THE PLAN

1 Coding Connect-Four

Not writing anything here because it seems pretty easy.

2 The ComputerPlayer Class

- Function *init* - Instantiate a NeuralNet object (see below)
- Function *takeTurn* - Input the current gameBoard (which should be an element of the Game class). Find the list of possible moves. For each possible move, take the corresponding game board and put it through the NeuralNet. Note that the NeuralNet somehow needs to know who it is optimizing for (should it find the move that makes player1 win or player2?). My proposal is to save one board for each player, where a 1 in some locations means "my piece", -1 means "their piece", and 0 means empty. The reason I suggest this is so that we can handle the issue before it ever reaches the NeuralNet (so we don't need to add currentPlayer as an input to the NeuralNet).
- var *gameBoard* - Read the above discussion for why I want this player to have his own gameboard. If we wind up agreeing on this then we should move around how variables are stored (the Game class shouldn't store the board I don't think).

3 The Neural Net Part

Note: A lot of this is subject to change, especially since I'm writing this up in a plane with no internet access so it could wind up being bogus. But here's my first pass at putting together the neural net structure.

The overall plan is for each neuron to accept a linear combination of values from the previous layer, pass it through our activation function (my first vote is ReLU it's so easy), and send this value with the corresponding weights onto the next layer. I propose we implement this with the following structure

3.1 NeuralNet Class

- Function *init* - The constructor. From a separate file it loads the neuron structure, along with each of the connections and weights.
- Var *numLayers* - Number of layers, including input and output layer
- Var *neurons* - a 2d list. First index is which layer, second index is which neuron in layer. Actual list elements are from the Neuron class, which is below.
- Function *doTheThing* - The function that calculates the value based on a given input. This should load up the first layer (index 0) of the network with the input values, *fire* each layer (one at a time), and collect the output from the last layer.

3.2 Neuron Class

- Function *init* - Inputs the *nextNeurons* and *weights*. Sets *value* to zero.
- Var *value* - Holds the input from the previous layer. Initialize this to zero, then update it as the nodes from the previous layer get fired off. Once the previous layer is finished this is the final value.
- Var *nextNeurons* - A list of pointers to the next layer of neurons
- Var *weights* - A list of weights corresponding to *nextNeurons* above
- Function *fire* - Apply the activation to *value* and send it (with appropriate weights) to each guy in *nextNeurons*, updating the *value* in that dude.

4 Update Process

Hahahhahaha I don't know how this works. This is where most of the studying will come in. But we'll somehow update the file that stores all of our network weights.

There's even a decision to be made between updating after every turn of the game, or only updating after a full game has been played out.

5 Backpropagation

Backpropagation is a method for updating the weights of the neural net when desired output values are known for given input values.

