Neural Networks

Carlos Matheus Barros da Silva, *Computer Engineering Bachelor Student of ITA*
Prof. Marcos Ricardo Omena de Albuquerque Máximo

***Abstract*—In this paper is the evaluation of an Evolutionary Optimization Method denominated Simple Evolution Strategy (SES) and its performance is compared with the Covariance Matrix Adaptation Evolution Strategy (CMA-ES).**

The results show that since the SES approach not so sophisticated it needs a larger population in order to achieve as good results as CMA-ES gets with a smaller population.

Although in some cases they performed similarly, and with a small increase in population in SES makes it perform as good as CMA-ES, and in some cases, SES with a larger population performed better then CMA-ES. With this, therefore, SES is a simple and effective optimization method.

***Index Terms*—Simple Evolution Strategy, SES, Covariance Matrix Adaptation Evolution Strategy, CMA-ES, optimization**

I. Introduction

NEural networks (NN) are computing systems vaguely inspired by the biological neural networks and astrocytes that constitute animal brains. The neural network itself is not an algorithm, but rather a framework for many different machine learning algorithms to work together and process complex data inputs. Such systems "learn" to perform tasks by considering examples, generally without being programmed with any task-specific rules. For example, in image recognition, they might learn to identify images that contain cats by analyzing example images that have been manually labeled as "cat" or "no cat" and using the results to identify cats in other images. They do this without any prior knowledge about cats, for example, that they have fur, tails, whiskers and cat-like faces. Instead, they automatically generate identifying characteristics from the learning material that they process.

An NN is based on a collection of connected units or nodes called artificial neurons, which loosely model the neurons in a biological brain. Each connection, like the synapses in a biological brain, can transmit a signal from one artificial neuron to another. An artificial neuron that receives a signal can process it and then signal additional artificial neurons connected to it.

In common NN implementations, the signal at a connection between artificial neurons is a real number, and the output of each artificial neuron is computed by some non-linear function of the sum of its inputs. The connections between artificial neurons are called 'edges'. Artificial neurons and edges typically have a weight that adjusts as learning proceeds. The weight increases or decreases the strength of the signal at a connection. Artificial neurons may have a threshold such that the signal is only sent if the aggregate signal crosses that threshold. Typically, artificial neurons are aggregated into layers. Different layers may perform different kinds of transformations on their inputs. Signals travel from the first layer (the input layer), to the last layer (the output layer), possibly after traversing the layers multiple times.

The original goal of the NN approach was to solve problems in the same way that a human brain would. However, over time, attention moved to performing specific tasks, leading to deviations from biology. Artificial neural networks have been used on a variety of tasks, including computer vision, speech recognition, machine translation, social network filtering, playing board and video games and medical diagnosis.

II. Neural Network Implementation

The implementation was based on the file *neural network*. The essence of the implementation is on the methods *forward propagation*, *compute gradient back propagation* and *back propagation*.

Those methods can be seen on the codes from Code 1 to Code 3.

```
def forward_propagation(self, input):
    """
    Executes forward propagation.
    Notice that the z and a of the first layer (l = 0)
 are equal to the NN's input.

    :param input: input to the network.
    :type input: (num_inputs, 1) numpy matrix.
    :return z: values computed by applying weights and
 biases at each layer of the NN.
    :rtype z: 3-dimensional list of (num_neurons[l], 1)
 numpy matrices.
    :return a: activations computed by applying the
 activation function to z at each layer.
    :rtype a: 3-dimensional list of (num_neurons[l], 1)
 numpy matrices.
    """
    z = [None] * 3
    a = [None] * 3

    z[0] = input
    a[0] = input

    z[1] = self.weights[1]*a[0] + self.biases[1]
    a[1] = sigmoid(z[1])

    z[2] = self.weights[2] * a[1] + self.biases[2]
    a[2] = sigmoid(z[2])

    return z, a
```

Code 1. Code of *forward propagation* method

```
def back_propagation(self, inputs, expected_outputs):
    """
    Executes the back propagation algorithm to update
 the NN's parameters.

    :param inputs: inputs to the network.
    :type inputs: list of numpy matrices.
    :param expected_outputs: expected outputs of the
 network.
    :type expected_outputs: list of numpy matrices.
    """
    weights_gradient, biases_gradient = self.
 compute_gradient_back_propagation(inputs,
 expected_outputs)
```

```
        self.weights[1] -= self.alpha * weights_gradient[1]
        self.weights[2] -= self.alpha * weights_gradient[2]

        self.biases[1] -= self.alpha * biases_gradient[1]
        self.biases[2] -= self.alpha * biases_gradient[2]
```

