CS 545 Assignment 7
Learning Dynamic Control with Reinforcement Learning

Dave Matthews

December 8, 2009

Contents

1 Introduction

In this assignment we studied reinforcement learning [1, 2] applied to a simple control problem. We used reinforcement learning to generate a set of samples that are used to training a neural network. The goal of the assignment was to study the parameters involved in the algorithm. We explored two variations of the problem, one with a fixed goal and one with a variable goal.

2 R code

We modified the R code provided with the assignment [**?**] to allow use with fixed and variable goals. The fixed goal of 5 requires only two state parameters (position and velocity). The variable goal of 1 to 10 requires three state parameters (position, velocity, and goal). The number of neural network input parameters ni determines which type of goal to use and the remainder of the code acts accordingly. There were no changes to the `epsilonGreedy` function so we do no list it here.

2.1 reinforcement

The `reinforcement` function was modified to accept the goal and proximity as parameters. The defaults correspond to those provided in the code with the assignment [4]. A reinforcement of 1 is returned when the new state is within the specified proximity of the goal.

```
### Return reinforcement given current state
defaultGoal      <- 5
defaultProximity <- 2

reinforcement <- function(s,s1,goal=defaultGoal,proximity=defaultProximity) {
  if (abs(s1[1]-goal) < proximity) {
    ## Within proximity of goal, return 1
    1
  } else {
    ## Not within proximity of goal, return 0
    0
  }
}
```

2.2 initialState

The `initialState` function produces a starting state at a random position on the track with no velocity. The function returns a state with two values (position and velocity) when there is a fixed goal ($ni = 2$). The function returns a state with three values (position, velocity, and goal) when there is a variable goal ($ni = 3$). This should correspond to the number of neural network inputs ni specified in the main program below.

```
### Return new random state

initialState <- function(g=defaultGoal) {
  if (ni > 2)
    c(sample(10,1),0,g) # include the goal.
  else
    c(sample(10,1),0)
}
```

2.3 nextState

The `nextState` function updates the state information (position and velocity) for one time step. The returned state `s` contains 2 values when using a fixed goal ($ni = 2$) and 3 values when using a variable goal ($ni = 3$).

```
### Return updated state, given the action to apply to old state

nextState <- function(s,a,goal=defaultGoal,deltaT=0.1) {
  ## s[1] is position, s[2] is velocity, s[3] is the goal when variable.
  ## a is -1, 0 or 1

  ## Euler integration time step
  ## deltaT <- 0.1 converted to parameter for variation
```

```
  ## Update position
  s[1] <- s[1] + deltaT * s[2]
  ## Update velocity. Includes friction
  s[2] <- s[2] + deltaT * (2 * a - 0.2 * s[2])
  ## Include the goal in the state when using a variable goal
  if (ni > 2)
    s[3] <- goal

  ## Bound next position. If at limits, set velocity to 0.
  if (s[1] < 1)
    if (ni > 2)
      s <- c(1,0,goal) # include the goal in the state when using a variable goal.
    else
      s <- c(1,0)
  else if (s[1] > 10)
    if (ni > 2)
      s <- c(10,0,goal) # include the goal in the state when using a variable goal
    else
      s <- c(10,0)

  ## Return new state
  s
}
```

2.4 getSamples

The `getSamples` function provides a sample set of interactions for training the neural network. The function operates with either a fixed goal ($ni = 2$) or a variable goal($ni = 3$) that creates blocks of 100 consecutive samples for each possible goal.

```
### Return set of sample interactions.  All are matrices with one row per time step.

getSamples <- function(net,numSamples,epsilon,goal=defaultGoal,proximity=
    defaultProximity,deltaT=0.1) {
  X <- matrix(0,numSamples,ni)
  R <- matrix(0,numSamples,1)
  Q <- matrix(0,numSamples,1)
  Y <- matrix(0,numSamples,1)
  AI <- matrix(0,numSamples,1)

  ## determine the fixed or variable goal
  if (ni > 2)
    # produces random sample of goals in blocks of 100
    G <- matrix(apply(matrix(sample(rep(1:10,ceiling(numSamples/10/100)))),
                      1,
                      function(x) rep(x,100)),
                numSamples,1)
  else
    G <- matrix(rep(5,numSamples),numSamples,1)

  ## Initial state
  s <- initialState(G[1])

  ## Select action using epsilon-greedy policy and get Q
  eg <- epsilonGreedy(net,s,epsilon)
  a <- eg$a
  aI <- eg$aI
  q <- eg$Q

  for (step in 1:numSamples) {

    ## Update state, s1 from s and a
    s1 <- nextState(s,a,G[step],deltaT)
```

```
    ## Get resulting reinforcement
    r1 <- reinforcement(s,s1,G[step],proximity)

    ## Select action for next step and get Q
    eg <- epsilonGreedy(net,s1,epsilon)
    a1 <- eg$a
    a1I <- eg$aI
    q1 <- eg$Q

    ## Collect
    X[step,] <- s
    R[step,1] <- r1
    Q[step,] <- q1
    Y[step,] <- q
    AI[step,1] <- aI

    ## Shift state, action and action index by one time step
    s <- s1
    a <- a1
    aI <- a1I
  }
  list(X=X, R=R, Q=Q, Y=Y, AI=AI)
}
```

2.5 drawSurfaces

The `drawSurfaces` function produces graphs of the Q and policy for analysis. We made several improvements to the code provided with the assignment [4]. The function takes the goal as a parameter to allow analysis of the different goals, and determines whether or not to use the goal in computations based on the number of neural network inputs. We also modified the function to use the symbols ¡, ¯, and ¿ to signify left, stay, and right actions, made the colors consistent across the graphs, and labeled the graphs. This makes the interpretation of the policy a bit easier.

```
### Draw Q and policy as surfaces and contours or images

drawSurfaces <- function(net,goal=defGoal) {
  xs <- seq(1,10,len=20)
  xds <- seq(-4,4,len=20)
  q <- matrix(0,length(xs),length(xds))
  a <- matrix(0,length(xs),length(xds))
  nh <- ncol(net$V)
  z <- array(0,c(nh,length(xs),length(xds)))
  for (i in 1:length(xds)) {
    for (j in 1:length(xs)) {
      if (ni > 2)
        out <- nnOutput(net,matrix(c(xs[j],xds[i],goal),nrow=1))
      else
        out <- nnOutput(net,matrix(c(xs[j],xds[i]),nrow=1))
      q[j,i] <- max(out$Y)
      a[j,i] <- actions[which.max(out$Y)]
      for (h in 1:nh)
        z[h,j,i] <- out$Z[h]
    }
  }
  cat("range of Q",range(q),"\n")
  phi   <- 35
  theta <- 45
  ##Q
  persp(xs,xds,q,xlab="x",ylab="xdot",zlab="Q", phi=phi, theta=theta, ticktype="detailed",
    main=paste("Q max",signif(max(q),digits=3)))
  contour(xs,xds,q,xlab="x",ylab="xdot",main=paste("Q max",signif(max(q),digits=3)))
  ##policy
```

```
  persp(xs,xds,a,xlab="x",ylab="xdot",zlim=c(-1,1),zlab="Action", phi=phi, theta=theta,
      ticktype="detailed",main="Policy")
  image(xs,xds,a,col=c("black","red","green"),zlim=c(-1,1),main="Policy")
  ##hidden
  for (h in 1:nh) {
    persp(xs,xds,z[h,,],xlab="x",ylab="xdot",zlim=c(-1,1),zlab="Z", phi=phi, theta=theta
        , ticktype="detailed",
      main=paste("hidden unit",h))
  }
}
```

2.6 trainIT

The `trainIT` function. A reinforcement average (*ratrace*) was added to the code provided with the assignment

```
###  Train the neural network using reinforcement learning.

trainIt <- function(nh,nReps,N,maxIterations,lambda,gamma,finalEpsilon)
{
  dev.new(width=pdfwidth,height=pdfheight)

  epsilonRate <- exp(log(finalEpsilon)/nReps)
  cat("epsilon decay rate = ",epsilonRate,"\n")
  ### Initial epsilon is 1, for fully random action selection
  epsilon <- 1

  ### Variables for plotting later
  ftrace <- NULL
  rtrace <- NULL
  ratrace <- NULL
  xtrace <- NULL
  epsilontrace <- NULL
  sigma <- 0.1
  ra <- 0

  ### Number of inputs to Q neural network,  position and velocity and goal
  ni <- 3
  ### Values of each of the 3 actions
  actions <- c(-1,0,1)

  ### Set up graphics display
  nplots <- 10 + nh
  nc <- 4
  nr <- ceiling(nplots/nc)
  nempty <- nr*nc-nplots
  pr <- par(mfrow=c(nr,nc))

  ################################
  ### Create the neural network for Q function. It has ni (2) inputs, nh hidden units,
      and as many outputs as actions (3).
  ### When creating it, specify the range of each input so makeStandardizeF can create
      the correct function.
  if (ni > 2) # goal information included in inputs
    net <- makeNN(ni,nh,length(actions),matrix(c(1,10,-5,5,1,10),2,3))
  else # only position and velocity information
    net <- makeNN(ni,nh,length(actions),matrix(c(1,10,-5,5),2,2))


  ################################
  ### Main loop start
  for (reps  in 1:nReps) {

    ## Collect N[reps] samples, each being s, a, r, s1, a1
    samples <- getSamples(net,N[reps],epsilon) # vary goal,proximity,deltaT
```

```
    ## Update the Q neural network.
    net <- updateNN(samples$X, samples$R, samples$Q, samples$Y, samples$AI,
                    net, gamma=gamma, lambda=lambda, fPrec=1e-8, nIter=maxIterations)

    ## Draw Q, policy, and hidden unit surfaces
    drawSurfaces(net)

    ## Draw neural network
    if (ni > 2) # include the goal
      drawNNet(net, c("x","xd","g"))
    else # only position and velocity
      drawNNet(net, c("x","xd"))


    ## Track and plot trace of reinforcement values.
    meanR <- mean(samples$R)
    ra  <- (1.0-sigma)*ra + (sigma)*meanR  #exponential weighted moving average of
        reinforcements
    rtrace <- c(rtrace, meanR)
    ratrace <- c(ratrace, ra)
    plot(rtrace, type="b", ylim=c(0,1), main=paste("Reinforcement", signif(ra, digits=3)))
    mtext(paste("best", signif(max(rtrace), digits=3),
      ", best ra", signif(max(ratrace), digits=3)), side=3, line=0.1, cex=0.8)
    lines(ratrace, type="l", lwd=2, col="blue")

    ## Track and plot epsilon values.
    epsilontrace <- c(epsilontrace, epsilon)
    plot(epsilontrace, type="b", ylim=c(0,1), main="Epsilon")

    ## Track and plot TD errors
    ftrace <- c(ftrace, net$ftrace)
    plot(ftrace, type="l", main="TD errors")

    ## Plot last set of interactions, forming a trajectory
    plot(samples$X[,1], samples$X[,2], type="l", xlab="x", ylab="x dot",
         xlim=c(1,10), ylim=c(-5,5), main="Trajectory")

    ## Draw the policy for 0 velocity states with a goal of 5.
    if (ni > 2) # include the goal
      acts <- apply(useNN(net, cbind(matrix(1:10),0,5)),1,which.max) ## use goal of 5
    else # only position and velocity
      acts <- apply(useNN(net, cbind(matrix(1:10),0)),1,which.max) ## assumes goal of 5
    x <- 1:10
    y <- rep(0,10)
    symbols(x,y, circles=rep(0.4,10), bg=acts,
            inches=FALSE, xaxt="n", yaxt="n", bty="n", xlab="", ylab="",
            main="Policy goal=5")
    text(x,y+0.4, c("<","^",">")[acts], col=acts, cex=2, font=2)

    ## Advance to correct axes if necessary, for next repetition.
    if (nempty > 0)
      for (i in 1:nempty)
        plot(0,1, type="n", , xaxt="n", yaxt="n", ylab="", xlab="", bty="n")

    ## Update epsilon
    epsilon <- epsilonRate * epsilon

  }
  net
}
```

