…As R can struggle with the speed necessary for reinforcement learning (which typically relies on large numbers of iterating behaviour), it also provided a good chance to crack out some C++ code using the always excellent Rcpp package, which is always worth practicing.

In this first example of Reinforcement Learning in R (and C++), we're going to train our computers to play Noughts and Crosses (or tic tac toe for Americans) to at least/super human level.

Let's get started with the libraries we'll need. I want to stick to base for speed here, as well as obviously Rcpp. In theory you can easily generalise all the code here to any size board, but I only have tested in with 3×3 boards so YMMV.

```
#will use ggplot
#everything else Ive used base of listed packages
library(ggplot2)
#Rcpp for some vroom vroom
library(Rcpp)

#in theory this stuff should work for boards of any size
#but I haven't tested that
board_cols = 3
board_rows = 3
squares <- board_cols * board_rows
```

The very first thing we'll want to do is find a way to store the information in a game state and convert between this, and a human readable form.

```
#function to plot boards in a human readable way
#not generalised to all board sizes but easy enough to
plot_board <- function(string) {
  pieced <- rep("", length(string))
  pieced[which(string == 1)] <- "x"
  pieced[which(string == -1)] <- "o"
  pieced[which(string == 0)] <- "*"
  board <- gsub(" \\|$", "", paste(pieced, "|", collapse = " "))
  board_lines <- gsub("(. \\| . \\| . )\\|( . \\| . \\| . )\\|( . \\| . \\| .)",
                      "\n \\1\n-----------\n\\2\n-----------\n\\3",
                      board
                      )
  return(writeLines(board_lines))
}
```

Next we're going to want to find every possible state we might encounter so we can test for any exceptions. I'm storing strings as a list of 9 0s (unused), 1s (Xs) and -1s (Os) representing the squares 1->9 from the top left corner.

It's simple and fast enough to do this with a quick R function

```
#get all possible boards
possible_boards <- gtools::permutations(
  board_cols, squares,
  c(-1,0,1),
  repeats.allowed = TRUE
  )

#can only have a sum of 1 or 0
possible_boards <- possible_boards[which(rowSums(possible_boards) %in% c(0,1)),]

#plot a random example
```

```
plot_board(c(1,0,0,-1,0,0,0,0,1))
```

```
##
##  x | * | *
## -----------
##  o | * | *
## -----------
##  * | * | x
```

Now we have the representations of any possible board, we want to find a way to store this is a more compressed format as a hash. I originally wrote a pretty quick function to do this in R and then threw up a quick one underneath compiled in Rcpp for comparison.

```
#get a unique hash for each board
calc_hash <- function(board) {
  hash <- 0
  for(piece in seq(squares)) {
    hash <- (hash*board_cols) + board[piece] + 1
  }
  return(hash)
}

#and the equivalent in Cpp
cppFunction('int calc_hashCpp(NumericVector board, int squaresize) {
    //need to init vals in C++
    int hash = 0;
    int boardsize = squaresize * squaresize;

    //C++ for loops have start, end, and by
    for (int i=0; i <= boardsize - 1; ++i) {
      hash = (hash * squaresize) + board[i] + 1;
    }

    //always have to declare a return
    return hash;
}')

#get a list of all the possible hashes
hashes <- lapply(purrr::array_tree(possible_boards, margin = 1),
                 calc_hashCpp, squaresize = 3)

#should all be unique
which(duplicated(hashes))
```

```
## integer(0)
```