Code 2. Code of *back propagation* method

```
    def compute_gradient_back_propagation(self, inputs,
 expected_outputs):
        """
        Computes the gradient with respect to the NN's
 parameters using back propagation.

        :param inputs: inputs to the network.
        :type inputs: list of numpy matrices.
        :param expected_outputs: expected outputs of the
 network.
        :type expected_outputs: list of numpy matrices.
        :return weights_gradient: gradients of the weights
 at each layer.
        :rtype weights_gradient: L-dimensional list of
 numpy matrices.
        :return biases_gradient: gradients of the biases at
  each layer.
        :rtype biases_gradient: L-dimensional list of numpy
  matrices.
        """
        weights_gradient = [None] * 3
        biases_gradient = [None] * 3
        weights_gradient[1] = np.zeros((self.num_hiddens,
 self.num_inputs))
        weights_gradient[2] = np.zeros((self.num_outputs,
 self.num_hiddens))
        biases_gradient[1] = np.zeros((self.num_hiddens, 1)
 )
        biases_gradient[2] = np.zeros((self.num_outputs, 1)
 )

        num_cases = len(inputs)
        outputs = [None] * num_cases

        for i in range(num_cases):

            z, a = self.forward_propagation(inputs[i])
            outputs[i] = a[-1]

            y = expected_outputs[i]
            y_hat = outputs[i]
            inp = inputs[i]

            dz2 = y_hat - y
            dw2 = np.dot(dz2, np.transpose(a[1]))
            db2 = np.sum(dz2, axis=1)

            dz1 = np.multiply(np.dot(np.transpose(self.
 weights[2]), dz2), sigmoid_derivative(z[1]))
            dw1 = np.dot(dz1, np.transpose(inp))
            db1 = np.sum(dz1, axis=1)

            biases_gradient[1] += db1 / num_cases
            biases_gradient[2] += db2 / num_cases

            weights_gradient[1] += dw1 / num_cases
            weights_gradient[2] += dw2 / num_cases

        return weights_gradient, biases_gradient
```

Code 3. Code of *compute gradient back propagation* method

III. Neural Network Analysis on Test Function

In this analysis was used two test functions: *sum greater than* and *xor*.

The Neural Network performed very well on *sum greater than* function. On the graphs represented by the images from Image 1 to Image 1

The Neural Network performed well on *xor* function. On the graphs represented by the images from Image 4 to Image 6. It is also possible to see that in this case it heapped some overfit causing some distorsion and leading to some mistakes on the data set on the graph represented by the Image 6.

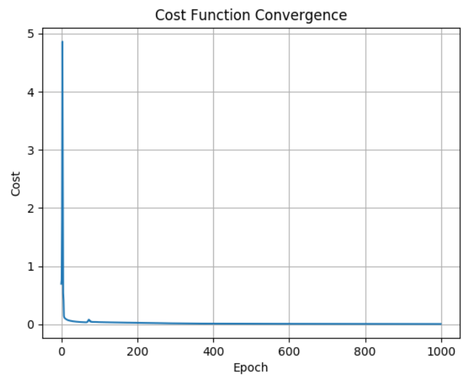

Fig. 1. Convergence of cost function on greater than function test case

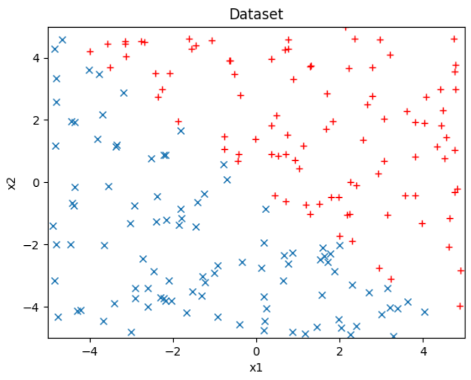

Fig. 2. Dataset of greater than function

A. Neural Network Analysis on Color Segmentation

In order to do a color segmentation, this Neural Network performed very well. This performance can be seen on the results figures on from Image 7 to Image 9.