2.7 main

The main routine initializes the global and default parameters, then runs a series of tests varying one parameter at a time. The remaining parameters use the defaults that were provided in the code with the

assignment [4] The same code is used with a fixed or variable goal, only the number of inputs to the neural network *ni* needs to change.

```
### Default parameter initialization.
actions <- c(-1,0,1)

### number of neural network inputs
### 3 inputs for varying goal, 2 inputs for fixed goal
ni <- 3

### Final probability of random action
finalEpsilon <- 0.001
### Hidden weight penalty.  0 means no penalty
lambda <- 0.00
### Discount factor
gamma <- 0.9
### Number of repetitions of generate-examples/update-net loop
nReps <- 100
### Number of interaction steps in each repetition
N <- rep(2000,nReps)
### Maximum number of iterations for SCG each repetition.
maxIterations <- 10
### Number of hidden units.
nh <- 6

### vary hidden units
for(nh in seq(2,20,by=1))
{
  net <- trainIt(nh,nReps,N,maxIterations,lambda,gamma,finalEpsilon)
  dev.copy2pdf(file=paste("part1/nh",nh,"train.pdf",sep=""))
  dev.off()
}
nh <- 6

### vary maximum gradient descent iterations
for(maxIterations in seq(5,20,by=1))
{
  net <- trainIt(nh,nReps,N,maxIterations,lambda,gamma,finalEpsilon)
  dev.copy2pdf(file=paste("part1/maxIterations",maxIterations,"train.pdf",sep=""))
  dev.off()
}
maxIterations <- 10

### vary learning repetitions, fixed number of samples per repetition
for(nReps in seq(50,200,by=10))
{
  N <- rep(2000,nReps)
  net <- trainIt(nh,nReps,N,maxIterations,lambda,gamma,finalEpsilon)
  dev.copy2pdf(file=paste("part1/nReps",nReps,"train.pdf",sep=""))
  dev.off()
}
nReps <- 100
N <- rep(2000,nReps)

### vary number of repetitions and number of samples per repetition,
### keeping total number of samples constant
for(nReps in c(10,20,40,50,100,200,400,500,1000, 2000,4000,5000,10000,20000))
{
  ### 100 * 2000 is the total number of repetitions
  N <- rep(2000*100/nReps,nReps)
  net <- trainIt(nh,nReps,N,maxIterations,lambda,gamma,finalEpsilon)
  dev.copy2pdf(file=paste("part1/N",2000*100/nReps,"train.pdf",sep=""))
  dev.off()
}
nReps <- 1000
N <- rep(2000*100/nReps,nReps)
```

```
### vary weight penalty factor
for(lambda in c(0.0, 1e-8, 1e-7, 1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 2e-1, 3e-1, 4e-1, 5
    e-1, 6e-1, 7e-1, 8e-1, 9e-1))
{
  net <- trainIt(nh,nReps,N,maxIterations,lambda,gamma,finalEpsilon)
  dev.copy2pdf(file=paste("part1/lambda",lambda,"train.pdf",sep=""))
  dev.off()
}
lambda <- 0.00
}

### vary the discount factor
for(gamma in seq(0.05,1,by=0.05))
{
  net <- trainIt(nh,nReps,N,maxIterations,lambda,gamma,finalEpsilon)
  dev.copy2pdf(file=paste("part1/gamma",gamma,"train.pdf",sep=""))
  dev.off()
}
gamma <- 0.9

### vary the final probability of taking a random action
for(finalEpsilon in c(0, 1e-8, 1e-7, 1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 2e-1, 3e-1, 4e
    -1, 5e-1, 6e-1, 7e-1, 8e-1, 9e-1))
{
  net <- trainIt(nh,nReps,N,maxIterations,lambda,gamma,finalEpsilon)
  dev.copy2pdf(file=paste("part1/finalEpsilon",finalEpsilon,"train.pdf",sep=""))
  dev.off()
}
finalEpsilon <- 0.001
```

3 Parameter Study

This section studies the algorithm parameters, varying one parameter at a time and keeping the others fixed. We used the reinforcements received as a measure of the performance of the algorithm. A table for each parameter highlights the maximum Q values obtained, the best reinforcement receiving, the best average reinforcement value, and the last average reinforcement value. A figure for selected parameter values show several aspects of the resulting neural network, including the learned Q function, policy, hidden unit, and reinforcements received. The average values are based on the exponential weighted moving average of the reinforcements received that appears as a blue line on the Reinforcement graphs in the figures.

We chose to use only a single run for each value rather than a weighted average. This leads to some interesting variability in the results for most parameters. In some cases there appear to be trends, while in others the results oscillate between the extremes.

3.1 Hidden Units

The reinforcements received as we varied the number of hidden units showed little correlation to the number of hidden units as seem in Table 1. Tests with 2, 7, 10, and 17 hidden units received similar high reinforcements on average while tests with 3, 11, and 19 hidden units all received similar low reinforcements on average. Reinforcement did not necessary correlate to learning the policy as only 6, 9, 10, and 15 hidden units appeared to learn a correct policy. The analysis in figure 1 shows a valid policy with the temporal difference error converging towards 0 and consistently high reinforcement.

3.2 Repetitions

We varied the number of repetitions from 50 to 200 while keeping the number of samples for each iteration constant at 2000. The reinforcements received in Table 2 as we varied the number of repetitions showed pretty consistent performance above 80 repetitions. Performance decreased significantly below 90 repetitions. This suggests a minimum number of repetitions is necessary to learn the policy. The analysis in figure **??**

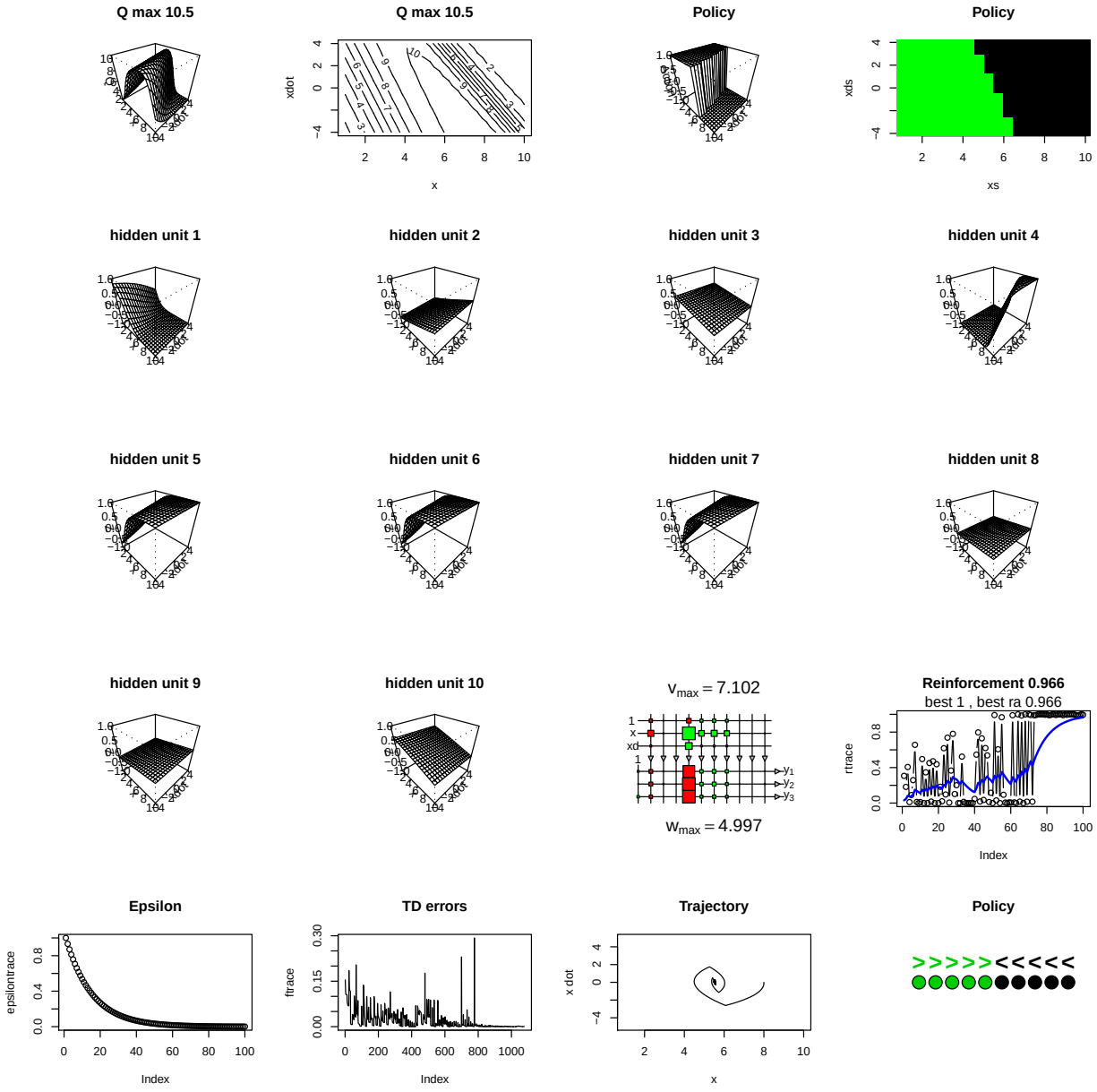

Figure 1: Analysis of 10 hidden units with a fixed goal

hidden units	Q_{max}	best r	best r average	last r average
2	11.5	1.000	0.934	0.934
3	7.8	1.000	0.253	0.046
4	10.4	1.000	0.969	0.663
5	10.6	1.000	0.914	0.802
6	11.7	1.000	0.836	0.819
7	10.0	1.000	0.926	0.926
8	11.2	1.000	0.638	0.508
9	10.6	1.000	0.651	0.651
10	10.5	1.000	0.966	0.966
11	9.6	1.000	0.479	0.359
12	10.6	1.000	0.805	0.636
13	10.3	1.000	0.746	0.516
14	11.0	1.000	0.511	0.428
15	10.5	1.000	0.773	0.700
16	10.7	1.000	0.859	0.599
17	10.5	1.000	0.951	0.951
18	11.2	1.000	0.571	0.263
19	7.8	0.468	0.081	0.005
20	10.1	1.000	0.600	0.414

Table 1: Q and Reinforcements for hidden units ranging from 0 to 1

shows a valid policy with temporal difference converging to 0. In a few cases the policy diverged after 100 repetitions, possibly due to random choices, but the policy was quickly restored.

