Artificial Intelligence Chess

Giacomo Garbarino

giacomogarbarino7@gmail.com

Introduction

Chess

Chess is a timeless and strategic board game that has been enjoyed and played by people around the world for centuries. In chess, two players face off on a square board divided into sixty-four squares of alternating colours, normally black and white. Each player controls sixteen pieces: one king, one queen, two rooks, two bishops, two knights and eight pawns. The objective of the game is to checkmate the king of the opponent, putting it in a position where it is threatened with capture and cannot escape capture on the next move. This is achieved by using your pieces to attack and control the board, while simultaneously defending your king. There are thousands of strategies to checkmate. In chess, the draw is possible, and it is known as a stalemate. Stalemate happens one side has no legal moves to make: the king is not in check, but no piece can be moved without putting the king in check. Now we look in detail at how each piece works:

- King: the king is the most important piece on the board. It can move one square in any direction: horizontally, vertically, or diagonally. It cannot jump over the pieces.
- Queen: the queen is the most powerful piece in chess. It can move any number of squares in any direction: horizontally, vertically, or diagonally. It cannot jump over the pieces.
- Rook: the rooks are represented by castle-like towers. They can move any number of squares or vertically. It cannot jump over the pieces.
- Bishop: the bishops are represented by pointed caps. They can move any number of squares diagonally. It cannot jump over the pieces.
- Knight: the knights are represented by horses. They have a unique movement pattern: they move in an L-shape, consisting of two squares in one direction (horizontally or vertically) and then one square in a perpendicular direction. They are the only pieces that can “jump” over other pieces.
- Pawn: the pawns are represented by small infantry. They have the most limited movement. Pawns move forward one square, but on their first move, they have the option to move forward two squares. Pawns capture diagonally, one square forward. They cannot jump over other pieces. If a pawn reaches the last square on the opposite side, it can become any other piece, king excluded.

The game begins with both players setting up their pieces on opposite sides of the board. One player plays with white pieces, the other one with black ones. The player who plays with white pieces starts first. Chess is a game of deep thinking, foresight, and planning. It requires analyzing the position, predicting the moves of the opponent, and making sound decisions. It can be played for fun or competitively at various levels, from local tournaments to international competitions.

Software

The developed software implements the chess game with a single intelligent agent. A human player faces off against an artificially intelligent agent that simulates Robert James Fischer, better known as Bobby Fischer. Bobby Fischer was an american chess grandmaster and a world chess champion. Here are his statistics: 954 professional games played, 546 (57.23%) wins, 284 (29.77%) draws and 124 (13%) losses. Bobby Fischer is considered one of the best chess players of all time.

Code Development

In this section, we are going to explore the two notebooks: “ModelGenerator.ipynb” and “Code.ipynb”. The first notebook implements the dataset processing and the training of a machine learning model to implement the agent, the other one implements the game logic.

Dataset Processing and Agent Training

The first phase of the development was to find a dataset to train a machine learning model to implement the agent simulating Bobby Fischer. To do that, we downloaded from github his games from 1955 to 1972 in which he won. For each game, there is a csv file containing the position of each piece on the chess board at each played turn. Then, we created a file called “ModelGenerator.ipynb” for our aim. The needed libraries and methods are glob, os, tensorflow, pandas, numpy and the shuffle method of sklearn library. After the libraries have been imported, we defined two functions:

- **makeInputFunction[1]:** the function takes as parameters the input data, the corresponding target labels, the number of epochs, a boolean flag called “shuffle” to shuffle the data and the batch size for training. It defines an inner function called “inputFunction”. In particular, the inner function creates a TensorFlow dataset “ds” through the method “from_tensor_slices()”. This method takes the data, as a dictionary, and the labels as inputs and combines them to build the dataset. Once built, the dataset is shuffled randomly using the method “shuffle(buffer_size)”. The buffer size determines the number of elements from the dataset that are randomly selected for each shuffle operation. Shuffling the data introduce randomness which is beneficial for learning. The dataset is then batched through the method “batch(batch_size)”. Batching the dataset is useful to process multiple data points simultaneously during the training. The dataset is then repeated using the function “repeat(num_epochs)” for the specified number of epochs. This means that the entire dataset will be processed “num_epochs” times during the training. The function “inputFunction” returns the processed dataset “ds” and the outer function “makeInputFunction” returns the inner function “inputFunction” as its result.

