Tuning Neural Network Parameters Through a Genetic Algorithm

Luca Gazzola
University of Milano-Bicocca
luca.gazzola@disco.unimib.it

ABSTRACT

Artificial Neural Networks (ANNs) have become increasingly popular during the recent years, modern hardware technology and the availability of labeled data allow developers to exploit the power of this tool that has been studied (at least as a mathematical model) since the the middle of last century. Recently many uses have been found for ANNs, spanning from decision making to pattern recognition, automated trading systems, and so on. One of the most famous (and successful) applications of ANNs is image recognition: classifying images as part of specific domain classes is an interesting problem that can find application in many different fields. An ANN needs to be trained before it is used to classify, this procedure depends on the quality of the training data but also on specific parameters that can be tuned to optimize the training procedure. In this report we explore a way to optimize the network training procedure (for both efficiency and effectiveness) by selecting the optimal parameter through evolution via genetic algorithm.

Keywords

neural network, genetic algorithm, image recognition

1. NEURAL NETWORKS

Artificial Neural Networks (ANNs) are a powerful computing tool, loosely based on how the human brain works. An interesting characteristic of ANNs is that they are not programmed to perform any specific task, but instead they learn their expected behavior from training examples. In Figure 1 we can see the general architecture of an ANN: on the left side we see the input layer where the appropriately encoded data can be fed to the network, on the right side we see the output layer where the network computation is returned and in the middle we can see one or more hidden layers, and each of these layers is connected to the previous one. Each layer is composed by neurons, which essentially compute a function of their inputs and forward the result on the output. To understand the concept of neurons we can use a simple implementation called perceptron. A perceptron is a very simple type of neuron that has one or more inputs or more and a single output that can be either 0 or 1. The perceptron in figure 2 for example, has three inputs and one output, the output value is simply computed by:

$$output = \begin{cases} 0, & \text{if } \sum_i w_i x_i + b_i \leq 0 \\ 1, & \text{if } \sum_i w_i x_i + b_i > 0 \end{cases} \quad (1)$$

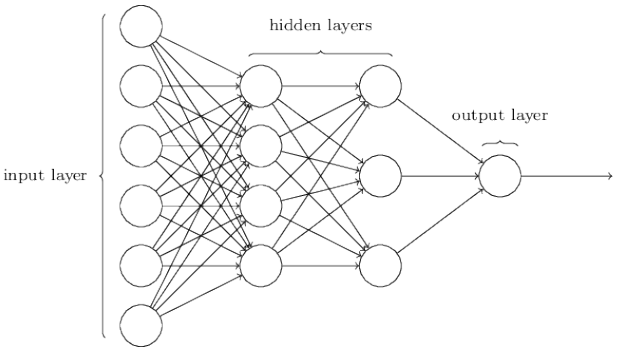

Figure 1: General neural network architecture

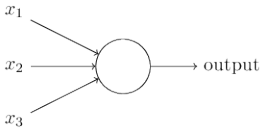

Figure 2: Architecture of a perceptron

where x_i is the i^{th} input, w_i is the i^{th} weight applied to the i^{th} input and b_i is called the activation value. A perceptron can be thought of as a device that makes decisions by weighing up evidence. Assuming a network is made of perceptrons, (1) can be applied to each perceptron that composes the network to calculate the output.

The perceptron is a very simple implementation of a neuron, a more powerful (and widely used) implementation is the sigmoid neuron, which instead of a step function (the output is 0 or 1) computes a sigmoid function on the input:

$$output = \frac{1}{1 + e^{-\sum_i (w_i x_i) - b_i}} \quad (2)$$

Sigmoid neurons allow the network to be more flexible, since a little change in the weights and biases corresponds to a little change in the output of the network. Changing the weights and the biases is indeed the procedure through which the network "learns" its intended behavior. There are multiple procedures to adjust the weights and biases to obtain a better output, but in general they always involve calculating the difference between the expected output (given by the training data) and the actual output, and then using an algorithm to adjust the weight and bias of single neurons

to minimize that difference. Such difference is often called *cost* or *loss*, and can be calculated as:

$$C(w,b) = \frac{\sum_x |y(x) - z|^2}{n} \tag{3}$$

where n is the number of training inputs, z is the output vector of the network when the input vector is x and $y(x)$ is the desired output (note that there are many variations of cost functions, this is just an example). The aim of the training phase of a network is to minimize $C(w,b)$. This is often done using variations of the *gradient descent* algorithm, which modifies the weights and biases in order to decrease the cost function:

$$w'_k = w_k - \mu \frac{\partial C}{\partial w_k} \tag{4}$$

$$b'_k = b_k - \mu \frac{\partial C}{\partial b_k} \tag{5}$$

where μ is called *learning rate*. These weights and biases changes are calculated in a first phase for the output layer and then for each hidden layer in the network, starting from the ones close to the output, until the input layer is reached. This procedure is repeated until no more examples in the training data are wrongly classified or some other stopping criteria is met. This algorithm that is used to update the weights and biases of the network is called *backpropagation*.

In the next chapter we will describe the implementation of a simple network that classifies hand written digits and explain how it works.