3.3 Samples per Repetition

In this section, we varied the number of repetitions and samples per repetition to keep the total number of samples constant. The results in Table 3 suggest the extreme ends of this range perform poorly. Too few samples does not provide sufficient training data to update the neural network, while too few repetitions does not provide sufficient updates to allow the neural network to converge.

Figures 3 and 4 analyze the extremes that learned the appropriate policy. Note the significant differences in the Q functions and boundaries between these approaches. With a large number of samples, the reinforcement converges towards 1 while the temporal difference error converges to 0. With a small number of samples, the reinforcement average converges towards 1 but there is still quite a bit of variability in the individual reinforcements and the temporal different errors.

3.4 Maximum Gradient Descent Iterations

In this section, we varied the maximum number of iterations $maxIterations$ used by the gradient descent method from 5 to 20. The goal would be the smallest value sufficient to train the neural network. The results in Table 4 suggests small values can severely impact the training process. We did use a large enough value to definitely determine an upper limit for this problem. The value of 10 suggested in the code for the assignment [4] may be too small and could be the reason for the significant variability in results. The analysis in Figure 5 shows a correct policy with both the reinforcement and temporal difference errors converging appropriately.

3.5 Weight penalty factor λ

In this section, we varied the weight penalty factor λ from 0 to 0.1. The results in Table 5 suggest that the smallest value of 0 performs the best and performance degrades as λ increases. The analysis in Figure 6

repetitions	Q_{max}	best r	best r average	last r average
50	11.4	1.000	0.452	0.346
60	7.01	0.996	0.201	0.010
70	11.5	1.000	0.490	0.298
80	11.3	1.000	0.641	0.552
90	10.4	1.000	0.951	0.951
100	10.4	1.000	0.927	0.873
110	10.4	1.000	0.931	0.854
120	11.4	1.000	0.669	0.588
130	10.7	1.000	0.987	0.975
140	10.3	1.000	0.995	0.965
150	10.7	1.000	0.864	0.864
160	10.2	1.000	0.995	0.887
170	10.1	1.000	0.997	0.997
180	10.3	1.000	0.996	0.987
190	10.5	1.000	0.972	0.809
200	9.71	1.00	0.992	0.862

Table 2: Q and Reinforcements for repetitions ranging from 50 to 200

samples	repetitions	Q_{max}	best r	best r average	last r average
10	20000	2.86	1.000	0.780	0.286
20	10000	12.6	1.000	0.891	0.463
40	5000	10.6	1.000	0.938	0.788
50	4000	10.4	1.000	0.924	0.827
100	2000	10.2	1.000	0.967	0.631
200	1000	10.3	1.000	0.978	0.909
400	500	10.6	1.000	0.987	0.857
500	400	10.3	1.000	0.995	0.977
1000	200	9.9	1.000	0.990	0.857
2000	100	10.3	1.000	0.988	0.988
4000	50	6.50	0.595	0.147	0.004
5000	40	9.97	1.000	0.736	0.736
10000	20	4.56	0.427	0.080	0.015
20000	10	0.421	0.299	0.030	0.017

