CISC 481 Programming Assignment 3

April 25, 2023

Assignment Objectives

In this assignment, you'll be implementing your own deep[1] feed forward artificial neural network. While completing this assignment, you'll learn:

- How to construct a directed graph representing the network
- How initial weight configuration affects the network's ability to learn
- A little about tuning a network's parameters - the learning rate, which activation function your neurons use, the number of layers in the network as well as the number of nodes in each layer.
- How to implement back propagation using the L_2 (loss squared) function to do gradient descent based learning.

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

[My examples here are in Lisp, but feel free to use something similar that's easy to parse in your target language.]

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.

However, we still need some way of representing these that a neural network will understand. Recall that the inputs and outputs of a neural network are vectors of real numbers. There are several schemes we could use to represent a state in Hexapawn, but the one we'll use is 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

[1]It's a little unclear how many layers are required for an artificial neural network to qualify as a deep neural network. You'll certainly be able to experiment with many layers in this project, though!

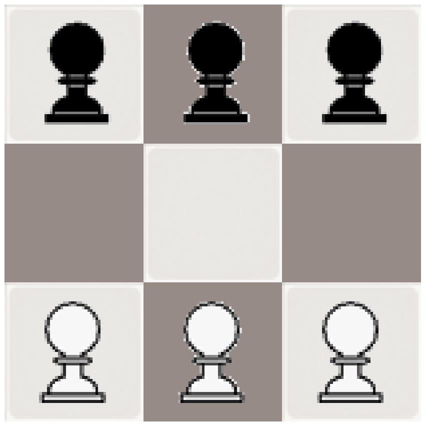

Figure 1: The initial state of Hexapawn.

```
#2A((:b  :b  :b)
    (nil nil nil)
    (:w  :w  :w))
```

Figure 2: A Lisp representation of the initial state of Hexapawn.

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 1 and 2 would be as shown in Figure 3.

```
#(0 -1 -1 -1 0 0 0 1 1 1)
```

Figure 3: The initial state of Hexapawn, encoded as a vector suitable for input to an artificial neural network.

To represent the output - that is, which moves are optimal in the corresponding input state - we'll use a vector of nine values. 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.[2] See Figure 4 for an example of how this works.

Part 1 [15 pts]

Implement a Formalization of Hexapawn based upon the definition given in Section $5.1.1$ of the book. That is a state description as well as functions $\textsc{ToMove}(s)$, $\textsc{Actions}(s)$, $\textsc{Result}(s, a)$, $\textsc{Is-Terminal}(s)$, and $\textsc{Utility}(s)$[3]

[2]The astute reader will note that this allows for some ambiguity - if it's possible for a pawn in the middle column to be captured by one to either side of it, this representation won't tell us which pawn should be the one to capture. That's OK, we're not going for perfect, here.

[3]Of course, UTILITY is defined in the formalization to have 2 parameters - one for the state, and one for the player whose utility you're interested in. Given that we're dealing with a zero-sum game, though, we can just define UTILITY to

```
#(0 0 0 1 1 1 0 0 0)
```

Figure 4: From the initial state, this encoded policy specifies that any of `(advance 2 0)`, `(advance 2 1)`, or `(advance 2 2)` will get MAX their optimal expected value.

Part 2 [10 pts]

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 [20 pts]

Design a graph data structure that lets you link nodes of neurons (units) in arbitrary ways as a directed graph.

There are two ways I know of to do this:

- Build your graph as an *adjacency list*. 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.

  One very simple way to do this in Lisp is to represent a neuron as a `cons`, where the `car` is a list of incoming edges and the `cdr` is a list of outgoing edges. An edge can be a property list, structure, or the like which keeps a weight, value (for the input/output), and a perceived error (for back-propagation). A neuron i feeds into neuron j when the same edge object appears in both i's `cdr` and j's `car`.