- **splitIntoBatches:** the function splits a DataFrame into batches of a given size. It takes as inputs a DataFrame “df” and “batch_size”, the size of each batch set to 100.000. This function determines the number of rows in the DataFrame with the instruction “numberOfRows = len(df.index)”. Then it uses the instruction “intervals = np.arange(0, numberOfRows+1, batch_size)” to generate an array called “intervals” of numbers starting from 0 up to “numberOfRows”, incrementing by “batch_size”. These numbers represent the starting and ending indices of each batch. The instruction “intervals[-1] = numberOfRows” is used to modify the last element of “intervals” to ensure that the last batch includes any remaining rows. Two empty arrays “batchesX” and “batchesY” are then created to store respectively the input features and their corresponding target values. The function loops over “intervals” to create the batches, excluding the last element because it has been already modified. Inside the loop, the first instruction stores in the variable “batchX” a copy of the input features by selecting the rows from “intervals[i]” to “intervals[i+1]” using the method “df.iloc()” from the columns specified by “features”. The function “copy()” is to create a copy of the matched rows to avoid any potential linkage with the original DataFrame. The second instruction has almost the same functionality: it stores in the variable “batchY” a copy of the target values by selecting the rows from “intervals[i]” to “intervals[i+1]” using the method “df.iloc()” from the column “good_move”. The third and the fourth instruction respectively append “batchX” and “batchY” to “batchesX” and “batchesY”. Once the iteration is finished, the function returns “batchesX” and “batchesY” which contains respectively the batches of the input features and the target values.

Once we defined these two functions, we prepared the data for the training of the model. The first step consisted in storing the path of the folder containing the dataset, “Data/CSV_FISCHER”, into a variable called “path”. Since the dataset is composed of many csv files, the instruction “files = glob.glob(path + “/*.csv”)” has been used to search all the csv files in the path specified by the variable “path” and save them into a list called “files”. We then created an empty list called “array” and through a loop over the “files” list, we created a DataFrame for each csv file. Each DataFrame has been appended to the list “array”. After that, each DataFrame into “array” has been concatenated to create a single DataFrame called “train”. The data inside the variable “train” are then shuffled to introduce randomness which is beneficial for the learning. The instruction “features = list(train.iloc[:,192].columns)” stores into the list “features” the name of the first 192 columns of the “train” DataFrame. The “train” DataFrame is then splitted into input features “x” and the corresponding target values “y” using “train[features]” and “train[“good_move”] respectively. We then defined two list called “categoricalColumns” and “numericalColumns”. The first list contains the names of the first 64 (0-63) columns of the “x” DataFrame, the second one the names of the columns from 64 to 191 of the same DataFrame. After an empty list called “featureColumns” has been created to store the feature columns for the machine learning model, we defined two loops. The first loop iterates over each feature name in "categoricalColumns". At each iteration, the unique values of the current feature column, obtained by “x[featureName].unique()”, are stored in a variable called “vocabulary” which is used to create a categorical feature column. The created categorical feature column is appended to the "featureColumns" list. The second loops iterates over each feature name in “numericalColumns”. At each iteration, a numerical feature column is created. The created numerical feature column is appended to the "featureColumns" list

Once the data have been processed properly, we created and trained a machine learning model. As the first step, we created and saved into a variable called “estimator” a linear estimator classifier using the function “tf.estimator.LinearClassifier()” from TensorFlow. It uses the list “featureColumns” as feature columns. Then, we defined a loop to iterate over the pairs of “batchesX” and “batchesY” using the “zip()” function. Each pair represents a batch of input features and target values. At each iteration, it calls the function “makeInputFunction” which takes in input “dfX” and “dfY”. They are respectively the current batch of input features and target value. The function “makeInputFunction” uses “dfX”, and “dfY” to create an input function. The input function is saved in the variable “trainInputFunction”. This variable is then used as a parameter in the instruction “estimator.train()” to train the estimator. Once the estimator has been trained, it is exported and saved in the folder “Model”.