Table 3: Q and Reinforcements for samples ranging from 400 to 20000

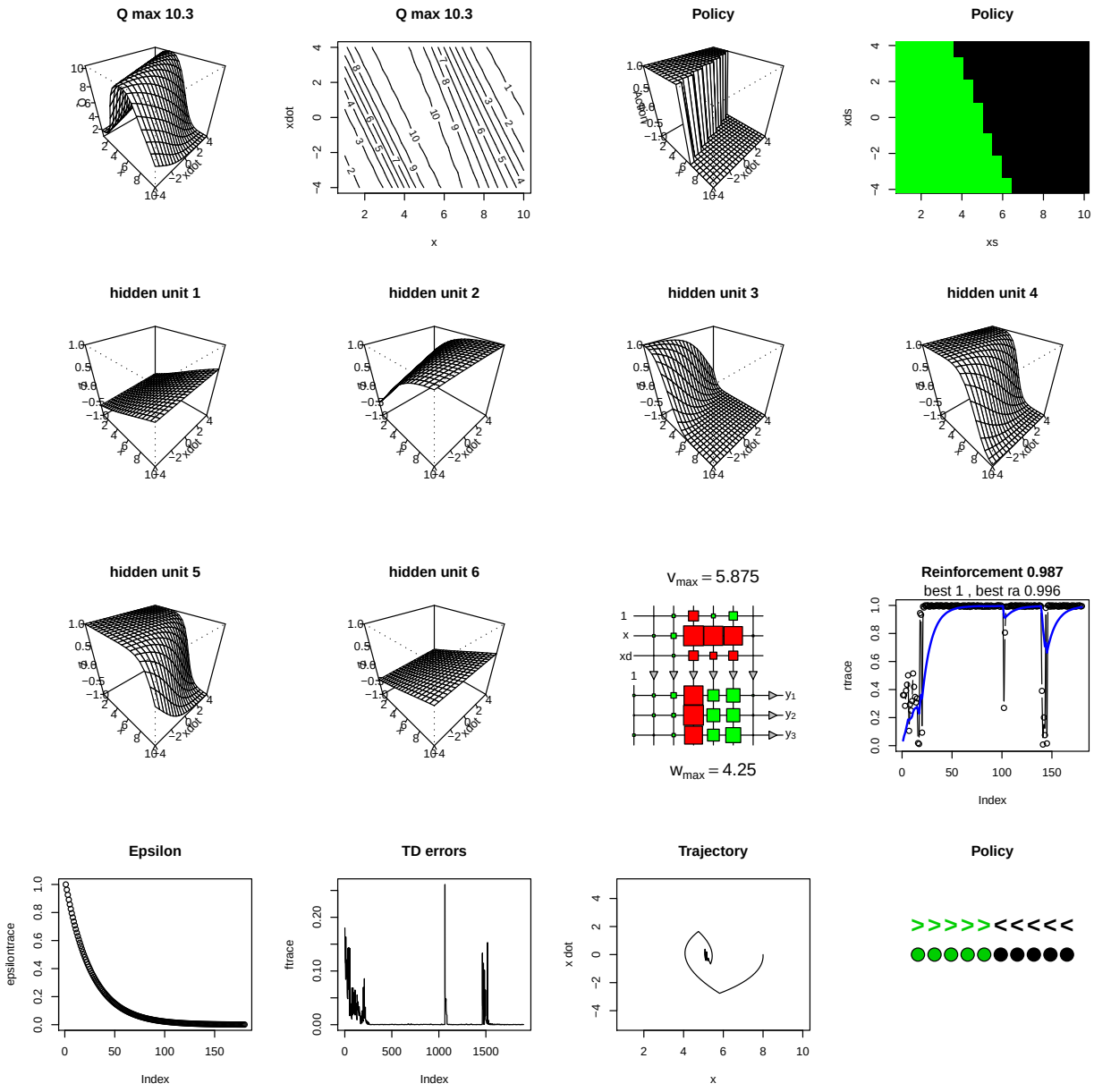

Figure 2: Analysis of 180 repetitions with a fixed goal

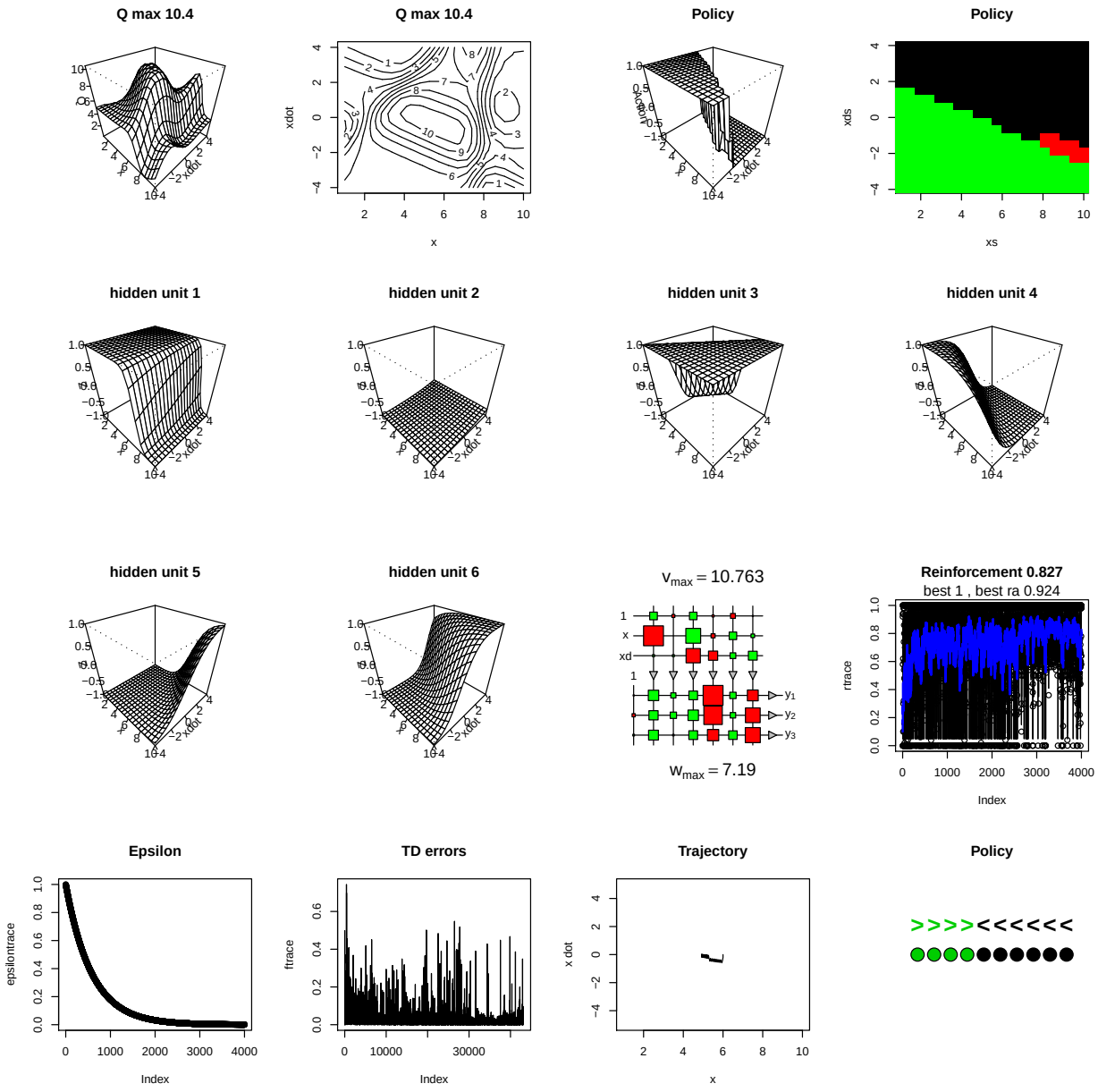

Figure 3: Analysis of 50 samples and 4000 repetitions with a fixed goal

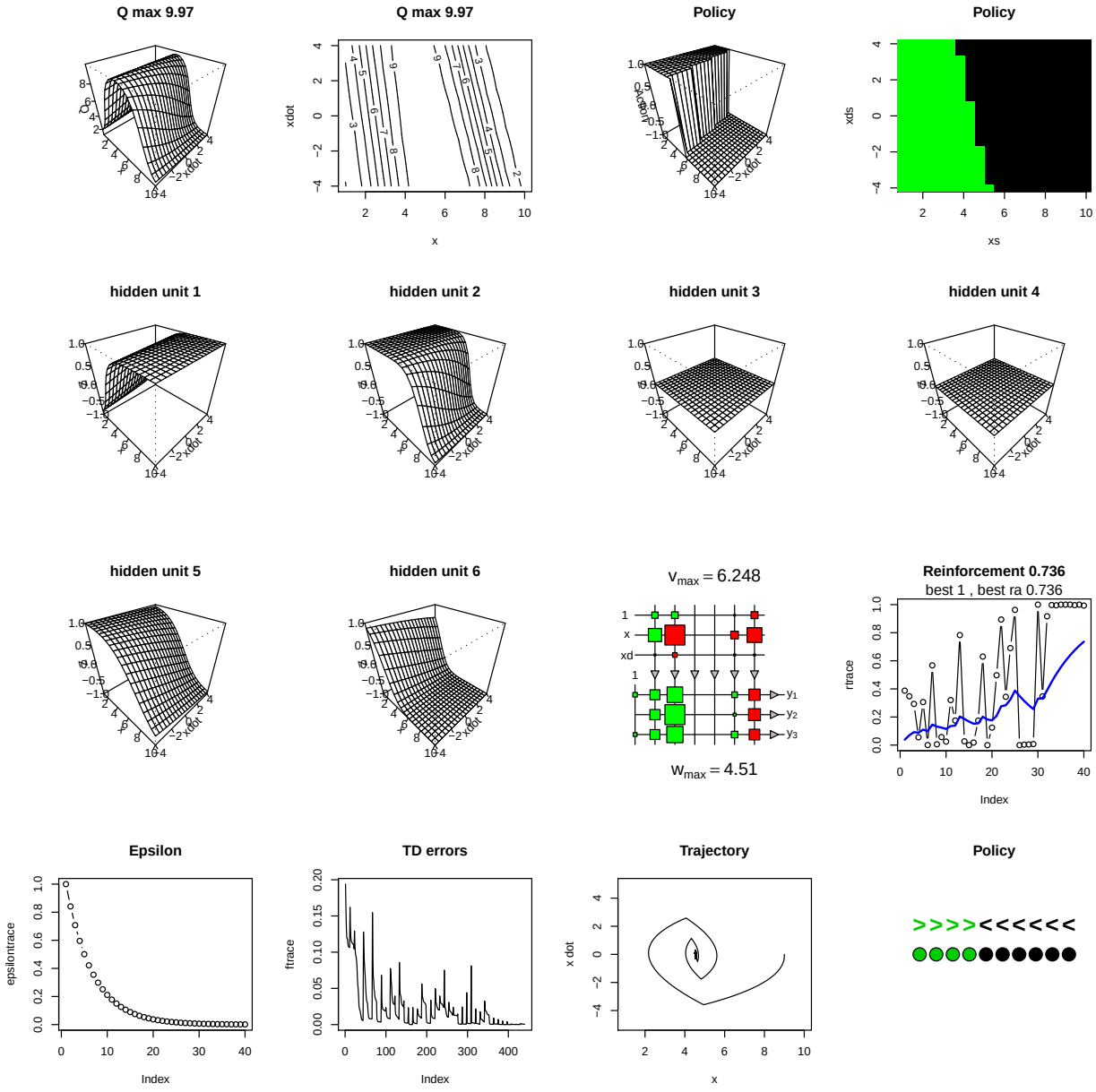

Figure 4: Analysis of 5000 samples and 50 repetitions with a fixed goal

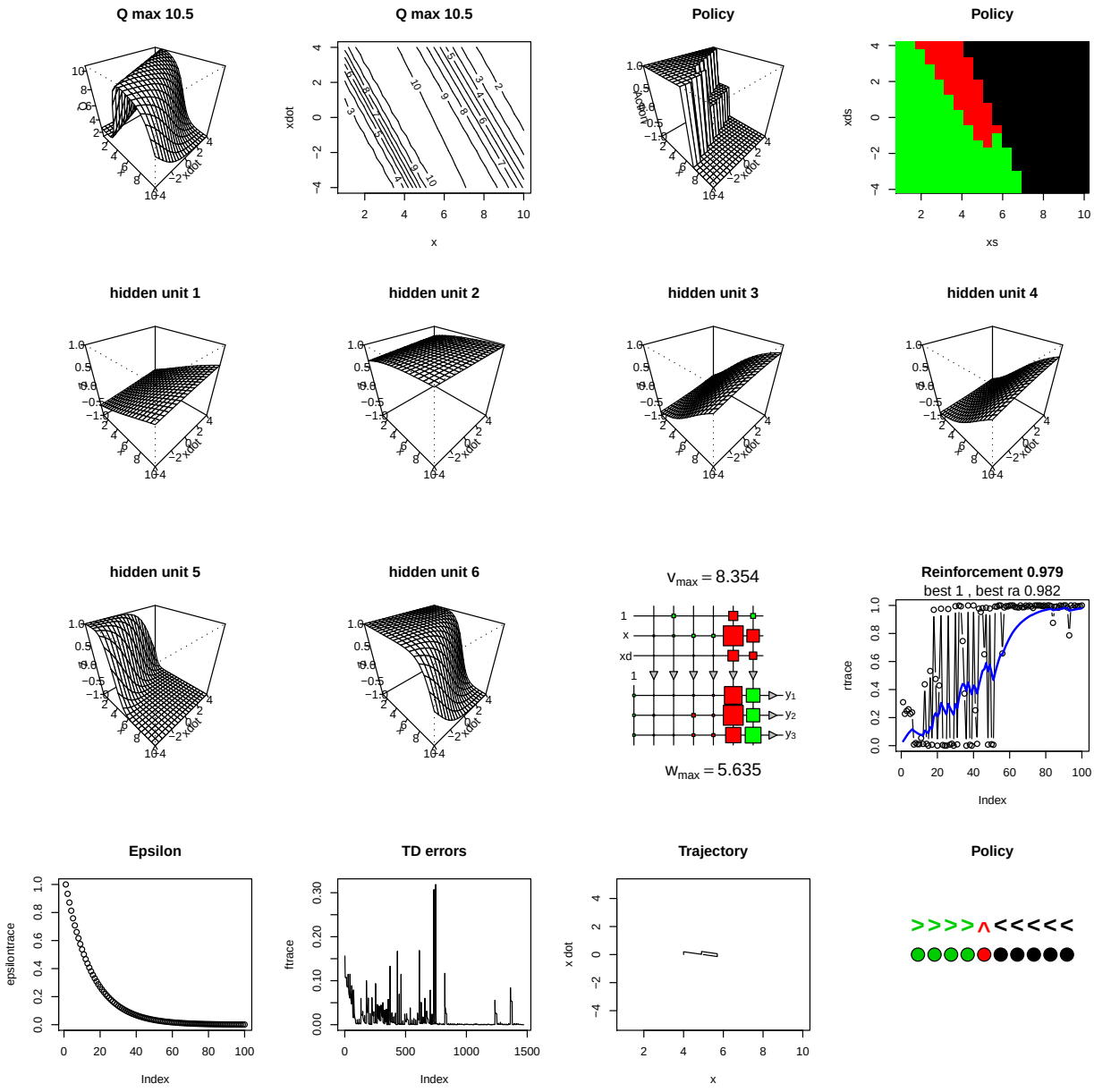

Figure 5: Analysis of a maximum of 14 gradient descent iterations for a fixed goal.

maximum iterations	Q_{max}	best r	best r average	last r average
5	5.74	0.511	0.211	0.005
6	6.07	1.000	0.251	0.005
7	6.53	0.885	0.154	0.006
8	7.59	0.949	0.181	0.006
9	10.8	1.000	0.776	0.646
10	7.56	0.445	0.117	0.007
11	12.0	1.000	0.956	0.722
12	10.2	1.000	0.997	0.829
13	9.97	1.000	0.964	0.787
14	10.5	1.000	0.982	0.979
15	10.4	1.000	0.834	0.557
16	10.6	1.000	0.941	0.875
17	11.2	1.000	0.975	0.802
18	10.4	1.000	0.994	0.850
19	11.4	1.000	0.896	0.809
20	9.98	1.000	0.981	0.783

Table 4: Q and Reinforcements for maximum iterations ranging from 5 to 20

λ	Q_{max}	best r	best r average	last r average
0	10.5	1.000	0.982	0.968
1e-08	11.8	1.000	0.796	0.638
1e-07	12.0	1.000	0.825	0.825
1e-06	10.5	1.000	0.961	0.961
1e-05	10.6	1.000	0.940	0.940
1e-04	10.5	1.000	0.945	0.549
1e-03	8.89	1.000	0.677	0.350
1e-02	5.66	1.000	0.450	0.256
1e-01	1.91	1.000	0.409	0.041

Table 5: Q and Reinforcements for λ ranging from 0 to 0.1

shows the convergence of both the reinforcement and temporal difference errors. Note that the policy is not quite correct in this case.

3.6 Weight penalty discount factor γ

In this section, we varied the weight penalty discount factor parameter γ from 0.05 to 1.0. The results in Table 6 suggest larger values of γ perform best, γ values less than 0.8 performed poorly with a single exception. The analysis in Figure 7 shows convergence of the reinforcement and temporal difference errors.

3.7 Random action probabilityϵ

The parameter ϵ in the range (0,1) determines the probability that a random choice will be taken rather than greedy choice based on the current policy. At $\epsilon = 0$ the algorithm always makes the greedy choice while at $\epsilon = 1$ the algorithm always chooses randomly. Table 7. Most effective range is $0.1 < \epsilon < 0.4$ where we observed consistent results. The two extreme values of ϵ at each end of the table exhibit significant reduction in the reinforcements received.

The network correctly learned the policy for ϵ in the range (0.0001 to 0.7). Figures 8, 9, and 10 show the training information corresponding to ϵ of 0.3, 0.7, and 0.0001.