2. IMAGE RECOGNITION WITH NEURAL NETWORKS

We will describe here the python implementation of a simple network built to classify hand written digits. The code of the implementation can be found at [1]. To train the network and test it we will use the MNIST dataset of handwritten digits [3].

The python class that implements the network is called *network.py*, the network initialization is done as shown in the code below:

```
import numpy as np
class Network(object):

def __init__(self, sizes):
  self.num_layers = len(sizes)
  self.sizes = sizes
  self.biases = [np.random.randn(y, 1) for y in
       sizes[1:]]
  self.weights = [np.random.randn(y, x) for x,
      y in zip(sizes[:-1], sizes[1:])]
```

sizes must be an array containing the number of neurons in each layer, starting from the input and ending with the output layer; self.biases and self.weights represents the biases and weights of the network, and are randomly generated. To create a network that can classify the examples in the MNIST dataset we will have 784 neurons in the input layer (28p x 28p images, each neuron taking as input the value of a pixel) and 10 neurons in the output layer (10 possible digits to recognize), hence a possible initialization for such network is:

```
Network([784,20,10])
```

the created network will have 784 input neurons, 20 neurons in a single hidden layer and 10 output neurons.

The *sigmoid* function that computes the output of single neurons is defined as:

```
def sigmoid(z):
    return 1.0/(1.0+np.exp(-z))
```

where z is an array of numbers (the sigmoid function is applied element-wise). To calculate the output of the network we define the *feedforward* function:

```
def feedforward(self, a):
  for b, w in zip(self.biases, self.weights):
    a = sigmoid(np.dot(w, a)+b)
  return a
```

this function simply works by applying the sigmoid function element-wise to the vector $wa + b$ for each layer.

The function that implements the learning of the network is the one that implements stochastic gradient descent:

```
def SGD(self, training_data, epochs,
    mini_batch_size, eta):
  n = len(training_data)
  for j in xrange(epochs):
    random.shuffle(training_data)
    mini_batches = [
      training_data[k:k+mini_batch_size]
      for k in xrange(0, n, mini_batch_size)]
    for mini_batch in mini_batches:
      self.update_mini_batch(mini_batch, eta)
```

the stochastic gradient descent differs from the traditional gradient descent algorithm in the fact that the gradient is computed only for a small sample of randomly chosen training inputs (called mini-batch), hence speeding up the weight adjustment process.

Let's break down the *SGD* function listed above: *training_data* is a list of tuples representing inputs and expected outputs, *mini_batch_size* and *epochs* represent the size of the subset of training inputs used to calculate the gradient of the cost function and the number of epochs to train for; *eta* is the learning rate.

Each epoch starts by randomly shuffling the training data and partitioning it into mini-batches, for each mini batch the function *update_mini_batch* is called. This function applies gradient descent with the mini-batch supplied as argument, updating the network weight and biases. This is repeated for the number of epochs set.

3. GENETIC ALGORITHMS

Genetic algorithms (GAs) are a family of computational models inspired by evolution [4], where only the characteristics of the fittest individuals survive to future generations.

A GA is a meta-heuristic used to generate solutions to optimization and search problems. The foundations of GAs were originally developed by Holland [2].

These algorithms provide a way to encode a potential solution to a problem on a simple data structure that represents a *chromosome* or *gene*.

During the evolution, the subjects with fitter characteristics will have better chances of surviving and reproducing, on the contrary, subjects with worst characteristics will be eliminated. Genes from successful subjects will be preserved along evolution, in order to improve the fitness of the species.

Similarly, during the searching process, we modify individuals representing our possible solutions, in order to find the fittest one after a certain number of iterations.

The alterations to solutions during iterations might be drastic or moderate, depending on the objective of the search problem.

A typical genetic algorithm requires a (1) genetic representation of the solution domain and a (2) fitness function to evaluate the solution domain.

3.1 How Genetic Algorithms Works

The general flowchart of GAs is shown in Figure 3.

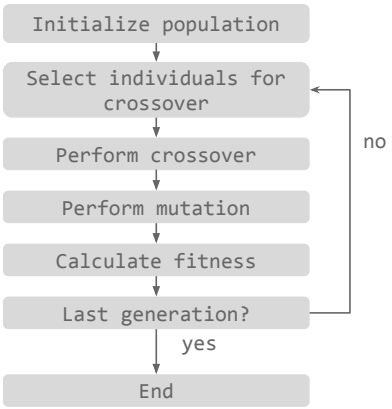

Figure 3: General genetic algorithm flowchart

Genetic Algorithms emulate the evolution process by (1) initializing a population of random individuals, (2) selecting individuals (parents) for reproduction, (3) performing a crossover procedure between both individuals (exchange of pieces of their genetic information) to produce new solutions (children) that share some characteristics taken from both parents. (4) Mutating some genes in the chromosomes, to alter the composition of the data structure. At this point (5) the fitness of the offspring is estimated, to evaluate the quality of the actual solution. The iteration of the algorithm finishes either when an optimal solution has been reached, or when an established number of iterations of reproduction has been performed.