In order to play noughts and crosses, we then need some way for a game to end. An easy way to check this is when our board string (0s,1s,and-1s) add up to 3/-3 along any row, column or diagonal.

```
#first we need a function to check when a game has been won
cppFunction('int check_winnerCpp(NumericVector board) {
  int winner = 0;

  int vec_length = board.size();
  int square_size = sqrt(vec_length);

  //check rows and columns for a winner
  for (int i=0; i <= square_size - 1; ++i) {
    //check row i
```

```
  NumericVector row_squares = NumericVector::create(0,1,2);
  row_squares = row_squares + (square_size * i);
  NumericVector row_elements = board[row_squares];
  int row_sum = sum(row_elements);
  if(abs(row_sum) == square_size) {
    if(row_sum > 0) {
      winner = 1;
    } else {
      winner = -1;
    }
  }

  //check col i
  NumericVector col_squares = NumericVector::create(0,3,6);
  col_squares = col_squares + i;
  NumericVector col_elements = board[col_squares];
  int col_sum = sum(col_elements);
  if(abs(col_sum) == square_size) {
    if(col_sum > 0) {
      winner = 1;
    } else {
      winner = -1;
    }
  }
}

//check the diagonalsNumericVector
NumericVector rising_diag_squares = NumericVector::create();
NumericVector falling_diag_squares = NumericVector::create();
for (int i=0; i <= square_size - 1; ++i) {
  int rising_diag_square = (square_size * i) + i;
  rising_diag_squares.push_back(rising_diag_square);

  int falling_diag_square = (square_size - 1) * (i+1);
  falling_diag_squares.push_back(falling_diag_square);
}

NumericVector rising_diag_elements = board[rising_diag_squares];
NumericVector falling_diag_elements = board[falling_diag_squares];
int rising_sum = sum(rising_diag_elements);
int falling_sum = sum(falling_diag_elements);

if(abs(falling_sum) == square_size) {
  if(falling_sum > 0) {
    winner = 1;
  } else {
    winner = -1;
  }
}
if(abs(rising_sum) == square_size) {
  if(rising_sum > 0) {
    winner = 1;
  } else {
    winner = -1;
  }
}
```

```
  //return the winner
  //0 for no winner, 999 for draw
  return winner;
}')
```

We can then apply this function to every possible board and find the ones that indicate a winning state. We also init a data frame containing all possible boards, their hash, and their 'value' (0 for all for now, more on this later). Finally, I plot the first one in this set just because why not?

```
#find which boards are winning positions
winning <- purrr::map(purrr::array_tree(possible_boards, margin = 1),
check_winnerCpp)

#going to create a df to store the values of all moves
moves_df <- data.frame(hash = unlist(hashes),
                       value = 0,
                       winning = unlist(winning))

#store all boards as a list
#purrr::aray_tree is a really nice way to convert matrix to lists
moves_df$board = purrr::array_tree(possible_boards, margin = 1)

#plot the first board just why not
plot_board(unlist(moves_df$board[1]))
```

```
##
##   o | o | o
##  -----------
##   o | * | x
##  -----------
##   x | x | x
```

As we can see, we still have some impossible boards here. This particular board will never occur in actual play because X wins before O can make a move to complete the top row. It doesn't matter, but useful to keep in mind for a plot later.

We then need a function telling the computer how to make a move. For this post we're going to use what's called 'E (epsilon)-greedy' selection. A computer has a parameter epsilon such that

$$\begin{cases}
v &\text{if } \epsilon \leq \rho\\
V_{max} &\text{if } \epsilon > \rho\\
\end{cases}$$

if epsilon is greater than a random number rho, the computer makes the most valuable choice possible. It chooses whatever it thinks (rightly or wrongly) will lead to the best outcome. This is called *exploitation*. If epsilon is less than or equal to rho, the computer randomly chooses any possible action v. This is known as *exploration* to test any possibly rewarding but unvalued paths.

(I may have gotten epsilon the wrong way round here. It really doesn't matter at all.)

Let's implement this in C++

```
cppFunction('int choose_moveCpp(NumericVector epsilon, NumericVector values) {
  //random number to decide if computer should explore or exploit
  NumericVector random_number = runif(1);
  int move_choice = 0;
  NumericVector choices = NumericVector::create();

  //exploit the best move
```

```
  if(epsilon[0] > random_number[0]) {
    double max = Rcpp::max(values);
    std::vector< int > res;

    int i;
    for(i = 0; i < values.size(); ++i) {
      if(values[i] == max) {
        res.push_back(i);
      }
    }
    IntegerVector max_indexes(res.begin(), res.end());
    if(max_indexes.size() > 1) {
      std::random_shuffle(max_indexes.begin(), max_indexes.end());
      move_choice = max_indexes[0] + 1;
    } else {
      move_choice = max_indexes[0] + 1;
    }
   //explore all moves randomly
  } else {
    int potential_choices = values.size();
    choices = seq(1, potential_choices);
    std::random_shuffle(choices.begin(), choices.end());
    move_choice = choices[0];
  }

  return move_choice;
}')
```

We also want a little helper func to find all the possible hashes so we can look up which moves a computer can make before choosing between them.

```
#find all possible next moves
get_next_hashes <- function(board, piece) {
  unused <- which(board == 0)

  next_boards <- lapply(unused, function(x, piece) {
    board[x] <- piece
    return(board)
  }, piece = piece)
  #get the hashes of the next boards
  hashes <- lapply(next_boards, calc_hashCpp, squaresize = 3)
}
```

Finally, we need to reward the computer for making good actions, and punish it for making bad ones. We'll do this using Temporal Difference (TD) error learning.

