Designing and Implementing a Neural Network: https://github.com/rhit-hsinr/EvoRoboPP

I have took the provided sample code and tweaked it a bit to be a bit more efficient in running. When running a neural network, it is common to have a form of mathematical expression as to how fast or how the neurons should progress. The progress that I have chosen to use is the sigmoid, which is a non-linear expression.

```
def sigmoid(x):
    return 1/(1+np.exp(-x))
```

Figure 1: sigmoid expression

I have tried to use a sigmoid derivative as well to smoothen the curves but it didn't work very well so I didn't use it later on. I have changed the code a bit but the base code is the same. I have added more methods for a bit of flexibility within the neuron to help with the evolutionary algorithm later on.

```
def set_weights(self, W, bias):
    self.W = W
    self.bias = bias

def get_weights(self):
    return self.W, self.bias
```

Figure 2: Setting and getting weights for EA

The Neural Network also consists of functions to set the weight and also get the weight as well. The hidden_weights and output_weights refer to the weights of the hidden layer, which is the middle part between the input and output layer, making the neural network a multi-layered neural network for non-linear problems such as an XOR problem.

```
def set_weights(self, hidden_weights, output_weights):
    for i in range(self.nh):
        self.H[i].set_weights(hidden_weights[i][0], hidden_weights[i][1])
    self.nO.set_weights(output_weights[0], output_weights[1])

def get_weights(self):
    hidden_weights = [unit.get_weights() for unit in self.H]
    output_weights = self.nO.get_weights()
    return hidden_weights, output_weights
```

Figure 3: Setting and getting weights for the Neural Network

Using the Neural network to solve the XOR problem

The rest is similar to the provided code and the simulation is run and can be seen below. Figure 4 is before the Neural Network has been trained and the gradient descent is just random that starts from 0.

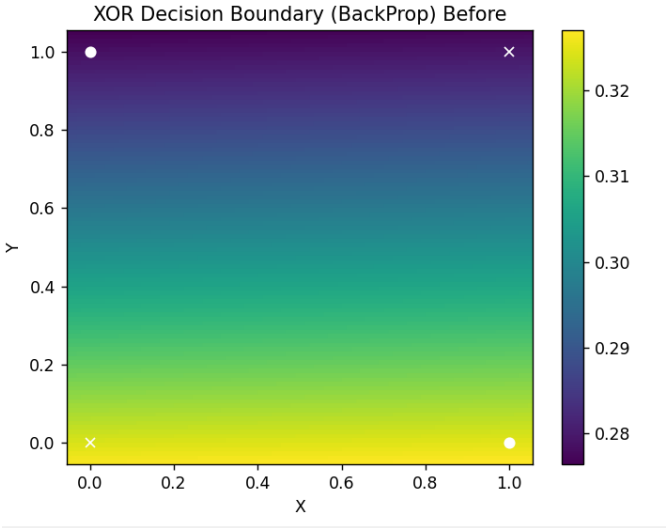

Figure 4: Before training of the Multi-Layered Neural Network

After training, it can be seen that there is a middle line of yellow that indicates that those are the answers; however, it is not that satisfactory since it also includes anything in between, not just the top left and bottom right. This caused me to question what is the reason this is taking place and started tweaking around with the code.

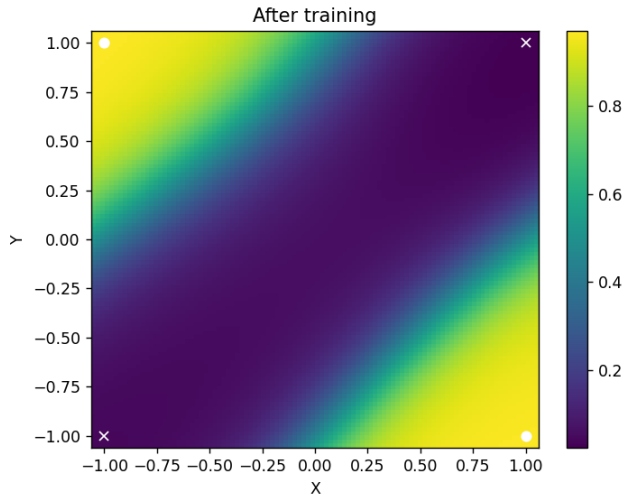

Figure 6: After training of the new Multi-Layered

After tweaking it a bit, it is the code above that helped. Being able to change the weights differently and setting them during the runs is helpful and it allowed me to achieve something like the figure above. It is more isolated and it is even closer.