- Build your graph as a series of *adjacency matrices*, such that each layer i of the network has a $N \times M$ matrix of weights, where N is the number of neurons on layer i and M is the number of neurons on layer $i - 1$ (or the number of inputs, if i is the first layer). **NB** that this scheme requires you to store your dummy weights (*biases*) in a separate vector.

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 [20 pts]

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.

If you use adjacency matrices, at each layer you'll take the weight matrix and multiply it by the outputs from the previous layer (or inputs from the outside world, if it's the first layer) and then add in the biases to get the values to input into your activation function.

You should implement two activation functions to use with your network:

give the utility of MAX and know that MIN's utility is just the negation of that.

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

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{1}$$

2. The *ReLU* (*rectified linear unit*) function, described in Equation (2)

$$\textsc{ReLU}(x) = \textsc{max}(0, x) \tag{2}$$

Part 5 [20 pts]

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`.

Recall that for neuron j with link from nueron i, we update the weight between i and j by taking the output o_i from node i and multiplying by j's "error delta" Δ_j and then subtracting the result from the existing weight. These error deltas are defined in a recursive fashion:

- for some output neuron i, we can take the value $\hat{y}$ that was output by i and the value y that we *expected* to get from i given our input to the network and get $\Delta_i = 2(\hat{y} - y)g_i'(in_i)$, where g_i is the activation function for i (and thus g_i' its deriative), and in_i is the weighted sum of outputs from the previous layer that was passed as input to i to obtain $\hat{y}$.
- for neuron i in hidden layer L_n, we get:

$$\Delta_i = \sum_{j \in L_{n+1}} \Delta_j w_{i,j} g_i'(in_i) \tag{3}$$

  That is, Δ_i is the weighted sum of all the errors of the nodes that i feeds into.

Keep in mind for this that the derivative σ' of the sigmoid function is defined in Equation (4)

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)) \tag{4}$$

and the derivative $\textsc{ReLU}'$ of $\textsc{ReLU}$ is defined in Equation (5)

$$\textsc{ReLU}'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{otherwise} \end{cases} \tag{5}$$

Regardless of how you implemented your network, you should be able to go through the layers in *reverse* and employ a similar operation to what you did in classify.[4]

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 Figure 5, initializing all the weights to random numbers between -1 and 1, and then training it on the data in Table 1. 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.

[4] *e.g.* for the matrix implementation, you can get layer n's errors by multiplying layer $n+1$'s errors by $n+1$'s weight matrix. Be mindful of the ordering here - when classifying, you multiplied the *matrix* by the *inputs* (thus summing over the *rows* of the matrix). When obtaining the errors, you'll want to put the next layer's errors *first* in the multiplication so that you're summing over the *columns* of the matrix.