The computer looks at how good an end point was (for noughts and crosses this can be a win, lose, or draw) and then decides if that outcome is better or worse than it was expecting. It then re-evaluates its beliefs about the choices it made to lead to that end state. It can be formulated as

$$V_{state} = V_{state} + TD\ error \cdot scalar$$
the scalar here is the *learning rate* of the computer. Do we want it to forget everything it new about the world seconds earlier and take only the most recent information (1), or update it's beliefs very slowly (~0). We'll refer to this as lr in subsequent equations.

The TD error itself is calculated as

$$TD\ error = (\gamma \cdot reward - V_{state})$$
Where gamma acts to make sure we don't overfit too far back into the past. It reduces the reward as you go

further back and is set between 0 and 1. The reward here will (e.g.) be 1 if the computer has just won with it's latest move, otherwise it will be the value of the state the computer might move into.

Putting these together we get

\[V_{state} = V_{state} + lr \cdot (\gamma \cdot V_{state+1} – V_{state}) \]

Let's implement this using Rcpp

```
#function to feed reward back to the agent based on results
cppFunction('NumericVector backfeed_rewardCpp(NumericVector values, double
reward, double learning_rate, double gamma) {
  int states = values.size();
  NumericVector new_values = NumericVector::create();

  //go from last state backwards
  for( int state = states-1; state >= 0; state--) {
    double new_value = values[state] + learning_rate * ((gamma * reward) -
values[state]);
    new_values.push_back(new_value);
    //recurse the reward
    reward = new_value;
  }
  return new_values;
}')
```

Now we can start actually playing games! I wrote out a long function in R to play through the various bits. It surely could be refactored a little more concisely but it works for now and I was getting tired by this point.

We first add two functions (one to make moves/play the game, and one to update the values using the formula above) then put it all into to one uber-function

```
#function to choose and implement computer moves
computer_move <- function(piece, board, epsilon) {
  #get potential moves
  potential_move_hashes <- get_next_hashes(board, piece)
  #get the values of the potential moves
  potential_move_vals <- moves_df$value[
    unlist(lapply(potential_move_hashes, function(x) which(moves_df$hash ==
x)))]
  #choose move based on rewards
  player_move <- choose_moveCpp(epsilon, potential_move_vals)
  #update the board with the new move
  updated_board <- unlist(moves_df$board[
    moves_df$hash == unlist(potential_move_hashes)[player_move]])
  return(updated_board)
}

#function to get the values for each state based on the reward
update_move_vals <- function(player1_reward, player2_reward,
                             player1_hashes, player2_hashes,
                             learning_rate,gamma) {
  player1_newvals <- backfeed_rewardCpp(moves_df$value[
    unlist(lapply(player1_hashes, function(x) which(moves_df$hash == x)))],
    player1_reward, learning_rate, gamma)
  player2_newvals <- backfeed_rewardCpp(moves_df$value[
    unlist(lapply(player2_hashes, function(x) which(moves_df$hash == x)))],
    player2_reward, learning_rate, gamma)
  new_vals <- list(player1_newvals, player2_newvals)
```

```
  return(new_vals)
}

#function to get two computers to play each other
play_game_computers <- function(player1_epsilon,
                                player2_epsilon,
                                learning_rate, gamma) {
  #init board
  board <- rep(0, squares)
  winner <- 0
  moves <- 0
  #init hash storage
  player1_hashes <- c()
  player2_hashes <- c()

  #keep moving until game is over
  while(winner == 0 & moves < 9) {
    #iterate moves
    moves <- moves + 1
    #player 1 moves
    board <- computer_move(1, board, player1_epsilon)
    player1_hashes <- append(calc_hashCpp(board, board_cols), player1_hashes)
    winner <- check_winnerCpp(board)

    #same for player 2
    if(winner == 0  & moves < 9) {
      moves <- moves + 1
      board <- computer_move(-1, board, player1_epsilon)
      player2_hashes <- append(calc_hashCpp(board, board_cols), player2_hashes)
      winner <- check_winnerCpp(board)
    }
  }

  #update policies
  if(winner == 1) {
    message <- "x wins!"
    new_vals <- update_move_vals(1, 0, player1_hashes, player2_hashes,
                                 learning_rate, gamma)
  } else if(winner == -1) {
    message <- "o wins!"
    new_vals <- update_move_vals(0, 1, player1_hashes, player2_hashes,
                                 learning_rate, gamma)
  } else {
    message <- "draw!"
    new_vals <- update_move_vals(0.1, 0.5, player1_hashes, player2_hashes,
learning_rate, gamma)
  }
  #push the values back into the dictionary data frame
  moves_df$value[unlist(lapply(player1_hashes, function(x) which(moves_df$hash
== x)))] <<- new_vals[[1]]
  moves_df$value[unlist(lapply(player2_hashes, function(x) which(moves_df$hash
== x)))] <<- new_vals[[2]]
  return(message)
}
```