Game Logic

The second phase of the development was to implement the game and its logic[5]. The file to implement them is “Code.ipynb”. The needed libraries are chess, numpy, tensorflow, pandas, numpy, and the needed imported methods are clear_output, OrderedDict and itemgetter. After the needed libraries and methods have been imported, we started to define a series of functions:

- **predict[2]:** the function takes a DataFrame as input, converts its rows into serialized TensorFlow objects, and feeds them to a TensorFlow model for prediction. It then collects the predictions and returns them as a list. This function takes two arguments: “df”, a DataFrame, and “model”, the model used for predictions. The variable “predictions” is initialized as an empty list to store the predicted results. The function starts iterating over each row of the DataFrame “df” using “df.iterrows()”. The underscore “_” is used to ignore the index of each row since it's not used in the loop. For each row, a new “tf.train.Example” object named “example” is created. This object will hold the features for a single example. Within the nested loop, the function iterates over each column name (“colName”) and its corresponding data type (“dtype”) using “zip(df.columns, df.dtypes)”. For each column, the function retrieves the value of that column for the current row using “value = row[colName]”. Based on the data type of the column, the function performs different actions. If the data type is an object representing string data, the value is encoded to UTF-8 using “value.encode('utf-8')” and appended to the “byte_list” feature of the “example” object. If the data type is a float, it appends the value to the “float_list” feature of the “example” object. If the data type is an integer, it appends the value to the “int64_list” feature of the “example” object. After processing all the columns of a row, the function calls the “predict” signature of the provided model using “model.signatures['predict']”. It passes the serialized “example” object as input to the “examples” parameter. The predicted results are then appended to the “predictions” list which is then returned.

- **createMovesDataFrame:** the function takes a chess board “board” as input and returns a DataFrame related to the legal moves on the board. The instruction "squareFeatures = [str(board.piece_at(i)) for i in chess.SQUARES]" creates a list called "squareFeatures" that contains the string representation of the chess pieces at each square on the board. It iterates over “chess.SQUARES”, which is a list of all the squares on a chessboard. An empty list called “data” is then initialized. It will store the rows of the DataFrame. Then, the instruction "moves = list(board.legal_moves)" retrieves a list of all the legal moves on the board using the "legal_moves" method provided by the "board" object of the "chess" library. After that, the function defines a loop on the “moves” list. At each iteration, it performs the following steps: it creates two arrays of zeros each one with a length of 64 (8x8) to represent the "from" square of the move and the “to” square of the move, sets the element at the index corresponding to the "from" square of the move to 1 in the “fromSquare” array, sets the element at the index corresponding to the "to" square of the move to 1 in the “toSquare” array, defines a “row” variable to store a row of data. The row is defined as the concatenation of the “squareFeatures” list, the converted “fromSquare” array to a list, and the converted “toSquare” array to a list into a single row list. The variable “row” is then appended to the list “data”. Once the rows of defined, the function then defines the columns. With the instruction "squareNames = chess.SQUARE_NAMES", it assigns the list of square names to the variable "squareNames". Then, it defines two instructions "moveFromFeatures = ['from_' + square for square in squareNames]" and "moveToFeatures = ['to_' + square for square in squareNames]" to create two lists to respectively contain strings representing the "from" square features and the “to” square features. It prefixes each square name in "squareNames" with the string "from_" and “to” respectively. The function prepares the columns for the DataFrame by defining a variable called “columns” as the concatenation of “squareNames”, “moveFromFeatures”, and “moveToFeatures” lists. With the instruction "df = pd.DataFrame(data=data, columns=columns)", the function then creates a pandas DataFrame called "df" using the "data" list as the data and "columns" as the column names. The instruction "df[moveFromFeatures + moveToFeatures] = df[moveFromFeatures + moveToFeatures].astype(float)" converts the columns corresponding to the "from" and "to" square features in df to float type using the "astype" method. In the end, the function returns the DataFrame “df”.

