., *Evolutionary Dynamics*, Cambridge: Harvard University Press, chap. 9 (2006).

An absolute must-read in game theory remains

ory
Robert

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

At a more technical/mathematical level, and also covering a broader range of games, see:

Gintis, H., *Game Theory Evolving*, 2nd ed., Princeton: Princeton University Press (2009).

The aforecited book by Nowak also offers an excellent introduction to game theory, including engaging and accessible analyses of stability and equilibria in populations of strategies. If §12.3 got you intrigued regarding the game theoretic approach to evolution and population dynamics, see also:

Hofbauer, J., & Sigmund, K., *Evolutionary Game and Population Dynamics*, Cambridge: Cambridge University Press (1998)

Chapter 13

Evolution

{chap:evolve}

All the complex systems we encountered so far in this book relied on the repeated action of *rules*, which led to interesting emergent behavior; but in all cases these rules were designed *a priori* and hardwired into the system. In some cases, such as avalanches (chapter 5) or earthquakes (chapter 8), the origin of these rules could be traced back more of less straightforwardly to basic physical laws, but in other cases not. Some aspects of driving behavior (chapter 7) and animal flocking (chapter 10) are certainly, at some level, constrained by fundamental physical laws, but the rules implemented in these models also incorporated some sort of "intelligent design". The IPD strategies encountered in the preceding chapter are a good case in point: Tit-For-Tat was *designed* by someone[1], who had a definite purpose in mind: winning Axelrod's tournament! But what if there is no such wise Grand Architect around to design strategies? Who sets the rules then ?

Rapoport, Anatol

Biologists have known for well over a century that evolution by means of natural selection breeds complexity. In this final chapter we evolve IPD strategies through the (un)holy trinity of evolutionary biology: selection, inheritance, and variation[2]. This will first require the introduction of another class of biologically-inpired agents: neural networks.

Agents: as neural netwo

13.1 The IPD player as a neural network

Some strategies introduced in the preceding chapter, such as ALLD and ALLC, are non-reactive, in that they are not influenced by prior moves. TFT, on the other hand, is a reactive strategy, as it is based on (and only on) the opponent's prior move. More general reactive strategies could take into account more information regarding prior moves by the opponent as well as the player itself. To be more specific, let us write player one's move at round n as some function (f_1, say) of the array $\mathbf{H}$ of its previous moves and those of opponent player two:

$$P_1^n = f_1(\mathbf{H}) \ , \qquad \mathbf{H} = [P_1^{n-1}, P_1^{n-2}, ..., P_2^{n-1}, P_2^{n-2}, ...] \ . \tag{13.1}$$

{???}

with the various P_1, P_2, etc taking on the two possible values "D" or "C", as before, and the function f_1 returning one of these two values as output. For example, if player 1 plays the award-winning TFT strategy, f_1 depends only on P_2^{n-1}; in fact, $f_1^{\rm TFT} = P_2^{n-1}$. For non-reactive strategies such as ALLC and ALLD, f_1 does not even depend on $\mathbf{H}$: $f_1^{\rm ALLD} = D$ and $f_1^{\rm ALLC} = C$.

A strategy can be generally defined as the association of a specific move, D or C, to every possible instance of $\mathbf{H}$. An enumerative approach (based, e.g., on a lookup table) becomes rapidly impractical as soon as $\mathbf{H}$ contains more than a few elements; likewise experts systems

[1] Anatol Rapoport, at the time a professor of Mathematics and Psychology at the University of Toronto.

[2] This is the only chapter in this book which is not self-contained, as it builds substantially on the previous one. Unless familiarity with game theory and the prisoner's dilemma is already in hand, readers in an evolutionary rush need to read at least §12.1.

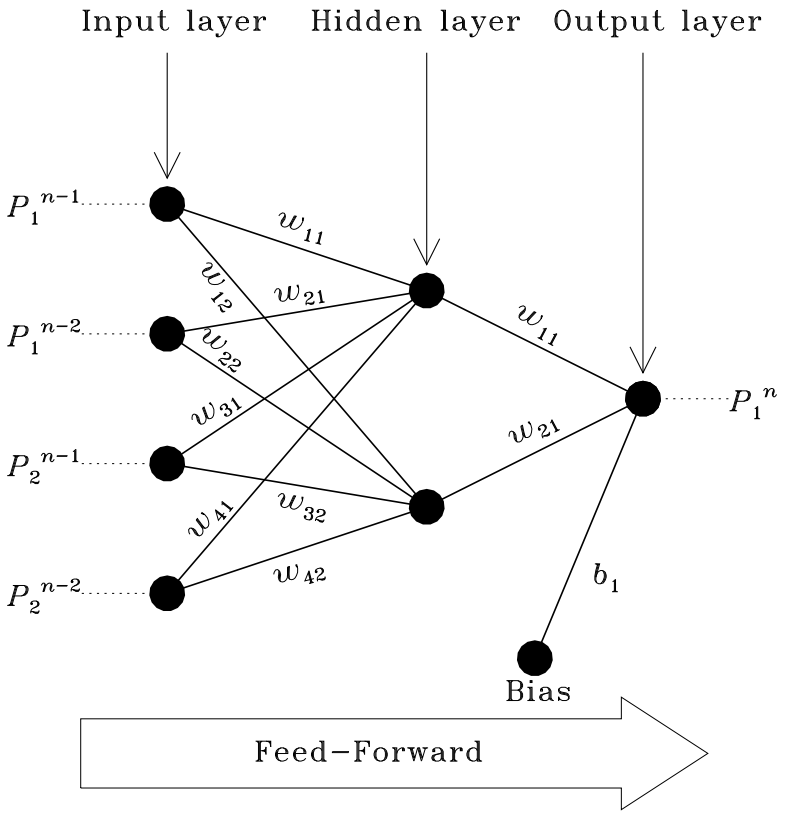

Figure 13.1: A simple "feed-forward" artificial neural network. This network has four input units, two hidden layer units, and a single output unit subjected to a bias. The input vector consists of the prior two moves of the player (P_1^{n-1}, P_1^{n-2}) and its opponent (P_2^{n-1}, P_2^{n-2}). The output is the move P_1^n of the player at the current round n. Information flows from left to right in such a network, and the hidden and output units process their incoming signals according to eqs. (13.4)—(13.5). The superscripts IH and OH have been omitted on the connection weights to avoid overcrowding the diagram. {fig:nndemo}

(based, e.g., on a slew of **if**...**else** logical statements) rapidly become a coding nightmare unless **H** is small. *Artificial Neural Networks* offer an interesting and computationally efficient approach when **H** does not contain an extremely large number of elements.