So that the computer can learn the value of moves, we first want to run this on a training epoch. We'll get the computer to play 100000 games against itself with an epsilon < 1 so that it explores the game state and learns by reinforcement. We'll then plot the values it's learn for all moves based upon if they are winning or

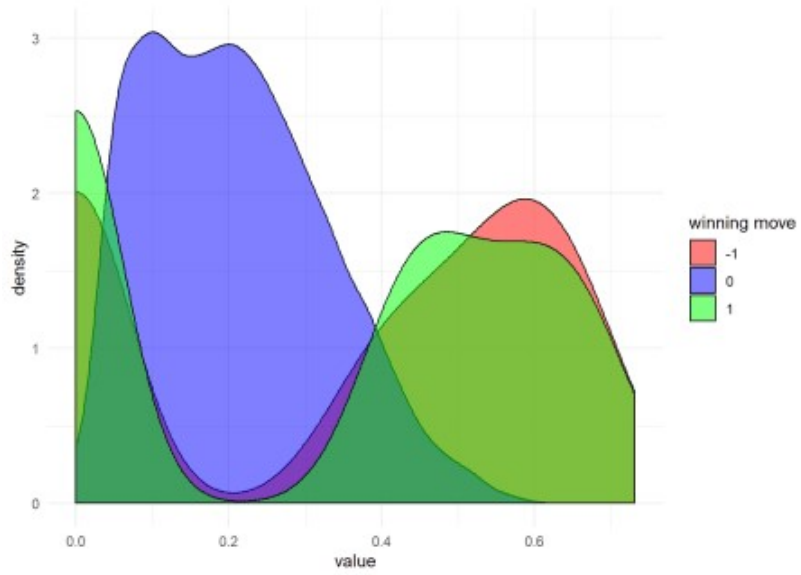
not.

```
#test on 10000 games with a little randomness thrown in
train <- purrr::rerun(100000, play_game_computers(0.8, 0.8, 0.35, 0.9))

#test how fast our function is
microbenchmark::microbenchmark(play_game_computers(0.8, 0.8, 0.35, 0.9), times =
1000)

## Unit: microseconds
##                                        expr   min      lq     mean median
##  play_game_computers(0.8, 0.8, 0.35, 0.9) 838.7 1061.05 1352.258 1222.2
##       uq    max neval
##  1361.45 4548.4  1000

#plot the updated values of moves
p1 <- ggplot(moves_df, aes(x = value, group = as.character(winning))) +
  geom_density(alpha = 0.5, aes(fill = as.character(winning))) +
  scale_fill_manual(values = c("red", "blue", "green"), name = "winning move") +
  theme_minimal()
p1
```



Thankfully the computer has learned that winning moves are more valuable than non-winning moves! The reason there are peaks at 0 is because these are ‘winning’ moves that are impossible as referenced nearer the top of the post.