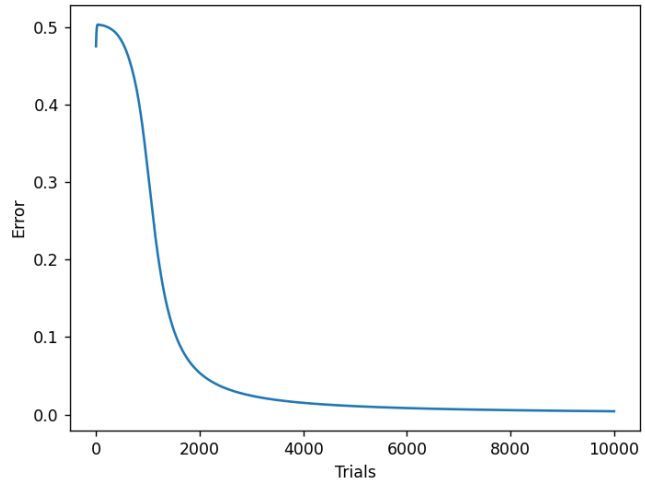

Figure 7: Fitness graph of the improvement

It can be seen from the fitness graph that it took about 2000 trials and it is very close to 0 error. However, it is not as desired as I thought I would like it.

Changing the code for better results???

After looking at the code, understanding it a bit more, I started to change it. Since it was already a multi-layered model capable of solving the XOR, it isn't really necessary to keep adding on layers in the hidden layers for no reason. I decided to swap away from the sigmoid function and try others instead. Eventually, I decided to bring back in the derivative to help me out.

```
errorOutput = (target - output) * tanh_derivative(output)
errorHidden = [errorOutput * self.nO.W[i] * tanh_derivative(hiddenOutput[i]

# Update output weights
for i in range(self.nh):
    self.nO.W[i] += 0.1 * hiddenOutput[i] * errorOutput
self.nO.bias += 0.1 * errorOutput

# Update input-hidden weights
for i in range(self.nh):
    self.H[i].W += 0.1 * errorHidden[i] * np.array(Input)
    self.H[i].bias += 0.1 * errorHidden[i]

return abs(errorOutput)
```

Figure 8: Using tanh derivative for the training

Because of that, I used tanh derivative for the errors in the training and used the tanh function for forwarding. It turned out to be great and can be seen in the following figures below.

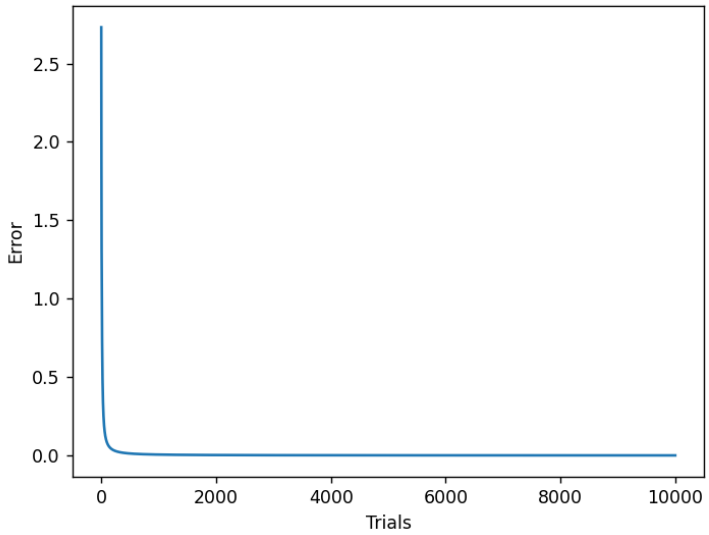

Figure 9: Fitness function of solving XOR

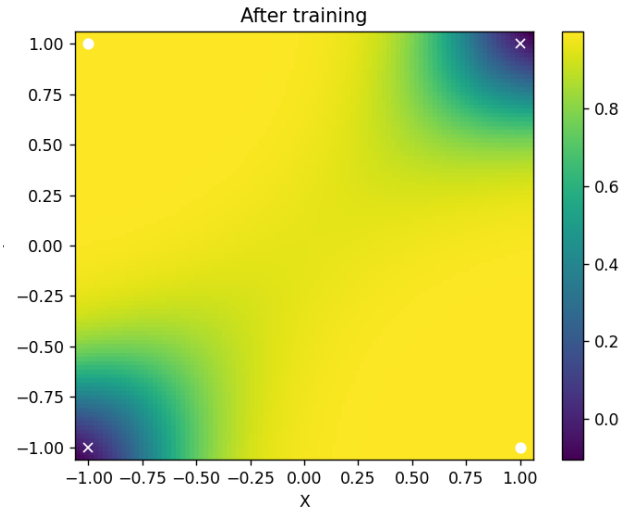

Figure 10: Gradient graph of tanh

The color gradient has changed but it can be seen that it is much closer to the target. The circles are smaller and the other answers seems to have been rejected even more, only leaving the big solutions to the XOR.