Neural networks are made up of a set of interconnected computing units, each processing input signals received from some (or all) other units, and broadcasting the result back to some (or all) of the other units. Here we consider a (relatively) simple network which is designed to process a signal entering the network through *input units*, and producing an output signal. Figure 13.1 illustrates the basic network architecture. Here the four input units each transmit to the rest of the network one of the prior two moves of the player or its opponent, converting them to numerical values according to the convention:

{eq:inmove}
$$S_1^1 = \begin{cases} -1 & \text{if } P_1^{n-1} = D \\ +1 & \text{if } P_1^{n-1} = C \end{cases}, \qquad S_1^2 = \begin{cases} -1 & \text{if } P_1^{n-2} = D \\ +1 & \text{if } P_1^{n-1} = C \end{cases}, \tag{13.2}$$

{???}
$$S_1^3 = \begin{cases} -1 & \text{if } P_2^{n-1} = D \\ +1 & \text{if } P_2^{n-2} = C \end{cases}, \qquad S_1^4 = \begin{cases} -1 & \text{if } P_2^{n-2} = D \\ +1 & \text{if } P_2^{n-2} = C \end{cases}. \tag{13.3}$$

These inputs are transmitted to the *hidden layer* units, which first compute their linear combi-

nations a^H, weighted by the network's corresponding connection weights, the result being fed into a nonlinear *activation function* φ^H which then returns the signals S^H which represent the output of the two hidden layer units:

$$S_j^H = \varphi(a_j^H) \ , \qquad a_j^H = \sum_{i=1}^{4} w_{ij}^{IH} S_i^I \ , \qquad j = 1, 2 \ . \tag{13.4}$$ {eq:nnwh}

The same computation (with one addition, to be discussed shortly) is now repeated, this time from the hidden layer to the output layer, here comprised of a single unit:

$$S_1^O = \varphi(a_1^O) \ , \qquad a_1^O = b_1^O + \sum_{i=1}^{2} w_{i1}^{HO} S_i^H \ . \tag{13.5}$$ {eq:nnwo}

The last step is to convert the signal of the output unit into an IPD move. One simple possibility is

$$P_1^n = \begin{cases} D & \text{if } S_1^O < 0 \\ C & \text{if } S_1^O > 0 \end{cases} \ . \tag{13.6}$$ {eq:outmove}

The role of the activation function is to restrict the numerical output of the computational units to some preset range, here set to $[-1, +1]$. This makes it more difficult for a single neural path (from input layer to output layer) to dominate the behavior of the network, A variety of simple mathematical functions can do the job, for example the logistic function:

Neural network: activation

$$\varphi(a) = 2\left(\frac{1}{1+\exp(-2a)} - \frac{1}{2}\right) \ . \tag{13.7}$$ {eq:activation}

In such networks the flow of information is unidirectional, from the input layer to the output layer, via the hidden layers. Such an architecture is called a *feed-forward* network (or *perceptron*). The hidden layer may appear superfluous, but it plays in fact a crucial role, as its units act as feature detectors allowing parallel processing of the signal patterns transmitted by the input units[3].

Note, on Figure 13.1, how the output unit receives signals from the two units of the hidden later, as well as from a "bias" unit whose input is always +1. Such a bias allows this neural network to encode non-reactive strategies such as ALLD and ALLC, which are not influenced by prior moves. For example, if the two connection weights between the hidden layer and output layer are very small in magnitude, then the move output is determined by the sign of the bias connection weight b_1^O:

- **ALLC**: $b_1^O > 0$, $|b_1^O| \gg |w_{11}^{HO}|, |w_{12}^{HO}|$.
- **ALLD**: $b_1^O < 0$, $|b_1^O| \gg |w_{11}^{HO}|, |w_{12}^{HO}|$.

In what follows, the network connection weights are restricted to the range $-2 \leq w^{IH}, w^{HO} \leq +2$, while the bias weight will never exceed the range $-1 \leq b_1^O \leq +1$.

How could TFT be encoded by such a neural network? TFT cares only about the opponent's prior move (P_2^{n-1} on Fig. **??**), so all "IH" nodes other than w_{31}^{IH} and w_{32}^{IH} can be $\simeq 0$. Keeping one eye on Fig. 13.1 and the other on eqs. (13.4)–(13.5), you should convince yourself that the following two distinct neural net configurations both yield TFT behavior:

- **TFT**: $b_1^O \simeq 0$, $w_{31}^{IH} > 0$, $w_{11}^{HO} > 0$, all other w's $\simeq 0$.
- **TFT**: $b_1^O \simeq 0$, $w_{32}^{IH} < 0$, $w_{21}^{HO} < 0$, all other w's $\simeq 0$.

Many more such TFT configurations are of course possible; this *redundancy* is in fact a hallmark

Neural network: redund

[3]The fact that the hidden layer is comprised here of only two units is not essential; two is often deemed minimal, and more than the number of input units would be unusual for the type of application considered here. Likewise, feed-forward neural networks can include more than one hidden layer. A few introductory textbooks on neural networks are listed in the bibliography at the end of this chapter, for the benefit of those interested in learning more about these fascinating information processing systems, as well as their biological inspiration.

of neural network computation.

With an input vector $\mathbf{H} \equiv (P_1^{n-1}, P_1^{n-2}, P_2^{n-1}, P_1^{n-2})$ containing 4 entries, $2^4 = 16$ distinct binary 4-element inputs to our NN are possible, each yielding a move D or C; there are consequently $2^{16} = 65536$ distinct move vectors possible, each defining a distinct "stragegy" played by the neural net; schematically we can express this as:

{eq:NN16moves}

$$\begin{array}{cccc} P_1^{n-1} & P_1^{n-2} & P_2^{n-1} & P_2^{n-2} \\ \left(\begin{array}{c} D \\ D \\ D \\ D \\ D \\ D \\ D \\ D \\ C \\ C \\ C \\ C \\ C \\ C \\ C \\ C \end{array}\right. & \left(\begin{array}{c} D \\ D \\ D \\ D \\ C \\ C \\ C \\ C \\ D \\ D \\ D \\ D \\ C \\ C \\ C \\ C \end{array}\right. & \left(\begin{array}{c} D \\ D \\ C \\ C \\ D \\ D \\ C \\ C \\ D \\ D \\ C \\ C \\ D \\ D \\ C \\ C \end{array}\right. & \left(\begin{array}{c} D \\ C \\ D \\ C \\ D \\ C \\ D \\ C \\ D \\ C \\ D \\ C \\ D \\ C \\ D \\ C \end{array}\right) \end{array} \rightarrow \text{NN} \rightarrow \left(\begin{array}{c} D \\ D \\ D \\ D \\ D \\ D \\ D \\ D \\ D \\ D \\ D \\ D \\ D \\ D \\ D \\ D \end{array}\right) \left(\begin{array}{c} C \\ D \\ D \\ D \\ D \\ D \\ D \\ D \\ D \\ D \\ D \\ D \\ D \\ D \\ D \\ D \end{array}\right) \left(\begin{array}{c} D \\ C \\ D \\ D \\ D \\ D \\ D \\ D \\ D \\ D \\ D \\ D \\ D \\ D \\ D \\ D \end{array}\right) \ldots \left(\begin{array}{c} C \\ C \\ C \\ C \\ C \\ C \\ C \\ C \\ C \\ C \\ C \\ C \\ C \\ C \\ C \\ D \end{array}\right) \left(\begin{array}{c} C \\ C \\ C \\ C \\ C \\ C \\ C \\ C \\ C \\ C \\ C \\ C \\ C \\ C \\ C \\ C \end{array}\right) \quad (13.8)$$

where the "..." stands for the missing 65531 move vectors! The leftmost move vector is our familiar ALLD strategy, while the rightmost is ALLC. The following is a list of strategies to be encountered in what follows, some familiar and some less so:

{eq:NNmovevec}

$$\text{TFT} \equiv \left(\begin{array}{c} D \\ D \\ C \\ C \\ D \\ D \\ C \\ C \\ D \\ D \\ C \\ C \\ D \\ D \\ C \\ C \end{array}\right) \quad \text{DTFT} \equiv \left(\begin{array}{c} D \\ C \\ D \\ C \\ D \\ C \\ D \\ C \\ D \\ C \\ D \\ C \\ D \\ C \\ D \\ C \end{array}\right) \quad \text{ATFT} \equiv \left(\begin{array}{c} C \\ C \\ D \\ D \\ C \\ C \\ D \\ D \\ C \\ C \\ D \\ D \\ C \\ C \\ D \\ D \end{array}\right) \quad \text{REP1} \equiv \left(\begin{array}{c} D \\ D \\ D \\ D \\ D \\ D \\ D \\ D \\ C \\ C \\ C \\ C \\ C \\ C \\ C \\ C \end{array}\right) \quad \text{TFTT} \equiv \left(\begin{array}{c} D \\ C \\ C \\ C \\ D \\ C \\ C \\ C \\ D \\ C \\ C \\ C \\ C \\ D \\ C \\ C \end{array}\right) \quad (13.9)$$

DTFT (Delayed TFT) operates as TFT, but replicates the opponent's move not from the last round, but rather from two rounds ago (P_2^{n-2}); ATDT (Anti-TFT) systematically plays the move opposite to the opponent's prior move; TFTT (Tit-For-Two-Tats) is a forgiving version of TFT, defecting only after two successive defection from the opponent. REP1 is a strategy that simply repeats its prior move, independent of the opponent's moves; if starting with D, REP1 will thus behave like ALLD, but starting with C it will behave like ALLC instead, the playing behavior being entirely controlled by the player's first (pre-wired) move. REP2 operates similarly, repeating the player's move from two rounds ago[4].

tegies

[4] A much more compact way to express strategies can be defined upon introducing a "don't care" symbol

In all cases, here the move "chosen" by the neural net remains a completely deterministic function of its inputs. This means that a stochastic strategy such as RAND (see §12.1) cannot be encoded by the type of neural network considered here.

13.2 Numerical Implementation

We now turn to evolutionary simulations conceptually equivalent to the fully-mixed population model encountered in §12.2. We consider a population of 100 agents (i.e., neural networks), each playing a 10-round IPD with a randomly selected 20-member subset of the population. The best-scoring agent then replaces a randomly-selected subset of 4 population members. This sequence defines a generational iteration, at the end of which the contest begins anew within the new agent population, which only contains 4 new members. The code listed on Figure 13.2 offers one straightforward implementation, which requires the four C-functions listed in Figure 13.3. Note the following:

1. From a practical point of view, running the neural network only involves the calculations described by eqs. (13.4)—(13.5). The complete "definition" of an agent thus includes a set of 11 connection weights, as well as the set of two prewired moves required to get the IPD going. Those are stored in the two-dimensional population array `w[NP][13]`, the j^{th} line the defining parameters for agent j, organized in the order:

$$\mathbf{w}_j \equiv (w_{11}^{IH}, w_{21}^{IH}, w_{31}^{IH}, w_{41}^{IH}, w_{12}^{IH}, w_{22}^{IH}, w_{32}^{IH}, w_{42}^{IH}, w_{11}^{HO}, w_{21}^{HO}, b_1^O, P^{-1}, P^{-2}) \ , \quad (13.10) \quad \texttt{\{eq:wvector\}}$$

2. The declaration for the population array `w` stands *outside* the main program, making it a *global variable* accessible to the program and all functions without having to pass it as an argument.

3. The evolutionary process is initiated by assigning random values to the connection weights defining each NN in the population. In all simulations considered here the weights are uniformly distributed in the interval $[-2, +2]$, except for the bias weight (b_1^O) which is restricted to $[-1, +1]$.

4. More

5. The neural network function `nn` requires two arguments, the first being a 4-element vector `priorm` containing the prior moves of the player and its opponent, and the second an index tag identifying the location of the playing agent in the population array `w`. The function returns the move played for a single round of the IPD.

6. The output move omits the calculation of $\varphi(a)$, since the outcome depends only on the sign of the weighted sum a_1^O and eq. (13.7) is sign-preserving.

13.3 Some representative simulations

Figure 13.4 shows the outcome of four evolutionary simulations, in the form of time series for the average score-per-move of the best (black) and worst (gray) player at each generational

(traditionally, #), which means that the corresponding position in the input vector has no influence on the choice of move. Under this notation the five strategies just described could be written as

- **TFT:** $[\#, \#, D, \#] \to D$, $[\#, \#, C, \#] \to C$.
- **DTFT:** $[\#, \#, \#, D] \to D$, $[\#, \#, \#, C] \to C$.
- **ATFT:** $[\#, \#, D, \#] \to C$, $[\#, \#, C, \#] \to D$.
- **REP1:** $[C, \#, \#, \#] \to C$, $[D, \#, \#, \#] \to D$.
- **REP2:** $[\#, C, \#, \#] \to C$, $[\#, D, \#, \#] \to D$.

