HW4 WriteUp

Thomas McDonald
University of Colorado – Colorado Springs
tmcdonal@uccs.edu

Abstract

This paper entails my implementation of creating a multilayer nueral net that trains using the Particle Swarming Optimization algorithm from scratch using python.

Approach

This assignment was to create a functional nueral net that can identify letters correctly. To do this we had to create a multilayered nueral net that trains with the Particle Swarming Optimization algorithm. The architecture of my net has 62 input nodes, an initial value of 30 hidden nodes, and 7 output nodes. I chose 62 input nodes to represent each pixel in our 7x9 letter like the one shown below.

```
..##...
...#...
...#...
..#.#..
..#.#..
.#####.
.#...#.
.#...#.
###.###
```

An input node is on (1) if the pixel inputted is anything other than "." and off (-1) otherwise. I used 1 and -1 to represent an on and off state respectively because output values my activation function, a bipolar sigmoid, range from -1 to 1. The output layer of the net is used to identify which letter the net thinks is most probable. I have 7 nodes to identify which of the 7 letters we are identifying (A,B,C,D,E,J,K). For an input A, the ideal signals of the output nodes should be set to (1,-1,-1,-1,-1,-1,-1). For an input B the nodes should have signals (-1,1,-1,-1,-1,-1,-1) and so on. I implemented this project from scratch using python due to personal preferance. The only libraries I'm using is random for the weights of each node, copy to deepcopy objects and math for the activation function. To begin, I made a node object that holds the signal value (-1 to 1). To calculate the signal, I used the bipolar sigmoid function to give me a signal (shown below).

```
##node object for net
class Node():
    def __init__ (self):
        self.signal = 0

def bipolarSigmoid(x):
    return (2 / (1 + math.exp(-1*x ))) - 1
```

To create my Nueral Net, I created a net object that holds an array of arrays, where each array holds nodes. The 0th entry is the array of input nodes, 1st entry for the hidden layer, and 2nd entry for the output nodes. The net object also has a buildNet method to build the net, feed to feed the net, and feedForward which feeds the rest of the net from the input nodes. The weights of the nueral net are stored in particle objects, where each particle has a vector that represents all the weights in the net and the last two values of the vector have the bias for the hidden nodes and the bias for the output nodes respectively. The weights and biases are initialized to be a random number between -1 and 1. Particles also hold a score, the nextVelocity, it's personal best velocity and personal best score. The score is determined by how many letters the particle got correct after reading the entire test document.

```
##particle for pso
class Particle():
    def __init__ (self):
        self.score = 0
        self.nxtVelocity = []
        self.velocity = self.initVel()
        self.pbest = self.velocity
        self.pbestScore = -1000

    def initVel(self):
        temp = (63*inner_nodes) + (inner_nodes* \
        output_nodes)
        arr = []
        for i in range(temp):
            arr.append(random.uniform(-1,1))

        ##biases
        arr.append(random.uniform(-1,1))
        arr.append(random.uniform(-1,1))
        return arr
```

To train the neural net with the particles, I also had to make a swarm object that can manage the particles. The swarm holds the particles, the global best score of the particles, and the vector from the global best particle. The swarm has just one function called update. The update function checks if each particle is better than either the global best or particle's personal best and if so updates the global best of the swarm or the personal best of the particle. Next, it calculates the next velocity for each particle based off the personal and global bests by following the traditional PSO algorithm (shown below); w,c1,and c2 are initialized at the top for easier testing.

```
#next velocity for particle
          for part in self.particles:
               index = 0
               part.nxtVelocity = []
               for i in part.velocity:
                    temp = (w*i) + (c1*random.\
          random()*(part.pbest[index] - i))
                    temp = temp + (c2*random.\
          random()*(self.gbest[index] - i))
                    index = index + 1
                    part.nxtVelocity.append(temp)
```

After checking if the calculated velocity is within our bounds, each particle is then updated with their new velocities for the next round. The overall process of my program goes as followed:

1. Create particles and add them to the swarm
2. Collect the pixels from a letter and feed it to the input layer of nets using weights from each particle
3. The net continues to feed through each layer giving a node a signal till it gets to the output layer
4. Collect the output layers of each net and give the particle tied to that net a score of score + 1 if the net guessed the letter right
5. Once there is no more to feed the nets, update the swarm's global best and each particle's personal best
6. Calculate and set the velocity vector of each particle for the next run and set the score back to 0
7. Repeat 2-5 for however many times you want to update the swarm
8. Use the global best velocity and feed the nueral net with the test values

Experiments

As a base test, I used the following constants:

```
inner_nodes = 30
output_nodes = 7
particles_const = 10
epochs = 100
c1 = 0.98
c2 = 0.60
w = 0.3
bias_range = 1
min_velocity = -100
max_velocity = 100
```

There's nothing special about these values other than this was the setup when my net started showing promising results. To calculate the average accuracy, I use the testing dataset on my net and test it 10 times while summing the reported accuracy of my net and resseting the net after a test.

Results

Running the test with the initial constants I set, I was given an average accuracy of 49.52380952380953. This is mostly due to the number of epochs I used initially, but as you can see below increasing the number of epochs gives much better results

Changing the Number of Epochs

# of Epochs	Avg. Accuracy
100	49.52380952380953
200	61.42857142857141
300	65.71428571428571
400	64.28571428571428
500	65.23809523809524
600	69.04761904761905
700	68.0952380952381

after 700 the runtime increased enough to make testing any further a long wait time.

you can see the lower range number of hidden nodes around 20 has the best accuracy and anything above or below that the accuracy of the net starts to decline.

Changing the Number of particles

# of particles	Avg. Accuracy
10	49.52380952380953
20	62.38095238095237
30	70.0
40	66.19047619047618
50	61.904761904761905

Changing C1

C1	Avg. Accuracy
0.98	49.52380952380953
-1	42.38095238095237
-0.5	46.19047619047618
0.5	56.66666666666667
2	40.47619047619047

Changing C2

C2	Avg. Accuracy
-1	49.52380952380952
-0.5	57.142857142857146
0.6	49.52380952380953
1	54.28571428571429
1.5	39.99999999999999

Changing w

w	Avg. Accuracy
0.3	49.52380952380953
0.5	50.95238095238095
1	38.57142857142857

Changing Bias Range

Bias Range	Avg. Accuracy
1	49.52380952380953
5	53.809523809523796
10	60.0
100	49.52380952380952