Using EA

Im actually a bit unsure about how to incorporate EA into this. It seems to be that we are supposed to ditch the training and just use EA to solve the XOR problem but then if that's the case why using Neural Network at all. What I decided to do was then feed the neural network and kind of make them compare with each other for EA. So they still do the training and the set_weights functions will allow them to be constantly changing based on the EA. I was stuck on this for quite a while and eventually I got an output, but not a solution.

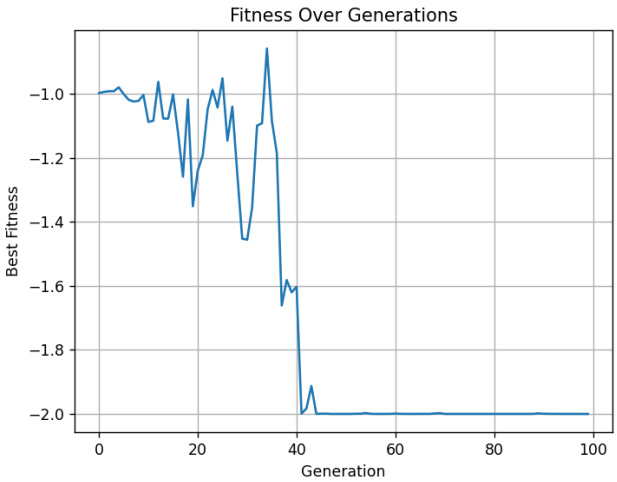

Figure 11: Fitness of using EA with Neural Network

I don’t know how it really ended and I was really curious about why it didn’t work.

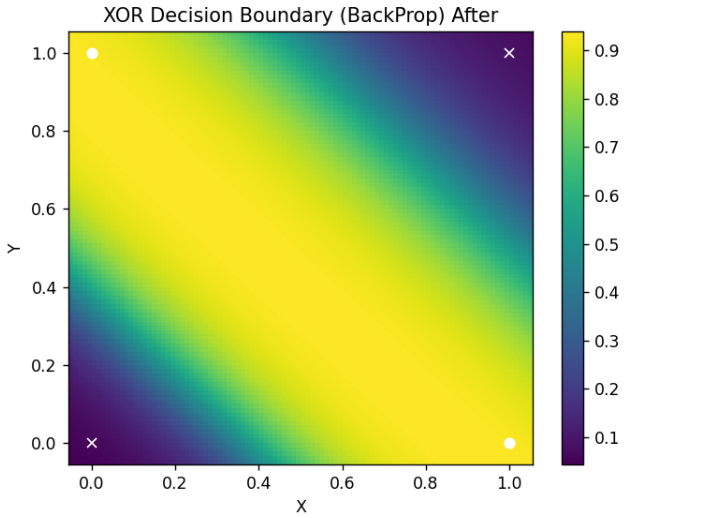

Figure 12: Gradient graph

I honestly have no clue why the gradient graph looks like this and why the fitness is just so weird. It was jumping in weird ways when using the EA even though it is supposed to be better. The fitness function that I have used is this

```
def evaluate_fitness(self, neural_net):
    # Define the XOR inputs and corresponding outputs
    xor_inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
    xor_outputs = np.array([0, 1, 1, 0])

    # Calculate the total error for this neural network
    total_error = 0
    for i in range(len(xor_inputs)):
        output = neural_net.forward(xor_inputs[i])
        total_error += abs(output - xor_outputs[i])  # Mean s
    return -total_error  # Return negative error as fitness (
```

Figure 13: Fitness evaluation function for using the EA.

The graph just gets messier and messier each time and so I was done with it and decided to move on.

Nature Inspiring

I think something cool that would benefit from a neural network, which is particularly very strong in classification or optimization, is for something like resource management or autonomous systems. I recently saw a post about how a 4.5 by 4.5 square can be optimized to fit **seventeen** 1 by 1 cubes by placing it the most optimal way even though people think the max is just 16. I find it pretty interesting that maybe a neural network that can process through a lot of different inputs to find the most optimal output can solve this. These unstructured data that has high-dimensionality can be solved by neural networks and I find it amazing.

A more obvious one that a feed-forward network can't really solve is if it's sequential or time-dependent. If a FNN needs to work with huge data sets and very big spatial or time complexity problems, it can be very difficult and hard for it to analyze the situation because it would take such a long time.