with ALLC defined simply as $[\#, \#, \#, \#] \to C$, and ALLD as $[\#, \#, \#, \#] \to D$.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#DEFINE NG 250              /* number of generational iterations */
#DEFINE NP 100              /* number of players (population size) */
#DEFINE NOPP 20             /* number of opponent for each player */
#DEFINE PM 0.0              /* mutation probability */
#DEFINE NR 4                /* number of agents replaced at each generation */
float w[NP,13] ;            /* population connection weights and first moves */
int main(void)
{
/* Declarations/initialisations -----------------x------------------------- */
  int nnipd( float[4], int ) ;                   /* neural network function  */
  int findbest( float[NP] ) ;                    /* max funtion for array    */
  void replaceandmutate ( int ) ;                /* generational replacement */
  int i, ip, io, iopp, ipd, move1, move2, i, ibest ;
  float pm1[2], pm2[2], score[NP] ;
  float payoff[2][2]={{3.,0.},{5.,1.}} ;        /* payoff matrix            */
/* Executable ------------------------------------------------------------- */
  for (ip=0 ; ip<NP ; ip++) {                 /* initialize population       */
    for (i=0 ; i<11; i++) { w[ip,i]=-2.+4.*rand()/RAND_MAX ; }    /* weights */
    w[ip,11]=-1.+2.*rand()/RAND_MAX ;                             /* bias    */
    for (i=0 ; i<1; i++) { w[ip,12+i]=floor(2.*rand()/RAND_MAX) ; } /* moves */
  }
  for (ig=0 ; ig<NG ; ig++) {                  /* loop over generations      */
    for (ip=0 ; ip<NP ; ip++) {                 /* loop over population      */
      score[ip]=0.
      for (iopp=0 ; iopp<NOPP ; iopp++) {       /* loop over opponents       */
        io=floor(np*1.*rand()/RAND_MAX)         /* pick an opponent          */
        pm1[0]=w[ig][12]; pm1[1]=w[ig][13]; pm1[2]=w[io][12]; pm1[3]=w[io][13];
        pm2[0]=w[io][12]; pm2[1]=w[io][13]; pm2[2]=w[ig][12]; pm2[3]=w[ig][13];
        score[ip]+=payoff[pm1[0]][pm2[0]]+payoff[pm1[1]][pm2[1]] ;
        for ( ipd=0 ; ipd<10 ; ipd++) {         /* play 10-round IPD         */
          move1=nnipd(prmv1,ip) ;
          move2=nnipd(prmv2,io) ;
          score[ip]+=payoff[move1][move2] ;         /* player's score        */
          pm1[1]=pm1[0] ; pm1[0]=move1 ; pm1[3]=pm1[2] ; pm1[2]=move2 ;
          pm2[1]=pm2[0] ; pm2[0]=move2 ; pm2[3]=pm2[2] ; pm2[2]=move1 ;
        }                                       /* end of this IPD game      */
      }                                         /* end of opponent loop      */
    }                                           /* end of population loop    */
    ibest=findbest( score ) ;                   /* find best scoring agent   */
    replaceandmutate ( ibest ) ;                /* alter population          */
    printf("best score-per-move %f at gen. %d\n",ip,score[ip]/10./NOPP,ig) ;
  }                                             /* end of generation loop    */
}
```

Figure 13.2: Source code for an evolutionary simulation in which each agent in the population plays a 10-round IPD against a randomly-selected 20-member subset of its colleagues, with 4 copies of the highest scoring agent being inserted in the population at the end of each generational iteration. {code:nnipd1}