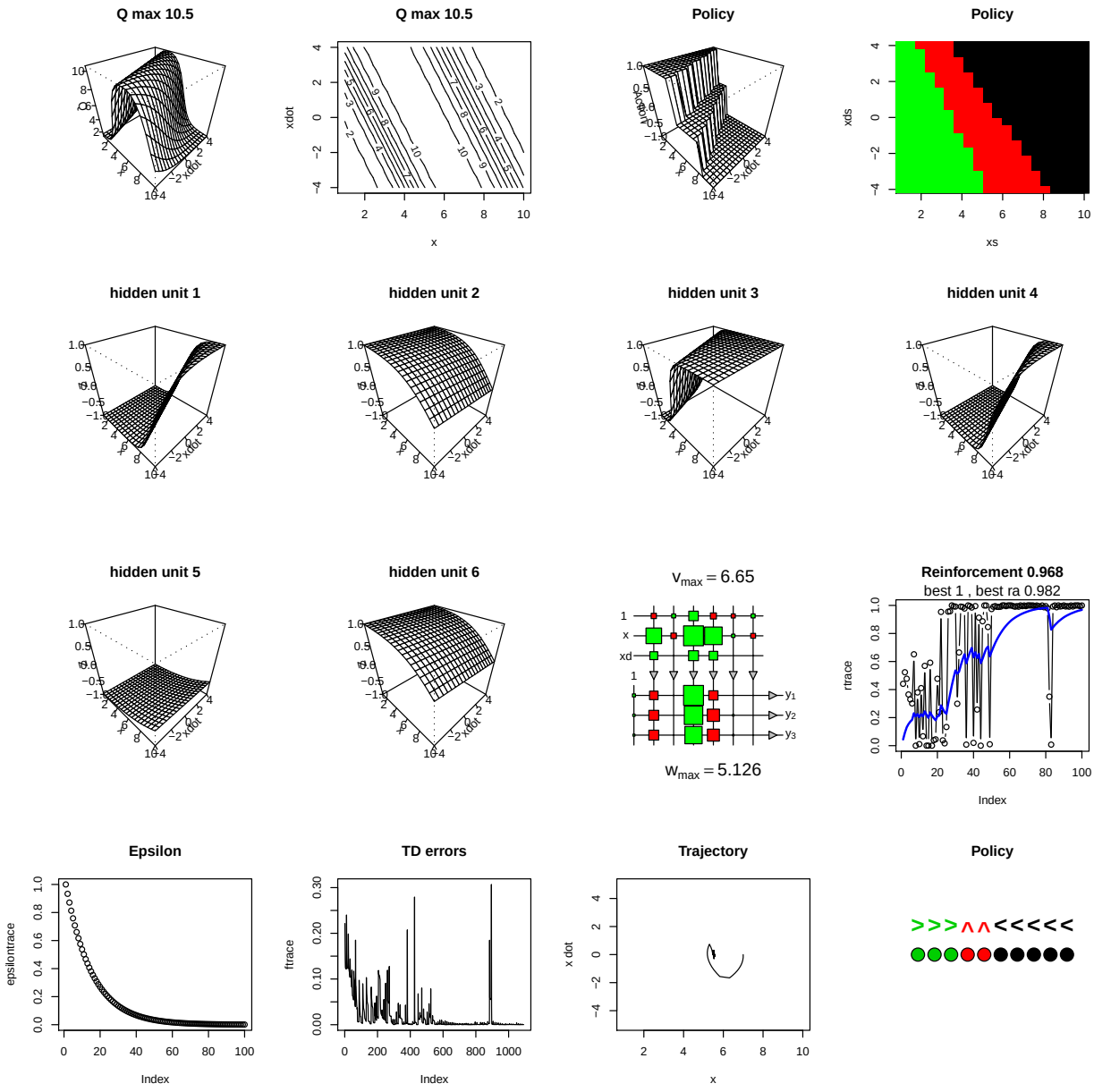

Figure 6: Analysis of the weight penalty factor $\lambda = 0$ for a fixed goal

γ	Q_{max}	best r	best r average	last r average
0.05	1.73	1.000	0.430	0.017
0.10	1.49	0.857	0.235	0.006
0.15	1.74	0.882	0.247	0.058
0.20	1.62	1.000	0.567	0.155
0.25	1.88	0.813	0.256	0.034
0.30	1.82	0.470	0.184	0.004
0.35	1.71	1.000	0.359	0.004
0.40	1.85	0.844	0.164	0.005
0.45	2.13	1.000	0.247	0.120
0.50	2.06	1.000	0.995	0.994
0.55	2.56	1.000	0.219	0.008
0.60	2.69	0.906	0.200	0.005
0.65	3.24	1.000	0.738	0.289
0.70	3.53	0.945	0.227	0.004
0.75	4.38	1.000	0.685	0.685
0.80	5.31	1.000	0.931	0.720
0.85	6.82	1.000	0.997	0.996
0.90	10.4	1.000	0.918	0.700
0.95	20.7	1.000	0.873	0.708
1.00	66.9	1.000	0.842	0.457

Table 6: Q and Reinforcements for γ ranging from 0.05 to 1

ϵ	Q_{max}	best r	best r average	last r average
0.9	10.7	0.758	0.585	0.585
0.8	10.2	0.965	0.808	0.806
0.7	10.1	1.0	0.945	0.945
0.6	10.2	1.0	0.987	0.986
0.5	10.2	1.0	0.989	0.879
0.4	10.1	1.0	0.993	0.990
0.3	10.2	1.0	0.996	0.996
0.2	10.2	1.0	0.993	0.993
0.1	10.0	1.0	0.994	0.994
0.01	11.4	1.0	0.928	0.787
0.001	11.0	1.0	0.534	0.500
0.0001	10.1	1.0	0.900	0.900
0.00001	9.96	1.0	0.608	0.351
0.000001	9.24	1.0	0.689	0.482
0.0000001	9.99	1.0	0.801	0.588
0.00000001	7.81	0.481	0.107	0.005
0	6.13	0.316	0.032	0.004