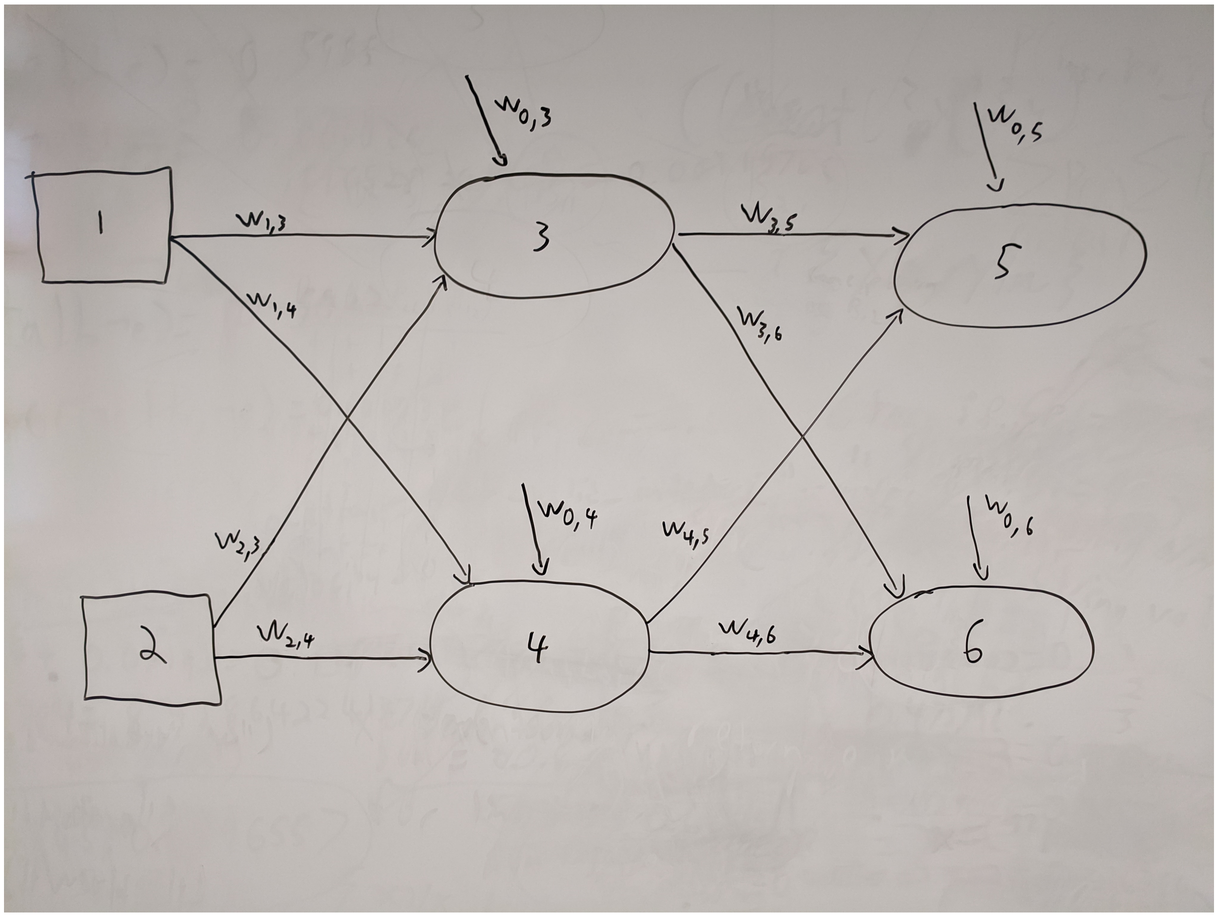

Figure 5: Simple 4 neuron, 2 layer feed-forward neural network

Part 6 [15 pts]

Now you get to design a network that learns (at least reasonably well) to play Hexapawn. Your network will have ten input nodes (one for specifying whose turn it is, and nine for the state of the board), and nine output nodes (for specifying which cells moving a pawn to would be optimal). The network should also be *fully connected* - that is, every input node should have an outgoing edge to every node in the first hidden layer, every node in a hidden layer should have an outgoing edge to every node in the next hidden layer, and every node in the last hidden layer should have an outgoing edge to every node in the output layer. Otherwise, the structure is entirely up to you. You can try varying numbers of layers, with varying numbers of nodes in each layer. You can try tuning α, and playing around with how you assign initial weights. You can play around with the order in which you show examples to the network[5]

Describe the architecture that you find learns to play the best game. For how many states does it confidently suggest at least one optimal move? For how many does it confidently suggest a bad (or illegal) move?

[5]For this, though, I would suggest sticking with doing several rounds of feeding a random permutation of the entire policy table.

x_1	x_2	y_1 (carry)	y_2 (sum)
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Table 1: Input/output sets for a two-bit adder.

Submitting

You should submit all of your code - *document it appropriately, as you'll be graded on style.* If you use any third party libraries, you should also submit instructions for setting up the environment in which to run your code.[6] You should also submit a short writeup of your findings from Part 6. If you'd like, you can also include a copy of the policy table that your best network suggests.

[6] That being said, no Pytorch, Tensorflow, etc - you should be implementing your neural network yourself!