```
int nnipd( int priorm[], int i )   /* neural network function/agent          */
{
   float activation( float ) ;
   int j, k, inputm[4] ;
   float a, sh[2], move ;
   for ( k=0 ; k<4 ; k++ ) {inputm[k]=2*priorm[k]-1 ; }   /* [0,1] to [-1,1] */
   for ( j=0 ; j<2 ; j++ ) {                    /* signals to hidden layer   */
      a=0. ;
      for ( k=0 ; k<4 ; k++ ) { a+= w[i][4*j+k]*inputm[k] ; }   /* Eq (13.3) */
      sh[j]=activation(a) ;                 /* signals from hidden layer     */
   }
   a=w[i][10]+w[i][8]*sh[1]+w[i][9]*sh[1] ;                    /* Eq (13.4)  */
   return (1+a/abs(a))/2 ;                  /* output signal, 0 or 1         */
}
/*-------------------------------------------------------------------------*/
float activation( float a )        /* activation function for neural network */
{ return 2.*(1./(1.+exp(-2.*a))-0.5) ; }    /* activation funct. Eq (13.7)   */
/*-------------------------------------------------------------------------*/
int findbest( float score[] )      /* find best scoring population member    */
{
  float best=0. ; int ibest=0 ;
  for( ip=0 ; ip<np ; ip++) {               /* loop over population          */
    if( score[ip] > best) { best=score[ip] ; ibest=ip ; }
  }
  return ibest ;
}
/*-------------------------------------------------------------------------*/
void replaceandmutate( int ib )    /* generational replacement with mutation */
{
  int ir, ip ;
  float wbest[11] ;
  for(iw=0 ; iw<14 ; iw++) { wbest[iw]=w[ib][iw] ; } /* save best agent      */
  for(ir=0 ; ir<NR ; ir++) {                         /* replace NR agents    */
    ik=floor(NP*1.*rand()/RAND_MAX) ;                /* pick a loser...      */
    for(iw=0 ; iw<10 ; iw++) {                       /* connection weights   */
      if ( 1.*rand()/RAND_MAX < PM ) {
        w[ik][iw]=-2.+4.*rand()/RAND_MAX ;           /* mutation hits        */
      } else { w[ik][iw]=wbest[iw] ; }               /* replace with best    */
    }
    if ( 1.*rand()/RAND_MAX < PM ) {                 /* bias                 */
      w[ik][10]=floor(2.*rand()/RAND_MAX) ;          /* mutation hits        */
    } else { w[ik][10]=wbest[10] ; }                 /* replace with best    */
    for(iw=0 ; iw<1 ; iw++) {                        /* prewired first moves */
       if ( 1.*rand()/RAND_MAX < PM ) {
         w[ik][iw]=floor(2.*rand()/RAND_MAX) ;       /* mutation hits        */
       } else { w[ik][iw]=wbest[iw] ; }              /* replace with best    */
    }
  }                                                  /* end replacement loop   */
}
```

Figure 13.3: The four C functions required by the evolutionary simulation code of Fig. 13.2. {code:nnipd2}

iteration, together with the population-averaged score (red). The NN plotted at right is the best player at iteration 250, and the vector at the far right gives its sequence of move when facing the 16 possible input vectors (RHS of eq. 13.8).

With random initial weights and faced with a variety of input moves, a large enough population of NN will return D or C equiprobably[5]. One would thus expect that such a population of random NN playing against one another will sample the score matrix equiprobably, producing a population-averaged score of $(1+5+0+3)/4 = 2.25$, and this indeed what is observed initially on Fig. 13.4.

In the first few tens of iterations, in all cases the best, average and worst scores all drop rapidly, as indiscriminately cooperating strategies are exploited to extinction by ALLD-like strategies. The gradual disappearance of the latter eventually makes things harder for the former, which sometimes leads to the rise and dominance of other, more complex strategies having managed to survive the early dominance of defecting strategies. The rate of evolution early on in the simulation reflects the level of *selection pressure*, here set by the number of copies of the best strategies introduced in the next generation; the larger that number, the faster the evolution.

Most simulations end up in a state where the best, worst and average scores are identical, suggesting that the whole population is made up of NNs playing the same strategy. In the first first run plotted on the top panel, the population simply converges to a pure ALLD strategy. At the neural network level, this is achieved here through a strong negative bias on the ouput unit, coupled with small connection weights from the hidden to the output layer.

The second run converges, after some 100 generational iterations, to a score-per-move of 3.0. However, here the neural net is playing neither ALLC nor TFT; while it does start off nice (playing C and C on its first two moves), it defects on all but two possible input vectors. Consequently, it does fairly well at exploitation early on, but as it becomes more and more frequent in the population, it still does well because it then behaves effectively like ALLC (or TFT). In other words, this "tribal" agent plays ALLC with copies of itself, but plays ALLD with almost everybody else.

Repeating such simulations starting with different sets of initial random connection weights soon reveals that the stabilized best scores are almost always 1.0, 2.0 or 3.0. For a 4-input neural network playing against itself, only four sequences of moves are possible: persistent dual defection, persistent dual cooperation, and alternating dual defection/cooperation with cadence of 1 or 2:

	...11111111...	...33333333...	...13131313...	...11331133...
P_1 :	...DDDDDDDD...	...CCCCCCCC...	...DCDCDCDC...	...DDCCDDCC...
P_2 :	...DDDDDDDD...	...CCCCCCCC...	...DCDCDCDC...	...DDCCDDCC...
	...11111111...	...33333333...	...13131313...	...11331133...