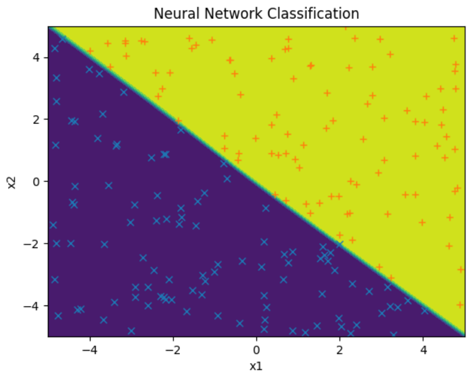

Fig. 3. Neural Network Classification on greater than function

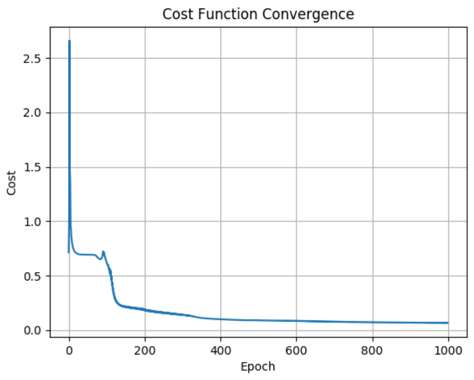

Fig. 4. Convergence of cost function on xor function test case

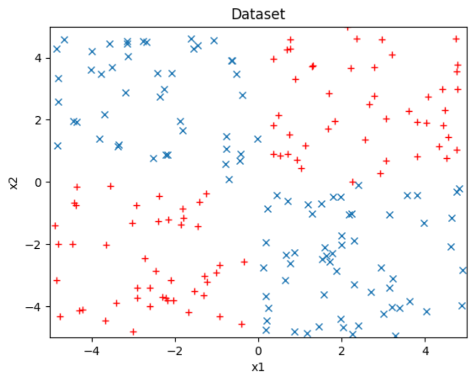

Fig. 5. Dataset of xor function

IV. Conclusion

It was clear, therefore, that the Neural Network worked as expected. Both test cases (greater than function and xor function) the Neural Network worked as expected.

For the color segmentation, the Neural Network also provided a concise well segmented result.

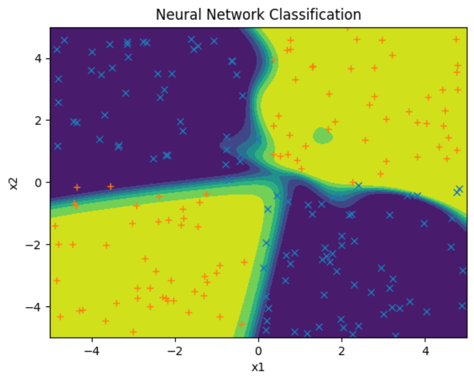

Fig. 6. Neural Network Classification on xor function

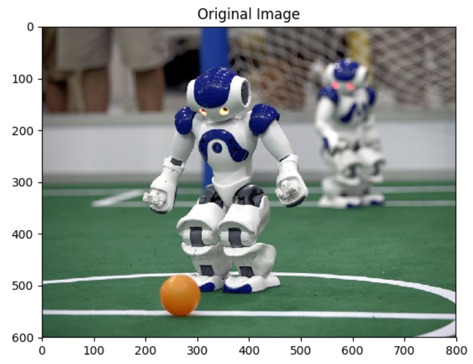

Fig. 7. Original image, before segmentation

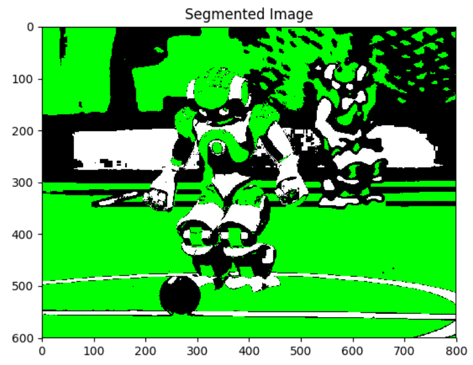

Fig. 8. Segmented image output

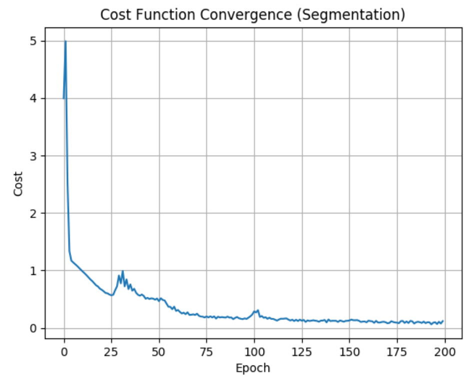

Fig. 9. Neural Network convergence of cost function on image color segmentation problem