- **getPieceValue[5]:** the function takes a piece “piece” and a square “square” as input to calculate the value of a piece on a specific square based on the piece type and position. Each piece is identified with one letter: “P” for pawn, “N” for knight”, “B” for bishop”, “R” for rook, “Q” for queen and “K” for king. If these letters are uppercase, they identify the white pieces, otherwise if they are lowercase, they identify the black pieces. The function first calculates the "x" and "y" coordinates of the square using the "divmod" function. The "square" parameter represents the position of the piece on the chessboard as a single integer ranging from 0 to 63, where 0 represents the top-left square and 63 represents the bottom-right square. The "x" coordinate represents the row number, and the "y" coordinate represents the column number on the chessboard. Next, the function checks if the agent is playing as black (“AI_black” is true) or white (“AI_black” is false). Based on this, it sets the values for white and black variables. If the agent is black, white is set to -1 and black is set to 1. If the agent is white, white is set to 1 and black is set to -1. The function then evaluates the value of the piece “piece” based on its type and position. It uses a series of if-elif-else statements to determine the piece type, calculate its value accordingly and return this value. If the piece is “None”, an empty square, the function returns 0. If the square is not empty, according to the piece and its colour, the value is calculated as “colour * (constant + pieceColorEvaluations[x][y])”. For example, if the piece is a white pawn ("P”), the value is calculated as "white * (10 + pawnWhiteEvaluations[x][y])", if the piece is a black queen (“q”), the value is calculated as “black * (90 + queenBlackEvaluations[x][y])”. For each type of piece, a matrix containing values based on its position is defined. “queenBlackEvaluations[][]” and “pawnWhiteEvaluations[][]” are matrices containing the values of the black queen and a white pawn respectively.
- **minimaxAlphaBetaPruning[3]:** the function implements the minimax algorithm with alpha-beta pruning to find the best move the agent can make. It takes three parameters: depth (the depth of the search tree), board (the current state of the chessboard), and “model” (the trained model for move predictions). Inside the function, there is a nested helper function called "minimaxAux" that performs the minimax search with alpha-beta pruning. It takes additional parameters: "alpha" and "beta" (the bounds for pruning) and "guard" (a boolean flag indicating whether it's the minimizing turn or the maximizing turn). If the "depth" parameter reaches 0, the function evaluates the current state of the board and returns the negated evaluation score. The evaluation is done by summing up the values of the pieces on the board using the "getPieceValue" function. If the depth is greater than 3, the function generates a list of legal moves "movesList" and creates a DataFrame "df" representing the board state. It then uses the "model" parameter to make predictions on the moves and retrieves the probabilities of good moves. The "movesAndProbs" list is created by pairing each move with its corresponding probability, and it is sorted based on these

probabilities in descending order. The function selects the top 50% of moves from the sorted list as "legalMoves". If the depth is 3 or lower, the function directly assigns "legalMoves" to the list of all legal moves on the board. The "bestMove" variable is initialized to a large negative value (-9999) if it's the maximizing turn ("guard" is True), and a large positive value (9999) if it's the minimizing turn ("guard" is False). The function iterates over each move in "legalMoves", applies the move to the board, and then recursively calls "minimaxAux" with a decreased "depth", updated "alpha" and "beta" values, and negated "guard" value. After the recursive call, the move is undone by calling "board.pop()". If it's the maximizing turn ("guard" is True), it updates the "bestMove" and "alpha" values if the current move yields a higher value. It also performs alpha-beta pruning by checking if the beta cutoff condition is met ("beta <= alpha"), and if so, it breaks out of the loop since further exploration is unnecessary. If it's the minimizing turn ("guard" is False), it updates the "bestMove" and "beta" values if the current move yields a lower value. It also performs alpha-beta pruning by checking if the beta cutoff condition is met, and if so, it breaks out of the loop. After iterating over all moves, the function returns the "bestMove" storing the best move. The main part of the "minimaxAlphaBetaPruning" function initializes the “legalMoves” variable repeating the same steps done in the "minimaxAux" function in case when "depth" is greater than 3, sets the variables “bestMove”, “bestMoveCalculated” and “guard” to -9999, None and True respectively. It then iterates over "legalMoves" again similarly to the "minimaxAux" function. It applies each move to the board, calls "minimaxAux" with the updated parameters, and then undoes the move. If a better move has been found ("value >= bestMove"), it updates the "bestMove" and assigns the current move to the variable "bestMoveCalculated". Finally, the function returns the "bestMoveCalculated", which represents the best move found during the search.