For our usual IPD payoff matrix (eq. (12.1), the corresponding scores per move are respectively 1, 3, 2, and 2.

In some rare instances (a few percent for the simulation parameters used here), the evolution never stabilizes to a uniform population playing the same strategy. The bottom panel of Fig. 13.4 shows an example, which starts off pretty much like on the panel above, but with the best and average scores settling respectively around 2.6 and 2.2 after some 200 generations, and mildly fluctuating about these values thereafter even with the simulation pushed to 1000 generational iterations. Here two groups of strategies are co-existing in a stable equilibrium, neither one managing to gain numerical dominance over the other.

By their very design, these evolutionary simulations simply pick the "best" strategy present in the initial population. However, a population size of 100 only samples a very small subset

[5]Note that this is *not* the same as RAND, which returns random moves in the course of a single IPD game; here a NN with random connection weights still behaves deterministically against any given opponent.

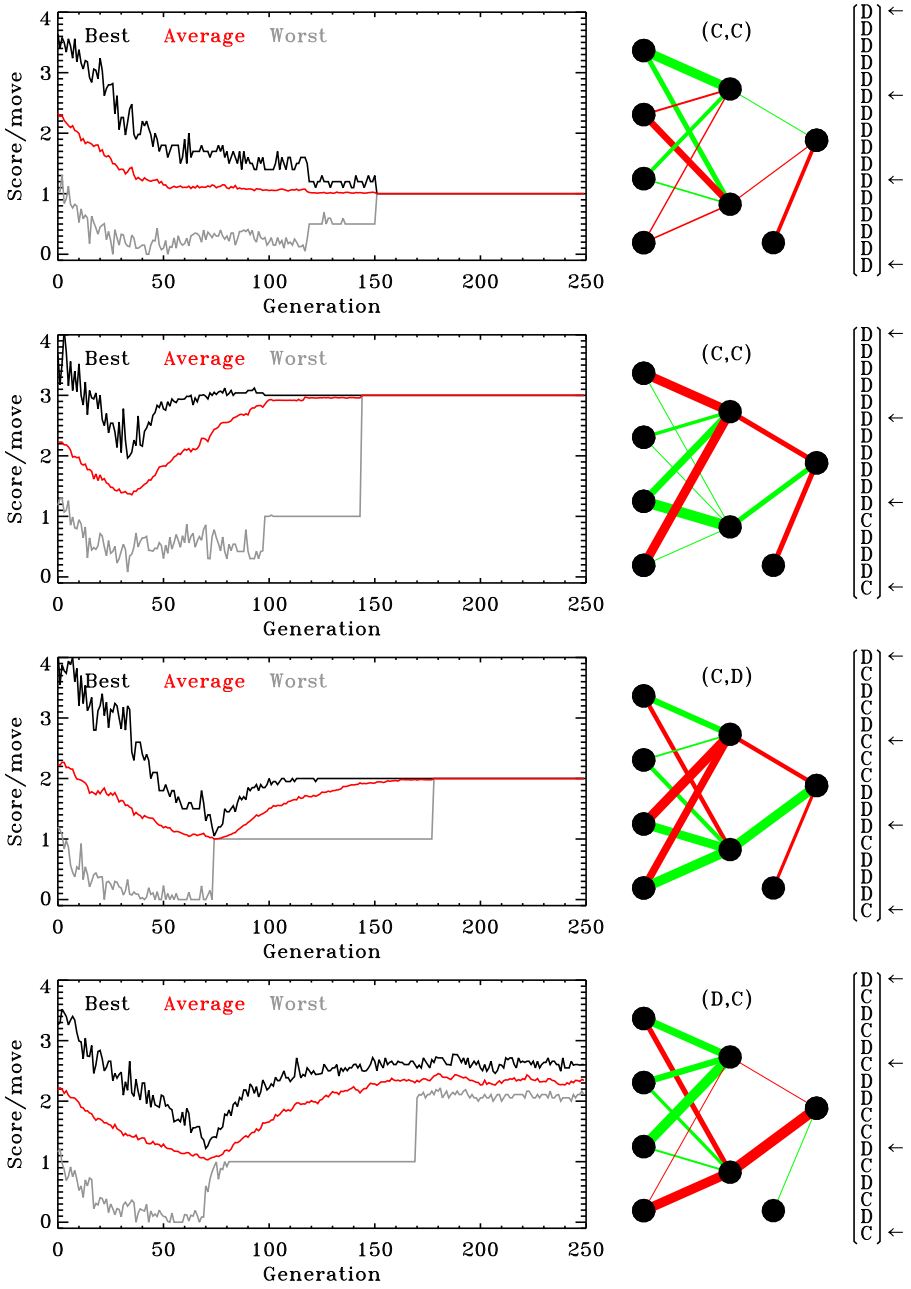

Figure 13.4: Four evolutionary simulations starting from different random populations of 100 agents. At each generational iteration, each agent plays a 10-round IPD with a randomly selected 20-member subset of the population. The black and gray lines plot the score of the best and worst population member, with the red line giving the population average. The neural net displayed on the right corresponds to the best scoring agent at the last iteration. Positive and negative connection weights are drawn in green and red, respectively, with the thickness proportional to the magnitude of the corresponding weight. Listed above the NN are the pre-wired initial moves -1 and -2 for this agent. The vector on the far right gives the move played by this NN for the 16 possible input vectors (see eq. (13.8); for a NN playing againt a copy of itself, only the four moves singled out by arrows are possible. {fig:nnplay}

of possible strategies. To do better we can certainly start with a larger initial population, but a more interesting possibility consists in maintaining variability in the evolving population of NN by introducing *mutations* during the replication process of the best strategy into the next generation.

Consider the following procedure: each time the best NN is copied into the next generation, its connection weights (and pre-wired initial two moves) can be randomly reset within their allowed range, with probability p ($\ll 1$). The impact of such mutations can be large or small, depending on the weight affected and the strategy being encoded. Changing an element of the initial move vector can make strategies like REP1 or REP2 switch from ALLD-like to ALLC-like behavior, or the other way around. In contrast, a random change in a w^{IH} connection weight will have negligible impact if the value of the w^{HO} for the corresponding hidden layer unit is very small, but may have a large impact if that weight dominates the signal to the output unit.

Figure 13.5A shows the first 50000 generational iterations of a representative evolutionary simulation carried out with a mutation rate $p = 0.005$, zero bias to the output unit, and a population of 40 players with 8 being replaced per generation. The black line shows the variations of the score obtained by the best player at each generation, and the red line is the population-averaged score. The horizontal colored bars indicate epochs during which the best-scoring strategy is one of those listed along the bottom horizontal axis, as color coded. A characteristic feature of this simulation (and others carried out using varying selection pressures, mutation rates or bias values) is the appearance of temporally extended epochs where the best and mean scores remain generally stable at values very close to either 1.0, 2.0 or 3.0, values we know already are associated with a single strategy playing against itself. Transitions between these three possible "equilibrium states", in contrast, are often quite rapid, as can be seen on the four closeups plotted on panels B though E. These closeups also include the time series of the worst score in the population (light blue), and in green 10 times the the root-mean-square deviation (σ) about the population mean ($\langle S^n \rangle$):

{eq:meanscore}
$$\langle S^n \rangle = \sum_{j=1}^{N} S_j^n \ , \tag{13.11}$$

{???}
$$\sigma = \left(\frac{1}{N} \sum_{j=1}^{N} (S_j^n - \langle S^n \rangle) \right)^{1/2} \ . \tag{13.12}$$

The r.m.s. deviation σ and the best-worth score difference both provide a measure of variability within the agent population.

The first closeup (panel B) shows the initial 1000 iterations of the simulation. As with the mutation-free simulations of Fig. 13.4, both the best and average score drop rapidly at first, here levelling off around an average score of two. Great variability characterizes this early phase, with TFT present and often extracting the best score, but failing to take over the population. Variability decreases markedly from iteration 670 onwards, with the appearance of a population of defectors behaving effectively like ALLD. Closeup C is extracted later in the simulation, and shows a two-step transition from a population of TDT to REP1 effectively acting like ALLD, with an brief intermediate phase dominated by ATFT. As exemplified by panel D, these transitions can become quite intricate, but fairly rapid transitions tend to be the rule rather than the exception. Sometimes the population switches strategies without leaving a strong signal in the score time series, for example the $\sim$ 100-generation-long transition from TFT to DTFT dominance taking place around generation 36100 on panel E. Neither ALLC, PAV nor TFTT managed to gain even short-lived dominance in this one specific simulation.