Table 7: Q and Reinforcements for ϵ ranging from 0 to 1

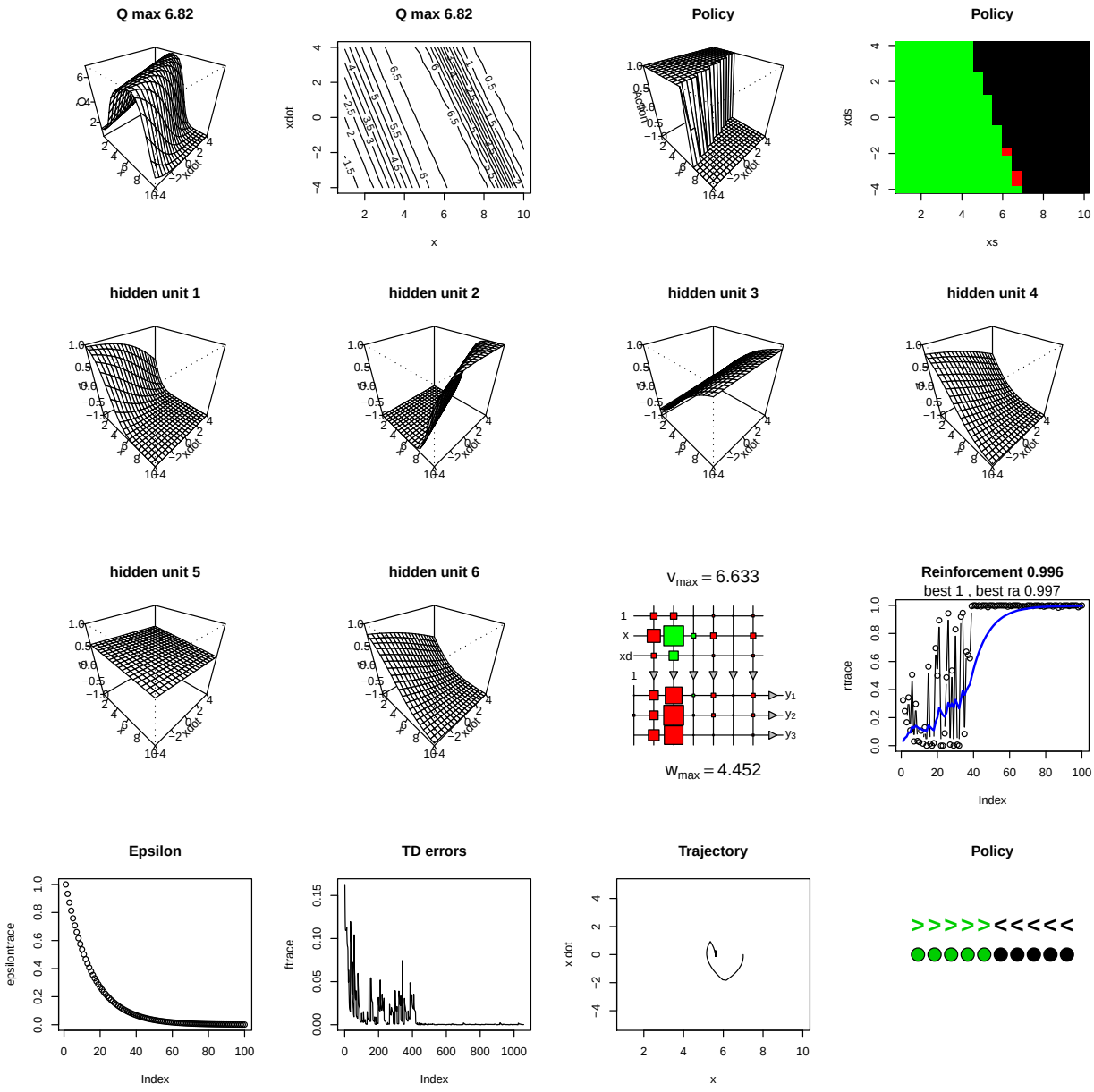

Figure 7: $\gamma = 0.85$

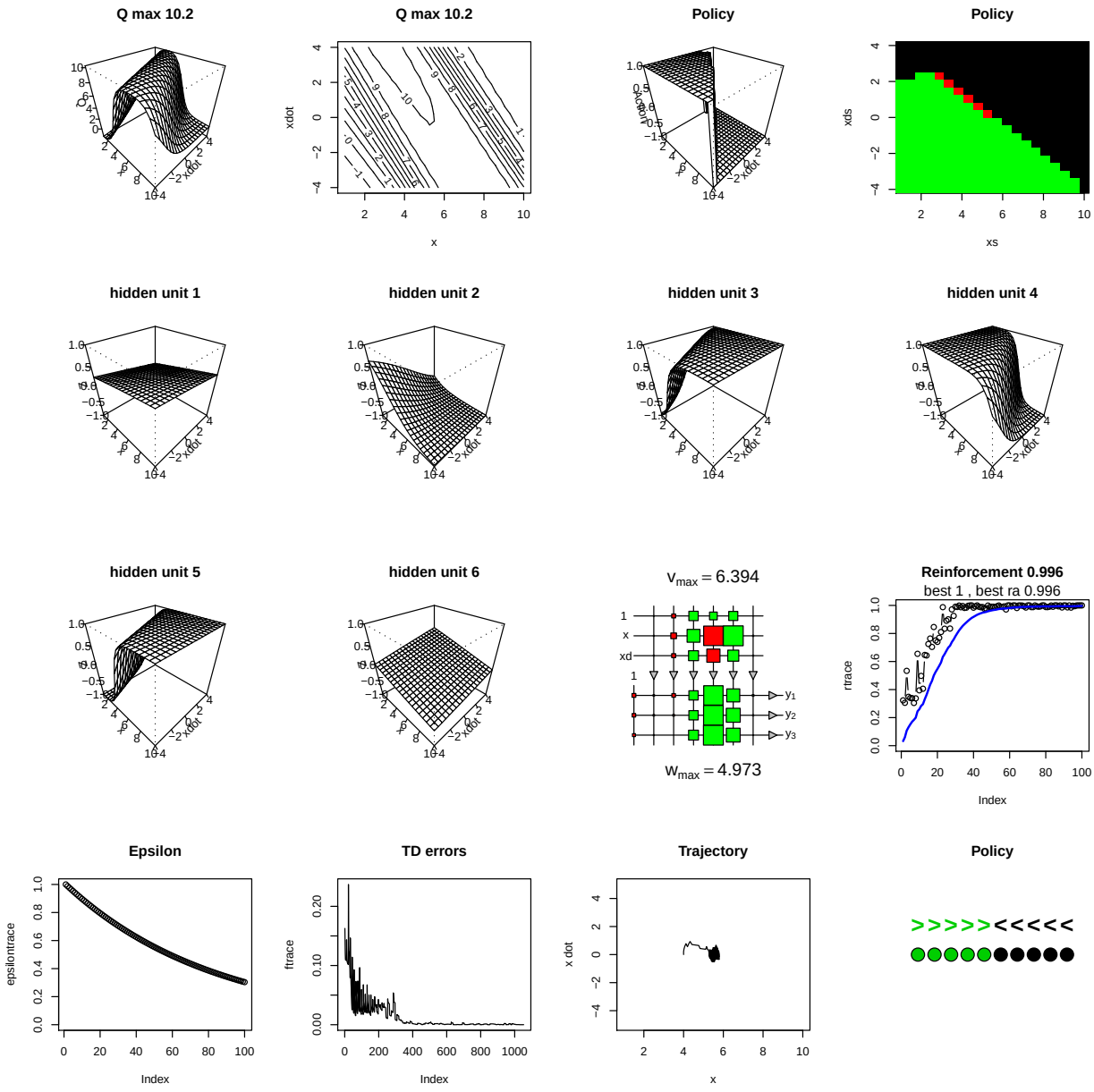

Figure 8: $\epsilon = 0.3$

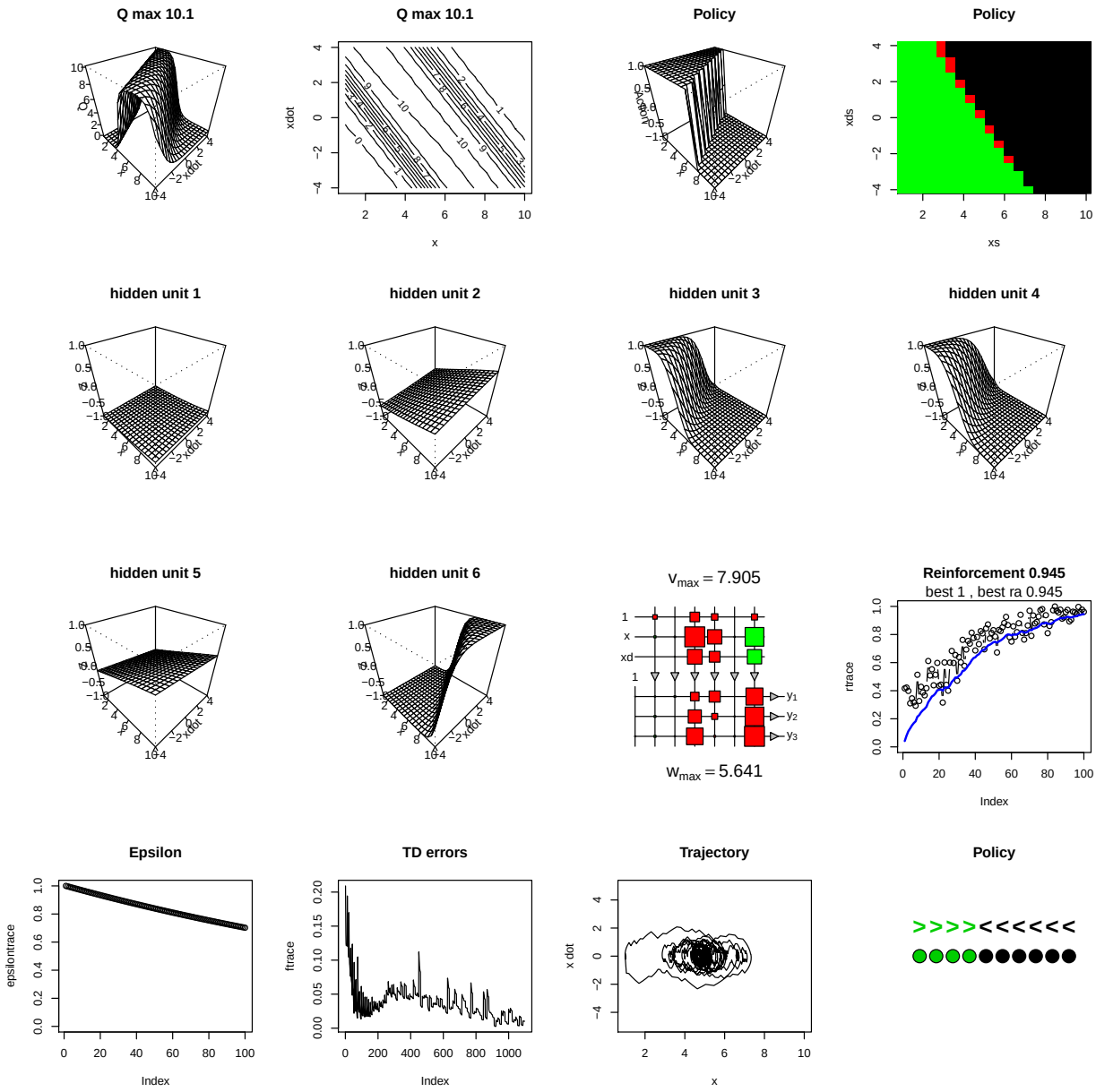

Figure 9: $\epsilon = 0.7$

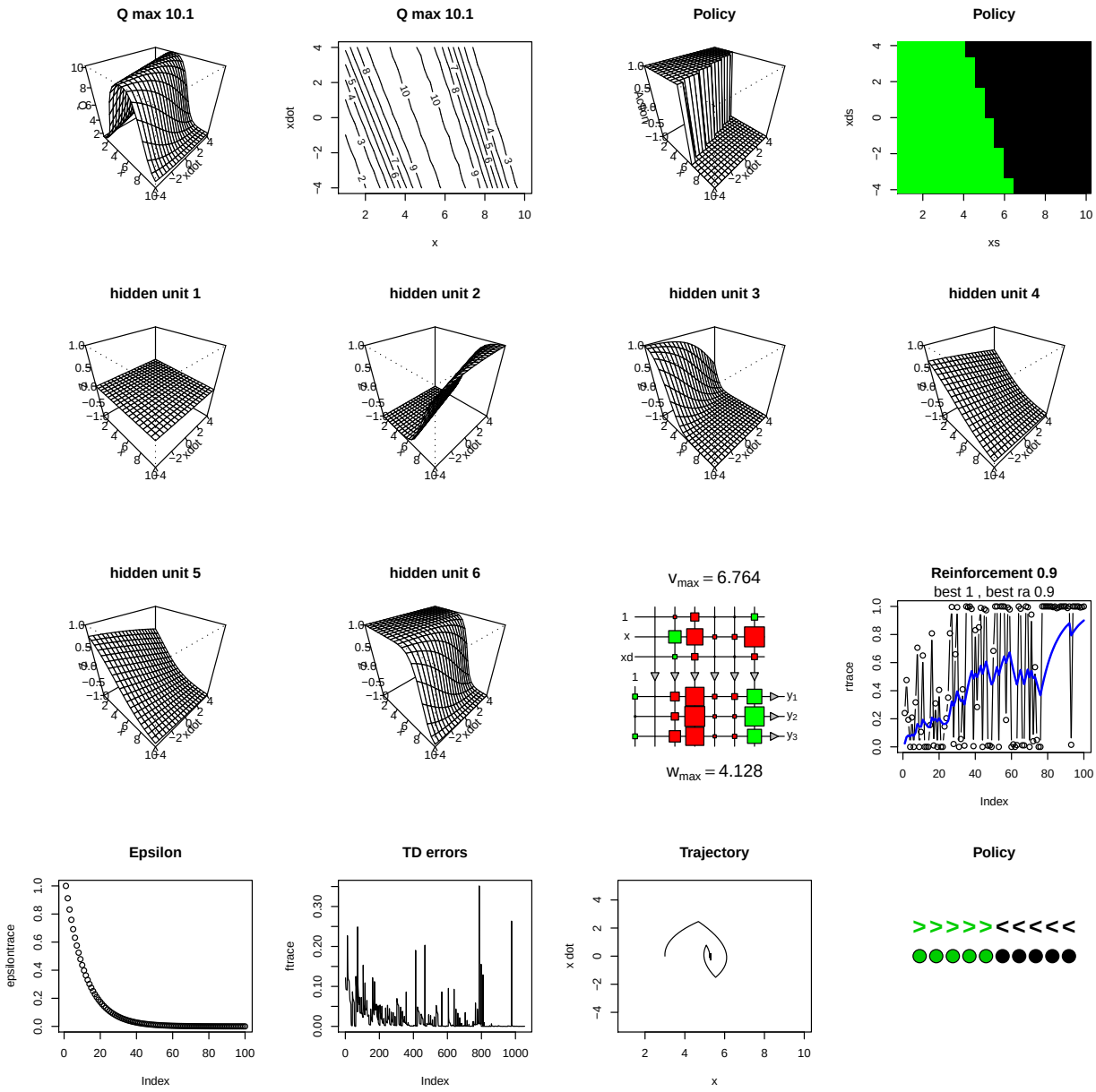

Figure 10: $\epsilon = 0.0001$

4 Goal Study

In the previous section we evaluated the algorithm with a single fixed goal of 5. In this section we varied the goal from 1 to 10 during the training and repeated the tests of the previous section. Figures 11, 12, 13, 14, 15, and 16 show selected results.

5 Conclusions

This was an interesting assignment. In CS440 I had studied reinforcement learning alone, without the neural network component. The amount of variability in the results was not surprising given my previous work with stochastic algorithms.

The tests with variable goals appeared to do much better learning the policies. This probably results form learning when to move left or right rather than how to achieve a specific goal.

References

[1] Bishop, C.M., *Pattern Recognition and Machine Learning*, Springer, 2006.

[2] Sutton, R.S. and Barto, A.G., *Reinforcement Learning: An Introduction*, MIT Press, 1998.

[3] Anderson, C., *Artificial Neural Networks*, `http://www.cs.colostate.edu/~anderson/cs545/Lectures/week8day2/week8day2.pdf`, Colorado State University, 2009.

[4] Anderson, C., *CS 545 Assignment 7*, `http://www.cs.colostate.edu/~anderson/cs545/assignments/assignment7.html`, Colorado State University, 2009.