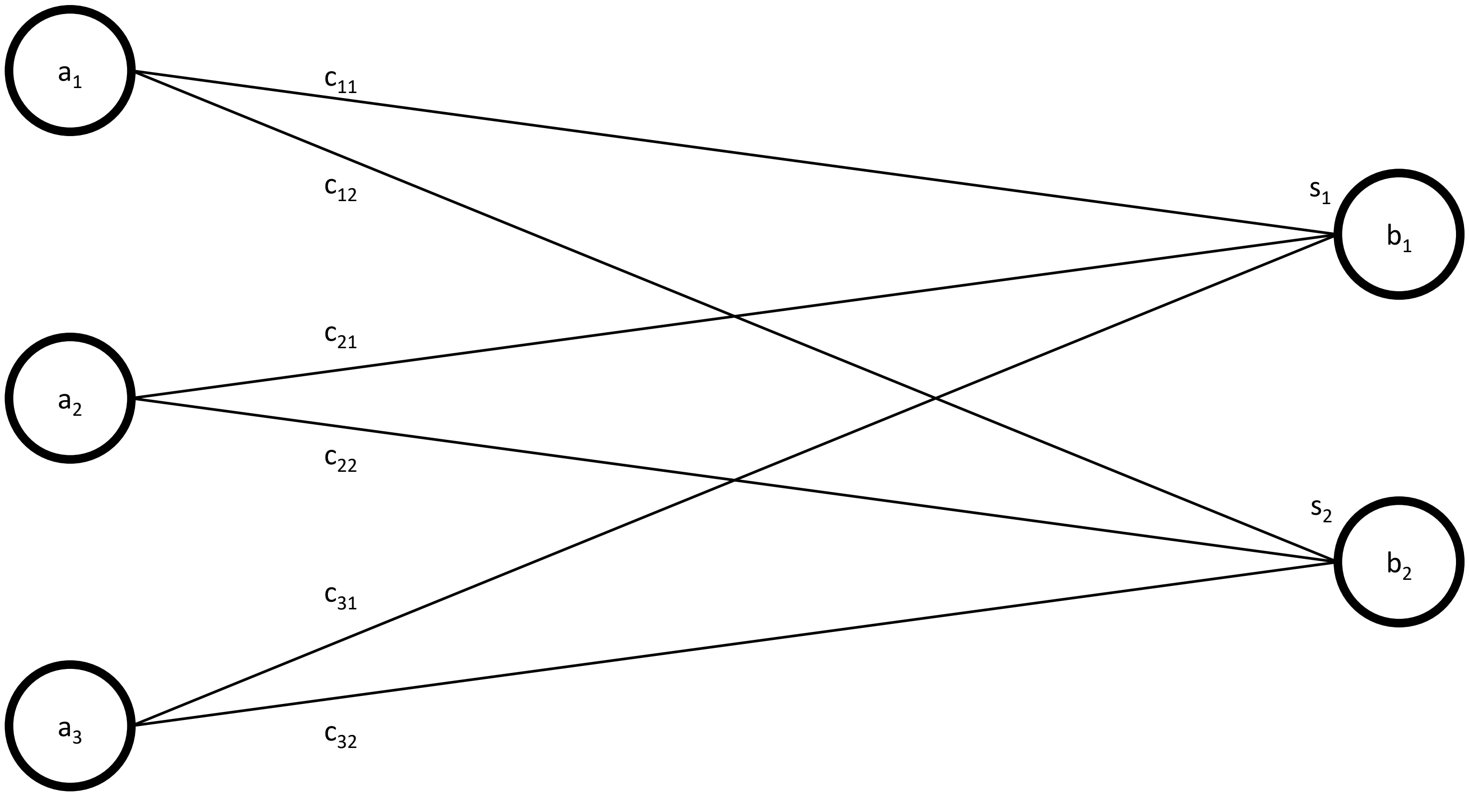

Figure 1: Neural net example with variable definitions used in backpropagation section.

5.1 Output Difference

Forward propagate the given input values through the neural net to produce resulting output. For each output neuron, compare the resulting value from the neural net to the given target value. We can then calculate how far off the mark we were for each (ith) neuron:

$$\delta b_i = b_i^{actual} - b_i^{target} \tag{1}$$

5.2 Sum Difference

From the output difference δb_i, we want to know how far off the corresponding sum (value of each neuron before activation function is applied) is δs_i. We can take advantage of the derivative of the activation function ($A(s_i) = b_i$) here:

$$A'(s_i) \equiv \frac{dA(s_i)}{ds_i} = \frac{db_i}{ds_i} \approx \frac{\delta b_i}{\delta s_i} \tag{2}$$

This final step of 2 can be obtained via Taylor expansion of the activation function and approximates the change in b_i as linear in s_i (just the tangent line at s_i). From here, as long as we are able to compute the derivative of the activation function ($A'(s_i)$), we can then approximate the variation of the sum necessary to correct our output:

$$\delta s_i = \frac{\delta b_i}{A'(s_i)} \tag{3}$$

5.3 Weight Difference

We're now in a position to approximate the variation in weights necessary to produce the correct sum difference (which in turn will approximately produce the correct output). We first recall the definition of the sums s_i:

$$s_i(a_j, c_{ji}) = \sum_{j=0}^{N} a_j c_{ji} \tag{4}$$

Here, N is the number of neurons (in the previous layer) connected to the ith output neuron. Using reasoning similar to that of the previous section, we can expand the variation of s_i:

$$\delta s_i(a_j, c_{ji}) = \sum_{j=0}^{N} \frac{\partial s_i}{\partial a_j}\delta a_j + \frac{\partial s_i}{\partial c_{ji}}\delta c_{ji} \tag{5}$$

However, at this stage in the process, the values of the neurons in the previous layer (a_j) are fixed, so their variations will vanish ($a_j = 0 \quad \forall j$). In addition, the derivative of the sum with respect to an individual weight is, by equation 4, simply equal to a_j. What's left is then:

$$\delta s_i(a_j, c_{ji}) = \sum_{j=0}^{N} a_j \delta c_{ji} \tag{6}$$

This equation simply states that the change in the output sum must equal the sum of each incoming activation times the change in its corresponding weight. This seems to match intuition. This equation does not, however, tell us how to divide up the load between the incoming neurons. For example, we could let a single neuron (say c_{ki}) do all the lifting and leave the rest unchanged. In this case:

$$\delta c_{ji} = \begin{cases} \delta s_i / a_k & j = k \\ 0 & j \neq k \end{cases} \tag{7}$$

This equation is under-constrained and we will have to make a choice in order to proceed. A decent choice seems to be splitting the load equally among the incoming neurons. In this case, each neuron connection is updated each time the backpropagation is run, which seems to have both benefits and malefits. With this choice, we dictate that each element of the sum have the same value:

$$a_j \delta c_{ji} = a_k \delta c_{ki} \equiv \epsilon \qquad \forall j, k \tag{8}$$

Then equation 6 becomes:

$$\delta s_i(a_j, c_{ji}) = \sum_{j=0}^{N} a_j \delta c_{ji} = \epsilon \sum_{j=0}^{N} = N\epsilon = N a_k \delta c_{ki} \tag{9}$$

Solving for the change in the kth weight:

$$\delta c_{ki} = \frac{\delta s_i}{N a_k} \tag{10}$$

Now that the change in each neuron weight has been approximated, we can simply add the variation to the original weights and save the updated weights.

5.4 Propagating Back

I thought propagating this procedure back through the other hidden layers to the input layer would be trivial once I had the single layer figured out, but I'm not seeing it right away so will have to go back to Steven's stupid article. This could be a sign that I've done something wrong/don't actually have the right understanding of this procedure. Will have to look further into it later.

6 Random Moves

We also need to throw random moves into the training. If you don't have random moves, the net will tend to optimize to a local minimum. There's apparently an ϵ-greedy constant (between 0 and 1) that gives the probability of taking a random move (1 means always random). I learned about something similar called simulated annealing in a computational physics class, but basically you decrease ϵ over training, which hopefully allows you to settle into a global rather than local minimum.