The time series plotted on Fig. 13.5 look rather noisy, giving the impression that mutations are continuously perturbing the evolving population. This is partly a consequence of the highly compressed time axis, because the mutation rate is actually quite low here. With 11 connection weights and two prewired moves determining the agent's playing behavior, the probability that a NN be replicated in the next generation without being affected by mutations is $(1-p)^{13} = 0.937$ for $p = 0.005$ here; a lot of the variability observed stems from the fact that each player faces

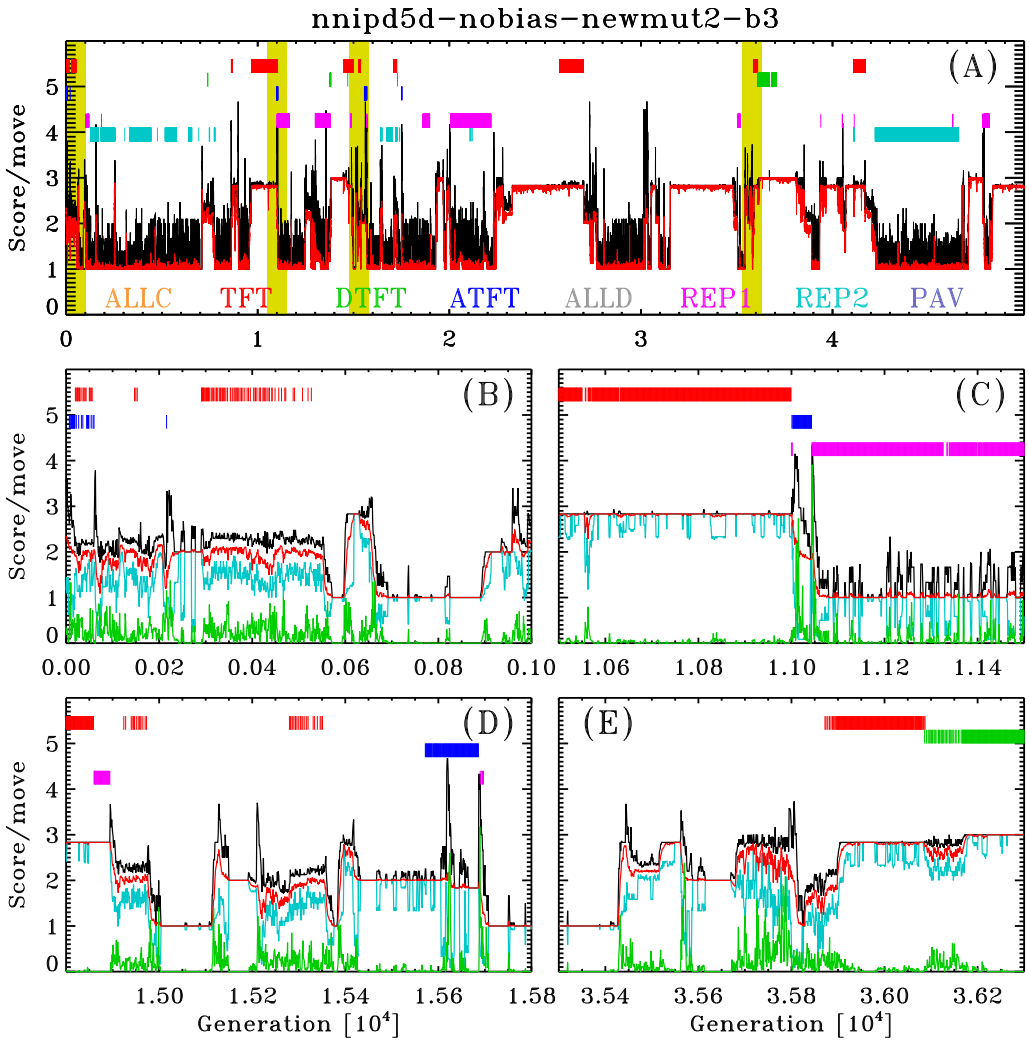

Figure 13.5: The top panel shows 50000-generation time series of the best (black) and population-averaged (red) scores in an evolutionary simulation using a population of 40 agents, mutation rate $p = 0.005$, no bias to the ouput unit ($b_1^O = 0$), and replacement of 10% of the population at each generational iteration. The colored bands along the top horizontal axis identify periods were certain specific strategies are the best players, the color coding being given along the lower axis. Panels B through D are 1000-generation closeups spanning the intervals indicated by yellow bands on panel A. Added on these these closeups are the time series for the worst score (light blue), and the r.m.s. deviation of the agents' scores about the population mean (green, multiplied by a factor of 10). {`fig:longts`}

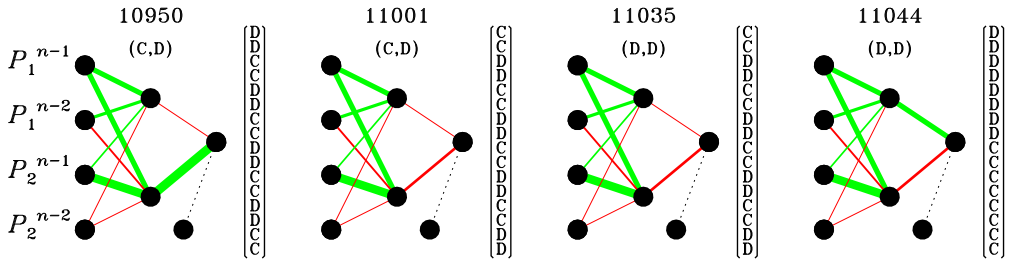

Figure 13.6: Top-scoring neural networks just before (left), during (middle) and after (right) the rapid evolutionary transition occuring around generation 11000 on Figure 13.5C. The plotting format is described in the caption to Fig. 13.4. The bias connection to the output unit is plotted as a dotted line since in this specific simulation the bias is set to zero. This evolutionary transition occurs as a consequence of mutations affecting first the w_2^{HO} connection weight, then the first prewired move, and subsequently w_1^{OH} (see text). {fig:plot3nn}

off with only 10 randomly chosen opponents per generation, so score fluctuations are expected even in the absence of mutations.

Even if they are relatively rare, mutations can have a profound evolutionary impact. Figure 13.6 shows the mutation-driven changes in the top-scoring neural network during the rapid evolutionary transition taking place around generation 11000 on Fig. 13.5C. Prior to the transition the best NN is playing TFT[6], with the NN's behavior dominated by the neural path $P_2^{n-1} \to w_{32}^{IH} \to S_2^H \to w_{21}^{HO} \to S_1^O$. At generation 10999 a mutation hits w_{21}^{HO}, changing its sign; the NN then starts returning the exact opposite move as before the mutation, i.e., it now plays ATFT. A mutation at generation 11035 changes the player's prewired first move from C to D, but this has no impact since ATFT's behavior is determined only by the opponent's prior move. However, a further mutation at generation 11044 changes the connection weight w_{11}^{HO} from a small, negative value to a moderately large positive value, which transfers dominance to the neural path $P_1^{n-1} \to w_{11}^{IH} \to S_1^H \to w_{11}^{HO} \to S_1^O$. The NN is now playing REP1, but effectively behaving like ALLD since its prewired first move is D.

13.4 Punctuated equilibrium ?

As a loose analog of "evolution", Figure 13.5 stands a long way from the picture of geologically slow, gradual change associated with the original formulation of Charles Darwin. Instead, new populations of strategies are produced in intermittent bursts of evolutionary activity, delineating long period of statis. Paleontologists had been staring at this very same pattern in the fossil record for a long time before coming to grips with the fact that it did not result from "missing data", but represents a real pattern, now known as *punctuated equilibrium*. But are we really seeing a similar pattern in our immensely simpler evolutionary IPD simulations?

If we identify the uniform population of strategies during stable phases as "species", then one relatively unambiguous possibility of comparison lies with the distribution of species lifetimes, which is reasonably well-documented in the fossil record. In our simulations this means identifying boundaries of stable regions, a task nowhere as straightforward as one might imagine in view of the significant variability present even during stable phases. The procedure adopted is illustrated on the top panel of Figure 13.7, and operates as follows. First, the time series of population-averaged score is first smoothed using a 5-point trapezoidal running mean,

[6] Careful examination of Fig. 13.5C will reveal that the best and population-averaged scores-per-move during the TFT phase are slightly below 3.0; this is because the pre-wired move P_1^{-2} is here D, which yields a small but noticeable drop in total score since the game is being played only 10 times per encounter in the simulations considered here.

mathematically defined as:

{???}
$$\bar{S}_k = \frac{1}{8}\left(\langle S\rangle_{k-2} + 2\langle S\rangle_{k-1} + 2\langle S\rangle_k + 2\langle S\rangle_{k+1} + \langle S\rangle_{k+2}\right) \ , \tag{13.13}$$

where $\langle S\rangle_k$ is the population average at generational iteration k (viz. eq. (13.11)). Then, the resulting time series is then scanned, and a *transition* is deemed to take place at generational iteration k whenever

$$|\bar{S}_{k+1} - \bar{S}_{k-1}| \geq 0.1 \ . \tag{13.14}$$ {???}