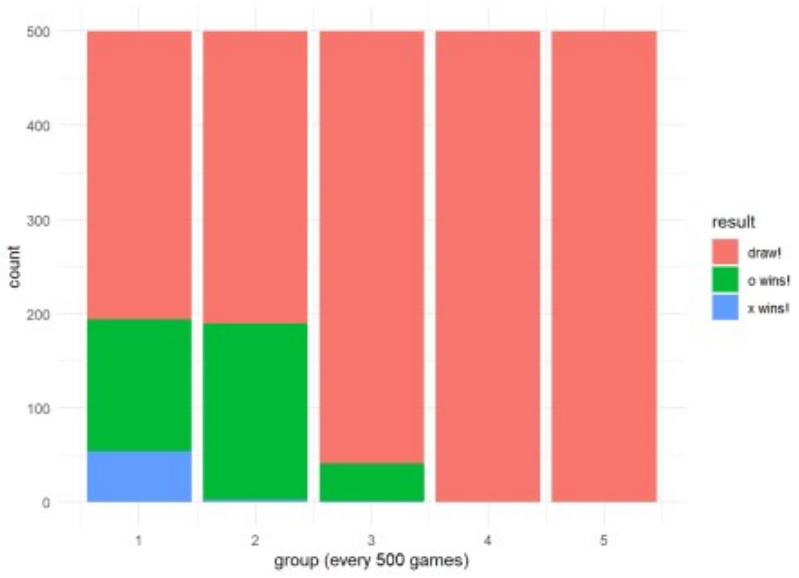
We’ll then run 2500 testing games where the computer is trying to play optimally. Noughts and crosses is a solved game. Unless a play chooses a non-optimal move, the game should end in a draw. Let’s see what proportion actually do end in a draw by grouping every 500 games of the testing set.

```
#run on an extra 2500 games with no exploration (just exploit)
test <- purrr::rerun(2500, play_game_computers(1, 1, 0.35, 0.9))

#group by each 500 games
test_df <- data.frame(result = unlist(test),
                      group = rep(1:5, each = 500))

#plot percentage of games that are drawn
p2 <- ggplot(test_df, aes(x = group, fill = result)) +
  geom_bar(stat = "count") +
  labs(x = "group (every 500 games)") +
  theme_minimal()
```

```
p2
```



And it seems like the computer learns after a final bit of optimisation to always draw! hooray!!

Finally, because obviously this post wouldn't be complete without human testing, I wrote a quick and dirty function to play a game against the now proficient computer. Enjoy below!!

```
player_move <- function(board){
  #find free spaces a move can be made into
  free_spaces <- which(board == 0)
  cat("Please move to one of the following board spaces: [", free_spaces,
"]\n")
  #user input
  submitted_move <- as.integer(readline(prompt = ""))
  #need valid input
  while(!submitted_move %in% free_spaces) {
    if(submitted_move == 0) {
      break
    } else {
      cat("Illegal Move! Please move to one of the following board spaces: [",
free_spaces,  "] or press 0 to quit\n")
      submitted_move <- as.integer(readline(prompt = ""))
    }
  }
  #return move
  return(submitted_move)
}

#only need a computer epsilon and which piece (turn order)
play_game_human <- function(human_piece, computer_epsilon = 1) {
  board <- rep(0, 9)
  moves <- 0
  winner <- 0

  #play the game as before but with a human player
  if (human_piece == 1) {
    while (winner == 0 & moves < 9) {
      moves <- moves + 1
      plot_board(board)
      human_move <- player_move(board)
```

```
      if (human_move == 0) {
        break
      } else {
        board[human_move] <- human_piece
      }
      i <<- board
      j <<- board
      winner <- check_winnerCpp(board)
      if (winner == 0 & moves < 9) {
        moves <- moves + 1
        piece <- human_piece * -1

        board <- computer_move(-1, board, computer_epsilon)
        winner <- check_winnerCpp(board)
      }
    }
  } else {
    while (winner == 0 & moves < 9) {
      moves <- moves + 1
      piece <- human_piece * -1

      board <- computer_move(-1, board, player1_epsilon)
      winner <- check_winnerCpp(board)

      if (winner == 0 & moves < 9) {
        moves <- moves + 1
        plot_board(board)
        human_move <- player_move(board)
        if (human_move == 0) {
          break
        } else {
          board[human_move] <- human_piece
        }
        winner <- check_winnerCpp(board)
      }
    }
  }
  #little ending flavour
  if (winner == human_piece) {
    print("you win!!")
  } else if(winner == -human_piece) {
    print("oh no! you lost!")
  } else {
    print("a draw..")
  }
  plot_board(board)
}

#run like:
play_game_human(1, 1)
```