**Hexapawn**

Hexapawn is a simple turn based game played on a 3×3 board. Each player begins with 3 pawns - WHITE (MAX) in the bottom row and BLACK (MIN) in the top row. Pawns can move as normal in chess (i.e. white pawns can move up one square or can capture a black pawn diagonally up one square, and black pawns can move down one square or can capture a white pawn diagonally down one square). The goal of each player is to either get one of their pawns to the other end of the board, or to make it such that their opponent is stuck on their next move. Figure 1 shows the initial state of the game.

**Representing the Problem**

A board in Hexapawn can be pretty straightforwardly represented as a 2D array. For example, the initial state in Figure 1 can be represented by the array in Figure 2. Actions could be lists: (advance row column), (capture-left row column), (capture-right row column) - where row and column specify the location of the pawn to be moved.

A picture containing square

Description automatically generated

There are several schemes we could use to represent a state in Hexapawn, but the one required will use a vector of 10 values - the first one will be 0 or 1 and will specify whose turn it is, and the other 9 specify the state of the board in row major order. Each cell will either be a 0 (an empty square), 1 (a white pawn), or −1 (a black pawn). Thus, the initial state from Figures 1would be as: (0 -1 -1 -1 0 0 0 1 1 1)

To represent the output - that is, which moves are optimal in the corresponding input state - a vector of nine values is required. These values again represent the squares on the board in row major order, and a 0 specifies that moving a pawn to the corresponding square would not be optimal (or would be illegal), while a 1 specifies that moving a pawn to that square would be an optimal move. Whether the pawn was moved as a result of an advance or a capture is dependent upon the state of the input board.

**PROJECT REQUIRED FUNCTIONS**

Part 1: Implement a Formalization of Hexapawn as well as functions TOMOVE(s), ACTIONS(s), RESULT(s, a), IS-TERMINAL(s).

Part 2: Implement a minimax search that builds up a policy table for a game that has been formalized as you have for Hexapawn in Part 1. For each state, the policy should include the value of the game as well as every action that achieves that value.

Part 3: Design a graph data structure that lets you link nodes of neurons (units) in arbitrary ways as a directed graph. Keep in mind that an edge going from one unit i to another j means that i’s output will be given as input to j, so both i and j will need access to a data structure which stores such input/output. You’ll need to also consider how you’ll keep track of neurons in order to implement the classification and back-propagation algorithms in Parts 4 and 5, respectively.

Part 4: Implement a classify function that takes an instance of the network you designed in Part 3 and a vector of inputs, and then ”runs” the network on that vector of inputs. If you keep an ordered list of neurons (or ordered list of layers), then you should be able to walk the list in order, reading the inputs of each neuron and passing them to its activation function, and then writing the result to the neuron’s outputs.You should implement two activation functions to use with your network:

* 1. The sigmoid (or logistic or logit) function, described in Equation (1) (pictured below)

A picture containing text, clock

Description automatically generated

* 2. The ReLU (rectified linear unit) function, described below:

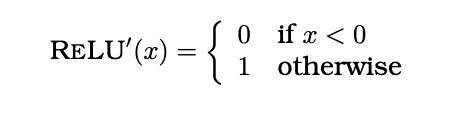
RELU(x) = MAX(0, x)

Part 5: Implement an update-weights function that takes an instance of the network you designed in Part 3 and a vector of expected outputs, and then uses back propagation to modify the weights in your network based on the differences between the expected outputs and the set of outputs obtained from the last call to classify. Keep in mind for this that the derivative σ 0 of the sigmoid function is defined below:

Text

Description automatically generated

and the derivative RELU derivative of RELU is defined below:



At this point, you should be able to build and train arbitrary artificial neural networks, so you may want to consider checking the correctness of your code by constructing the network shown in the figure below, initializing all the weights to random numbers between −1 and 1, and then training it on the data in the table below. After a few rounds of feeding it the examples in a random order, your network should be able to successfully classify all four of them.

Figure:

Diagram

Description automatically generated

Table:

A picture containing text, clock, clipart

Description automatically generated