- **checkmate[4]:** the function takes a move “move” and a “board” chess board object as input, applies the move to a separate copy of the chess board and determines if the resulting positioning is a checkmate. The first instruction "fen = board.fen()" retrieves the FEN (Forsyth–Edwards Notation) representation of the current chess board positionings. FEN is a standard notation for describing chess positionings. The second instruction "boardCheckmate = chess.Board(fen)" creates a new chess board object called "boardCheckmate" using the FEN representation obtained in the previous step to simulate the move without modifying the original board. Then, the function uses the instruction "boardCheckmate.push(move)" to simulate the move on the separate chess board object "boardCheckmate”. In the end, with the instruction "return boardCheckmate.is_checkmate()", the function checks if the move results in a checkmate position on the chess board "boardCheckmate" and returns true or false.

- **startGame[4]:** the function provides the main game loop using recursion, alternating between the human player and the agent until the game reaches a stalemate or a checkmate condition. It takes two parameters: a boolean called “guard” to indicate which player’s turn it is and a chess board “board”. The function first checks if the game has reached a stalemate using the instruction "board.is_stalemate()". If it is a stalemate, the function clears the output with the method “clear_output()”, displays the current game board with the “display(board)” instruction, prints "Stalemate!", returns and the game finishes. The instruction “clear_output()” followed by “display(board)” is a sort of upload operation of the chess board. If it's not a stalemate, the function proceeds based on the value of "guard". If “guard” is True, it means it's the human player's turn. If the game is in checkmate, the function uploads the board, prints “AI wins”, returns and the game finishes making the agent win. If the game is not in checkmate, the function uploads the board, prompts the user for their move using the "input()" function, and converts the input move into a chess move object using "chess.Move.from_uci()". If the input move is not a legal move in the current board position, the function recursively calls itself with the same “guard” and “board” parameters until a valid move is entered. Once a valid move is entered, it pushes the move onto the game board using "board.push(move)" and then recursively calls itself with the guard parameter flipped to switch to the agent's turn. If it is the agent’s turn ("guard" is False), it first checks if the game is not in checkmate. If the game is in checkmate, the function uploads the board, prints “User wins”, returns and the game finishes making the user win. If it's not in checkmate, the function first uploads the board. Next, it iterates over all legal moves in the current board position and checks if any of those moves result in an immediate checkmate for the agent using the "checkmate()" function. If such a move is found, it pushes that move onto the board and returns, uploads the boards, prints “AI wins” and returns making the agent win. If no immediate checkmate move is found, it calculates the number of legal moves remaining in the current position and assigns it to a variable called "numberOfMoves". Based on the value of "numberOfMoves", it selects a search depth for the minimax algorithm with alpha-beta pruning ("minimaxAlphaBetaPruning()") to determine the agent's move. The search depth increases as the number of moves decreases, suggesting that the agent puts more effort into finding a good move when fewer moves are available. It calls "minimaxAlphaBetaPruning()" with the appropriate search depth, the current game board, and a model object (not defined in the provided code) to calculate the agent's move. The chosen move is then pushed onto the game board. Finally, it recursively calls itself with the guard parameter flipped (from False to True) to switch back to the human player's turn.

After all the needed functions have been implemented, for each type of piece, we defined a matrix containing the values based on their position on the board. The values inside each matrix must be meaningful. In the end, we stored the trained model in a global variable called “model”, defined a global variable called “AI_black”, and implemented the functionality to let the user choose to start first or not use the “input” function. Once the user has chosen to start first (“AI_black=True”) or not (“AI_black=False”), the game starts.

Usage Instructions

To play a chess game againts Bobby Fischer, open with jupyter lab the file “Code.ipynb” and run the whole notebook. Once done this, under the last cell a chessboard will appear, and the user simply must follow the instructions provided by the notebook itself. The agent can take several minutes to perform a move.

References

[1] Tensorflow, Build a linear model with Estimators. https://www.tensorflow.org/tutorials/estimator/linear

[2] iAmEthanMai, 2021. https://github.com/iAmEthanMai/chess-engine-model/blob/main/python_chess_engine.ipynb

[3] Mina Krivokuca, 2021, Minimax with Alpha-Beta Pruning in Python. https://stackabuse.com/minimax-and-alpha-beta-pruning-in-python/

[4] python-chess, a chess library for Python. https://python-chess.readthedocs.io/en/latest/

[5] freecodecamp, A step-by-step guide to building a simple chess AI, 2017, https://www.freecodecamp.org/news/simple-chess-ai-step-by-step-1d55a9266977/