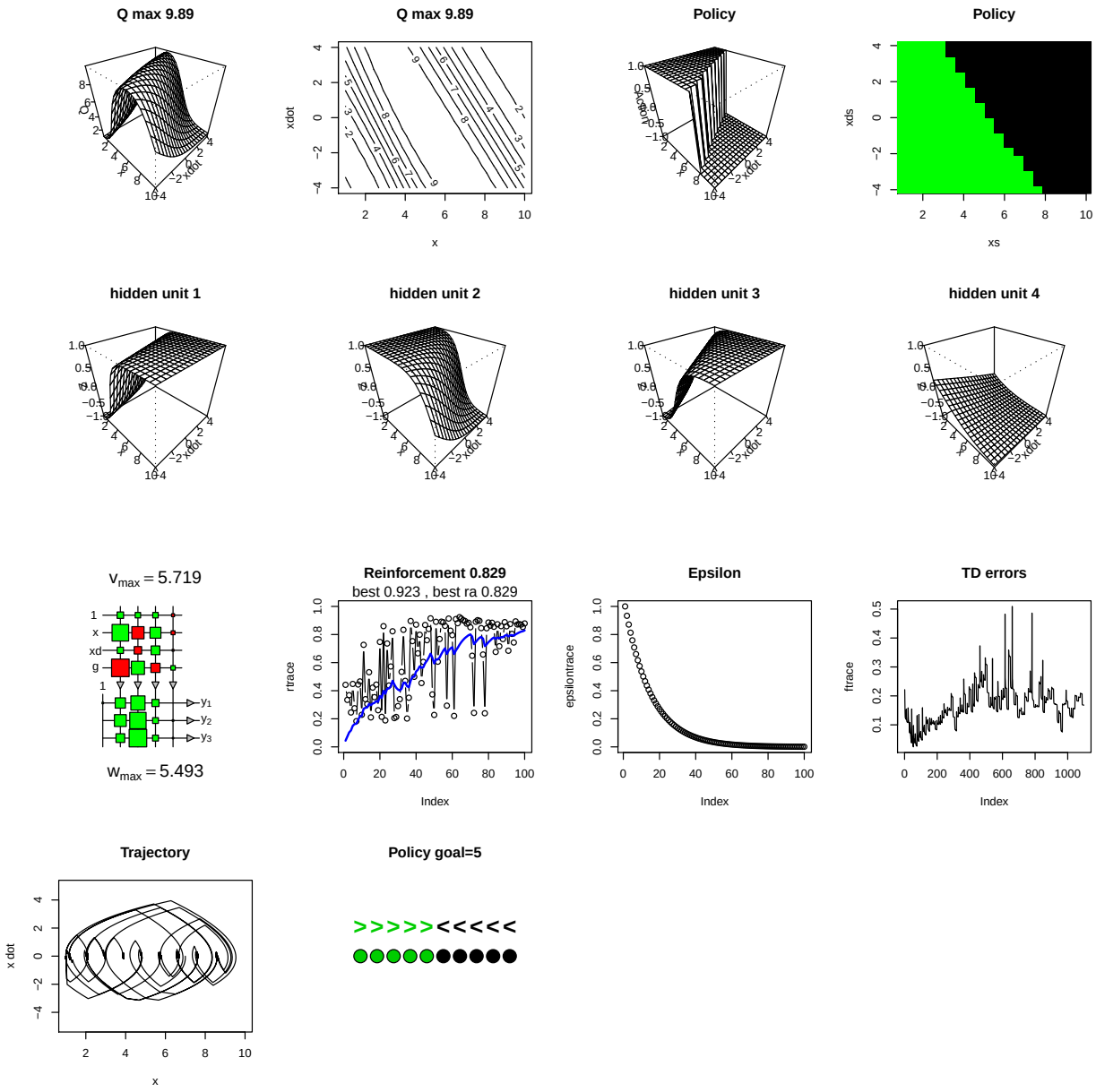

Figure 11: 4 hidden units with goal varying from 1 to 10

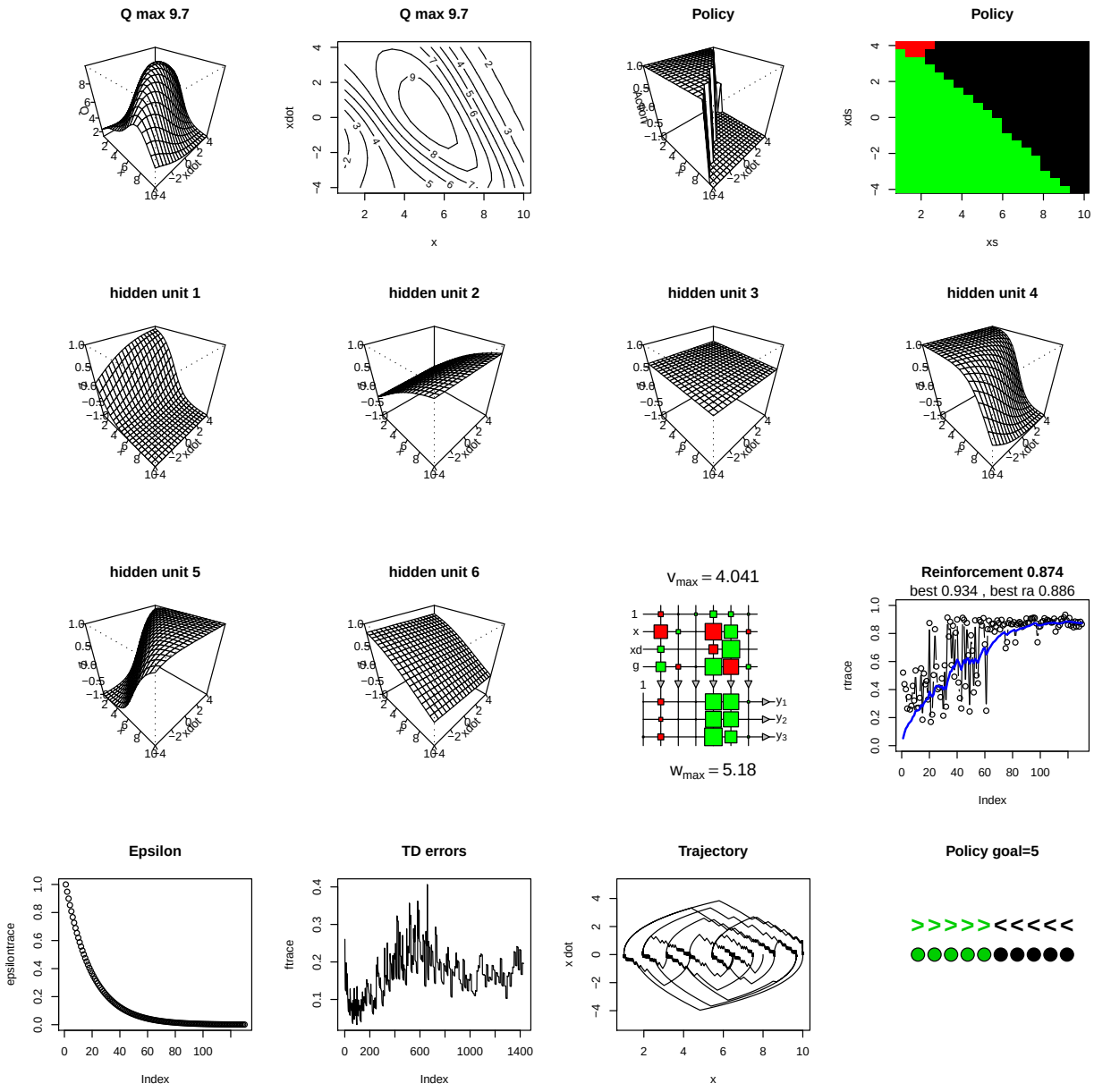

Figure 12: 130 repetitions with goal varying from 1 to 10

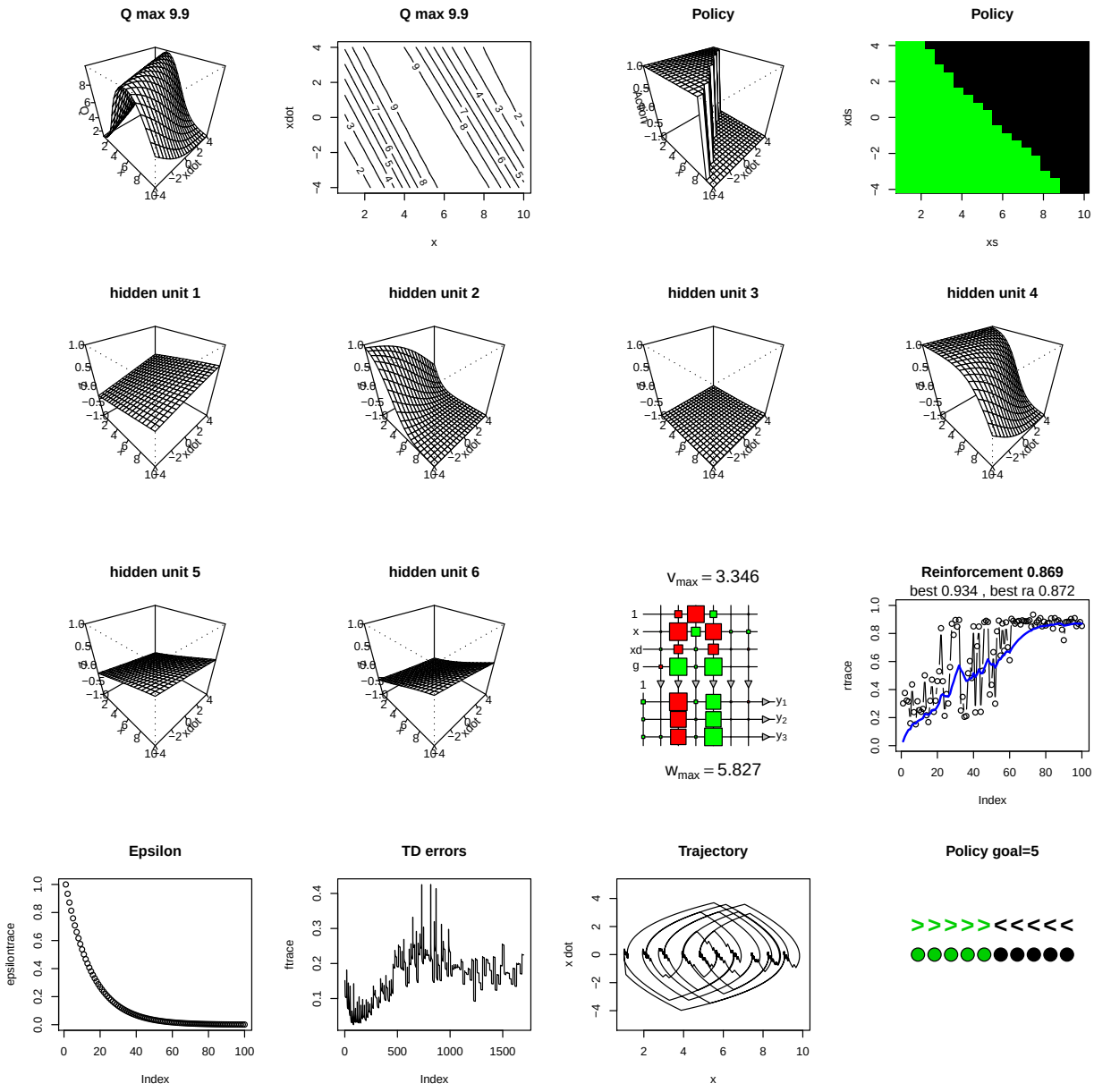

Figure 13: 16 iterations of gradient descent with goal varying from 1 to 10