Once the beginning and ends of stable phases have been identified in this manner, their duration, equivalent to species lifetime, is readily calculated. The bottom panel of Figure 13.7 shows a PDF of this quantity, for a 250000 generation simulation in which 561 stable phases have been identified via the procedure just described. The PDF takes the form of a power law, here with logarithmic slope -1.3. Higher mutation rates and stronger selection pressure also yield power-law PDFs, but with steeper slopes, reaching as high as -1.8. Remarkably, distribution of species lifetimes inferred from the fossil record are also distributed as a power law, with a somewhat steeper logarithmic of -2. Coincidence ? Maybe; or maybe not.

In the simulation setup used throughout this chapter mutations events are independent of one another and occur at a time-independent rate. If the transitions on Fig. 13.7 were directly and uniquely triggered by mutation, then for such a stationary memoriless random process one would expect Poisson statistics to hold, which should lead to an exponential distribution of inter-event waiting times, i.e., species lifetime. The fact that we observe instead a power-law indicates that some other process introduces "memory" into the system; that process is, of course, score-dependent selection and replication in the next generation. More...

13.5 Evolution on the computer

The evolutionary IPD simulations investigated in this chapter are worlds away from real biological evolution, yet they do embody its three fundamental operational principles: selection, inheritance and variability. The biological analogy can be pushed further, for example by encoding the weight vector (13.10) as a chromosome-like binary string; after which *pairs* of high-scoring agents are selected and their chromosomes recombined to produce offsprings through genetically-inspired crossover and mutation operators. *Genetic Algorithms* operate in this manner, and have been used quite succesfully for complex data modelling and engineering design tasks. That the complexity of biological evolution could be harnessed in this manner to crack tough problems in the physical sciences is both remarkable and wonderful in the most profound sense of the word. Some people may find this also surprising, but DNA co-discoverer Francis Crick perhaps expressed it best: "Evolution is cleverer than you are!"[7]

Genetic algorithms

Crick, Francis
Rapoport, Anatol
Hofstadter, Douglas

13.6 Exercices and further computational explorations

1. Construct the move vector (see eq. (13.9) and equivalent compact representation (see footnote 4) for the strategy PAV introduced in the previous chapter (§12.1).

2. Draw a feed-forward neural network where the *three* prior moves of the players and its opponent are used as input, and with three units in the hidden layer. How many connection weights ae needed to define the operation of such a network ? How many distinct input move vectors are there ? How many distinct strategies can such a network encode ?

[7] Admittedly, all we have really demonstrated in this chapter is that evolution is at least as clever as Anatol Rapoport was; this is already not bad at all! Then again, Anatol Rapoport's brain was a product of evolution, so we should perhaps have expected this... or not ? Following this line of reasoning further is likely to get us into what Douglas Hofstadter dubbed *strange loops*, so let's quit it while we're still just only mildly confused.

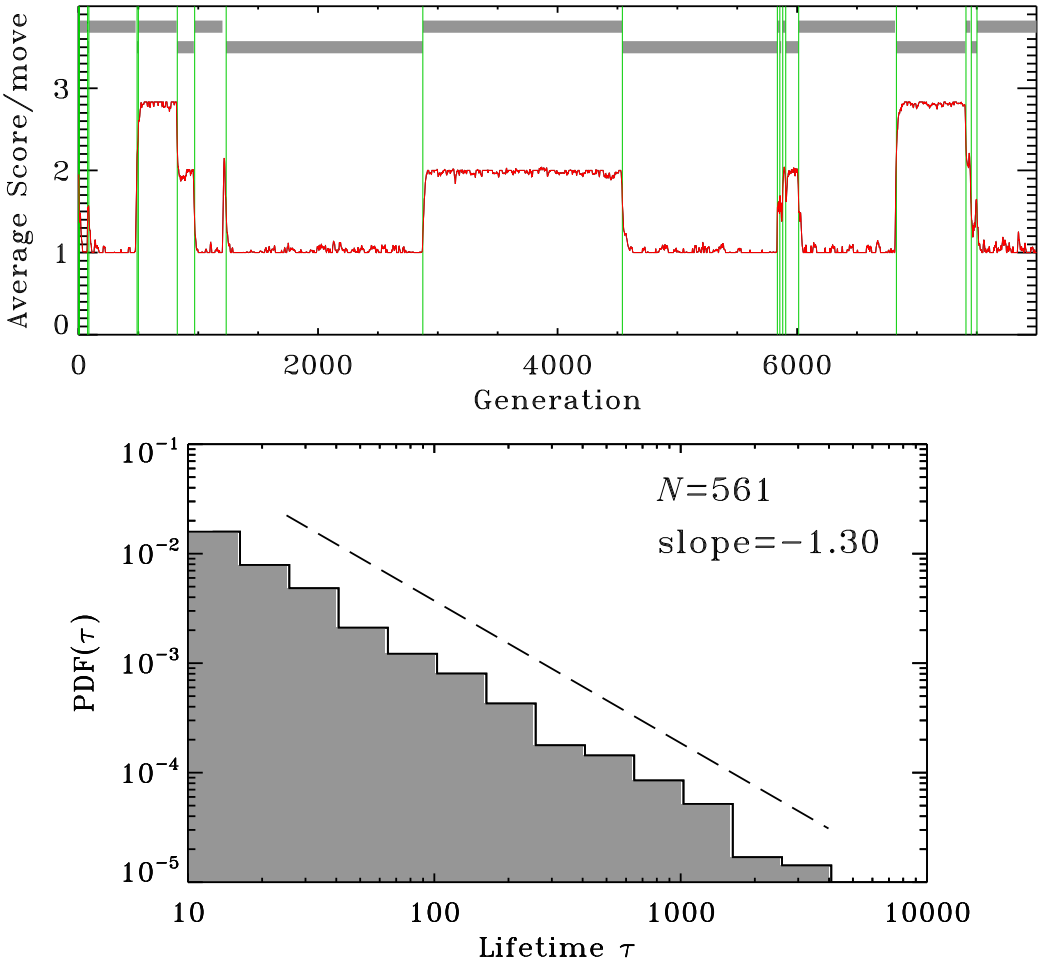

Figure 13.7: Probability density function of "species" lifetime for a 250000 generation evolutionary simulation using a population of 40 agents, mutation rate $p = 0.005$, bias $b_1^O \in [-1, +1]$ and replacement of 10% of the population at each generational iteration. The top panel shows the first 8000 generations of the smoothed time series of population-averaged score-per-move, with periods of population stability indicated by gray horizontal bands and the green vertical lines indicating the onset of stable epochs. The bottom panel show the probability density function of the durations of stable epochs, characterized by a power-law form spanning over two orders of magnitude in duration. {`fig:punctequ`}

3. repeat sims of 13.3 with increasing mutation rate; when do stable phases disappear ? how is PDF affected ?

4. roulette wheel algorithm for selection

5. Five

6. And now, the ultimate Grand Challenge: Spatial IPD with random walking NNs; whenn 2 NN meet, the winner replicates the loser.

13.7 Further readings

There is no second chance to make a first impression; my first contact with neural network was the following book, which made quite a first impression on me because twenty years and many more books later, I still find it an outstanding introduction to the topic at large:

Anderson, J.A., *An Introduction to Neural Networks*, MIT Press (1995).

At a mathematically more advanced level, I also learned a lot from :

Haykin, S., *Neural Networks: a comprehensive Foundation*, MacMillan Publishing (1994).

On punctuated equilibrium, you might as well start with a retrospective view from one of the original proponents of the idea:

Gould, S.J., *Punctuated Equilibrium*, Harvard University Press (2007).

Many good introductory textbooks on genetic algorithms are available. I happened to learn the topic from the following excellent tutorial-like book (yes, first impressions again...):

Genetic algorithms

Davis, L. (ed.), *Handbook of Genetic Algorithms*, Van Nostrand Reinhold (1991).

The following classic, although mathematically and conceptually more arduous, remains a must-read:

Holland, J.H., *Adaptation in Natural and Artificial Systems*, MIT Press (1992)