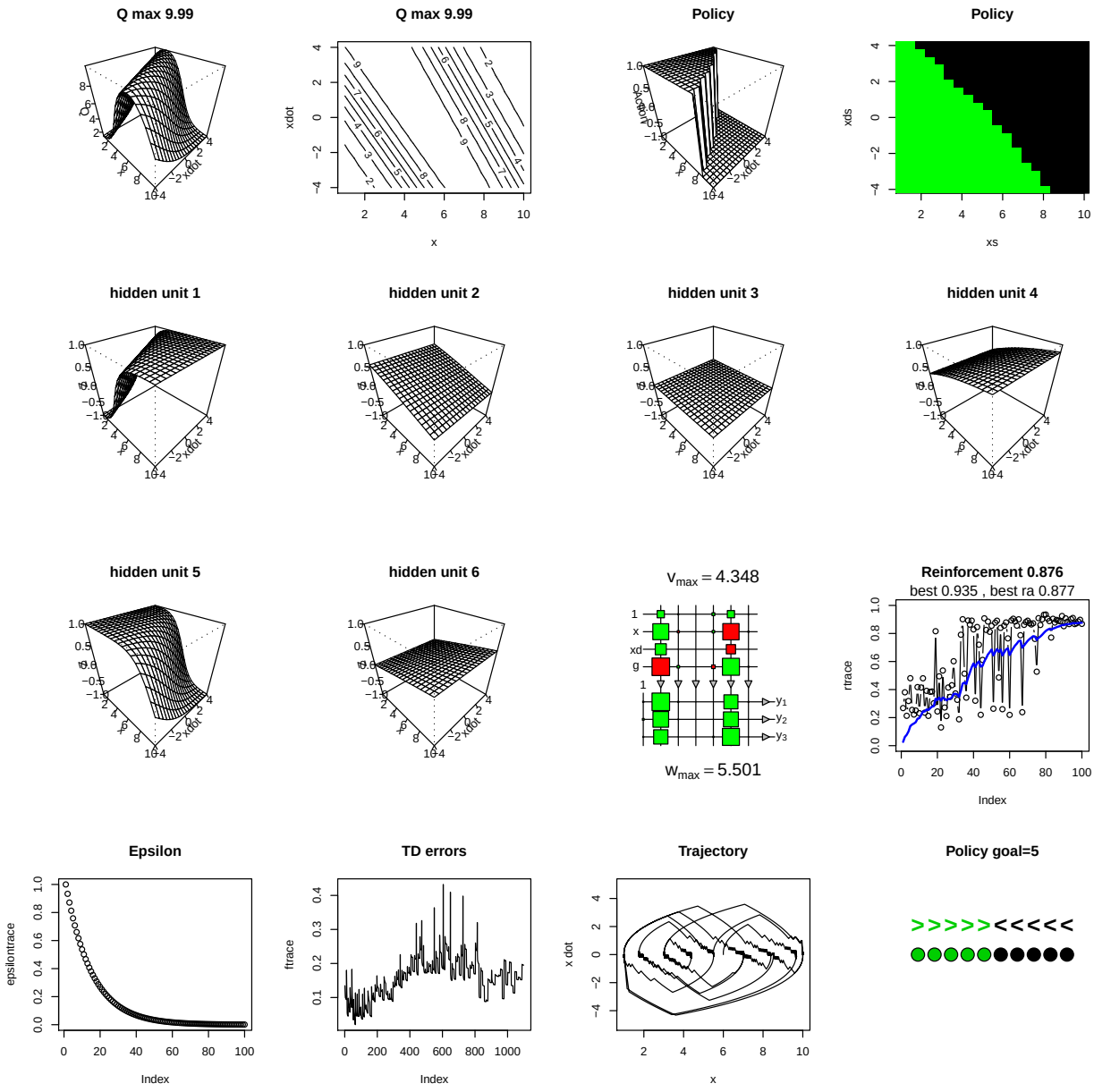

Figure 14: $\lambda = 1e - 05$ with goal varying from 1 to 10

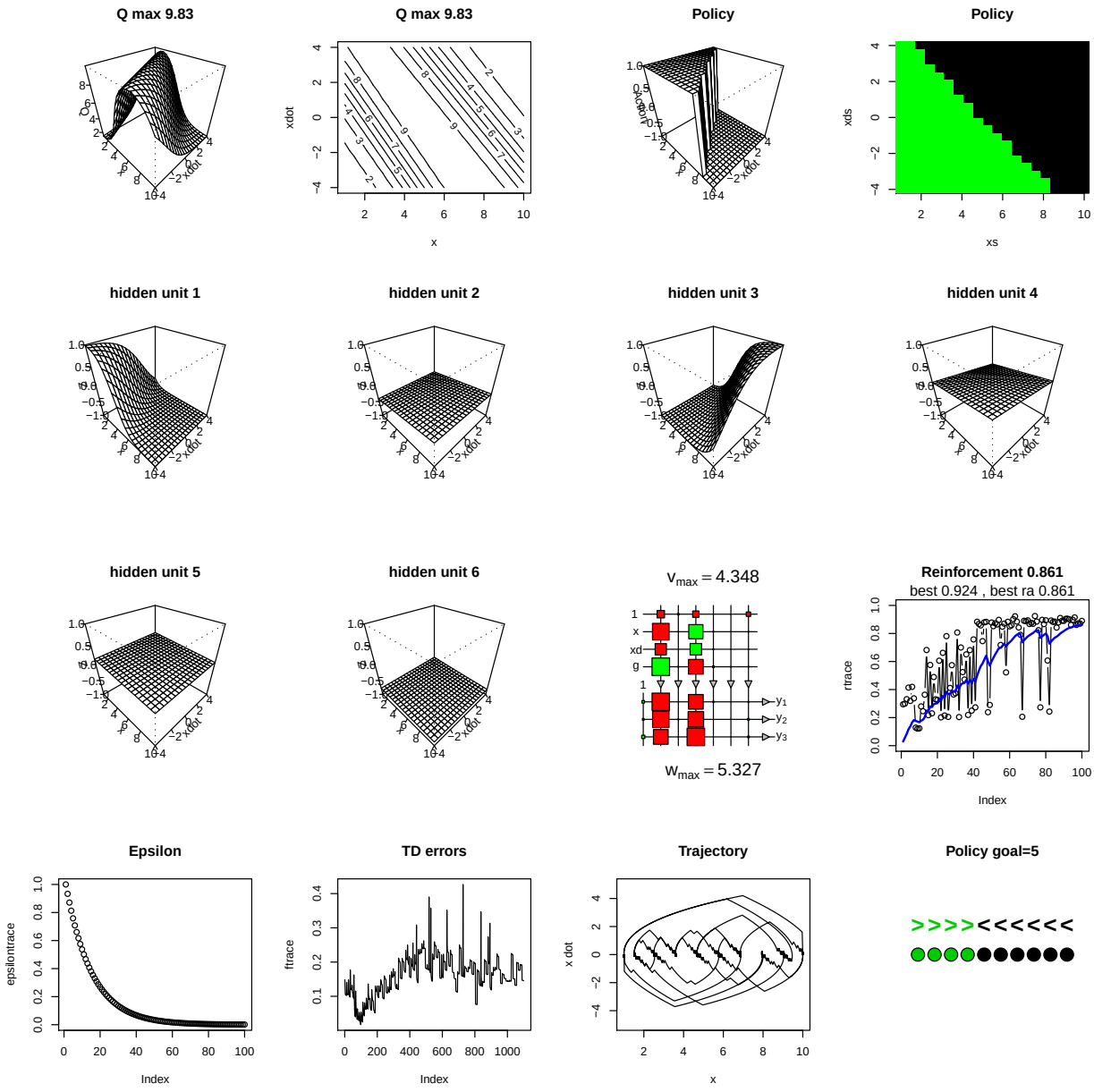

Figure 15: $\gamma = 0.90$ with goal varying from 1 to 10

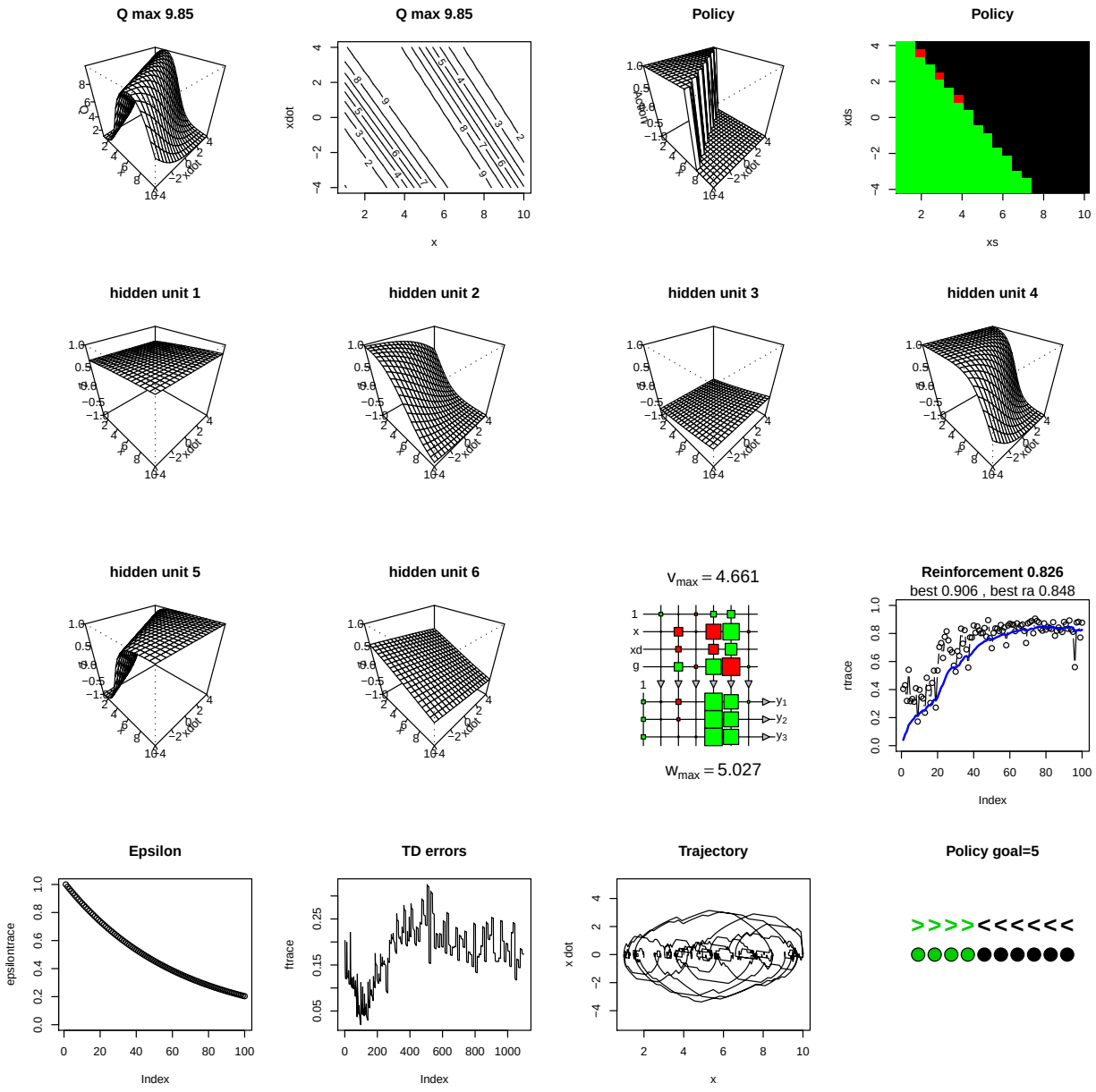

Figure 16: $\epsilon = 2e - 1$ with goal varying from 1 to 10