4. TUNING NEURAL NETWORK PARAMETERS WITH A GENETIC ALGORITHM

Every ANN needs to be trained before it can be used to classify elements. We are aware that the quality of the training session depends largely on the quality of the data set used to train the neural network. But, choosing correctly the parameters of an ANN is also a very important aspect, if we are looking for faster training sessions that have lower error rates.

How to select correctly the training parameters of an ANN might be approached as an optimization problem, where each parameter is considered as a *gene*, and the set of parameters is considered as an individual that should be improved by applying genetic operators to obtain the best individual, or solution over the time.

4.1 Genetic representation

The genetic representation for the ANN presented in Section 2 is given by a fixed-length array of size four, this array represent the four parameters to be optimized by the genetic algorithm. Figure 5 shows the information contained in each index; the first index α, contains the number of neurons allocated in the middle layer, the second index γ is the number of epochs for the training session, the third index σ shows the mini-batch size and δ expresses the optimal learning rate for the neural network. It is worth saying that all variables are integers except for δ that is a floating point variable.

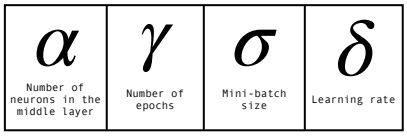

Figure 4: Genetic representation of the optimization problem

4.2 Fitness function

The fitness function that optimizes the parameter values is represented by:

$$F(\beta, \tau) = \beta + \tau \tag{6}$$

where, β is the number of misclassified examples by the neural network, and τ is the time spent by the neural network to train on the training set.

In order to find the optimal solution, this fitness function should be *minimized*.

Since this is a initial approach to an optimal solution, we are giving the same importance by weighting equally both parameters of the fitness function.

Depending on the objective of practitioners, it might be more important the precision of the technique rather than the performance, for example.

4.3 Crossover

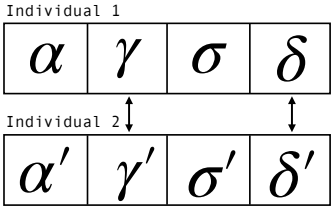

Figure 5: Crossover operator

When two possible solutions reproduce, they perform a crossover procedure in which they exchange genetic information to create new offspring with both parent characteristics.

In Figure 5 we show how the crossover is done; since the idea is that new offspring maintain information from parents, we create two children, one keeping the information from parent one, and the other keeping information about parent two. At last, between children we exchange γ and δ information, to have two new individuals for future iterations of the algorithm.

4.4 Mutation

Since is very important to explore the search space for finding different solutions, we introduce mutation as a genetic operator through iterations.

Basically, after the crossover procedure for each *gene* in the array we apply an increment or a decrement of one for α, γ and σ, and a variation of 0.1 for δ index.

The mutation for each element is decided probabilistically; with a 10% probability we modify the value of the index, and with a probability of 50% we increase or decrease the value itself.

4.5 Elitism

Elitism is a strategy to keep the best individual (*i.e.* elite) of the population in each generation of the algorithm, so we do not lose the best solution through iterations.

After that crossover and mutation operators are applied to the population, we estimate the fitness of each individual and rank them according to the quality of their fitness value. If the new best individual in the population is worst than the previous generation elite individual, we randomly replace an individual of the population with our current elite solution. Otherwise, we denominate the current best individual as the new elite for future generations.

4.6 Genetic Algorithm

Our genetic algorithm starts by creating a random population of five individuals, representing five artificial neural networks with the four parameters explained in Section 4.1.

We also define as a terminating condition five generations for the search procedure. Later, in each generation we perform two crossover operations, meaning that from the five individuals we select two pairs of parents to be reproduced. After crossover finishes, we then apply mutation to the entire population. Before the algorithm ends, we look for our best solution (elitism) and proceed with the search until we finish our search budget.

Further, we will see what happens when modifying the values of the population size, number of generations and number of crossovers.

5. EXPERIMENTATION AND RESULTS

In this section we will discuss the results obtained by applying the described approach to tune neural network parameters. The tool used to get the presented results is available for download at [1].

5.1 Experimental Setup

The genetic algorithm that tunes the network parameters is implemented in a single python class: *genetic.py*. *network.py* implements the network that is trained on the MNIST dataset, loaded through mnist_loader.py. When the genetic algorithm is launched, four parameters can be used to configure it:

- population size
- number of generations
- number of crossovers per generation
- mutation probability

We ran the genetic algorithm with ten different configurations, obtaining the results reported in Table 1. The experiments were conducted on a quad-cpu Intel Xeon E7540 with 32 GB DDR3 1066 MHz RAM.

5.2 Results

In Table 1 we can see the results obtained for the various experiment configurations in the columns *Fittest individual* and *Score*. *Fittest individual* represents the fittest individual in the population: remember that an individual is represented as (number of neurons in the mid layer, epochs, mini-batch size, learning rate). *Score* represents the fitness value that the fittest individual achieved (remember that the fitness score is calculated as *training time+number of wrongly classified examples*, so we are interested in minimizing the score).

We can see from the table that the best results achieved were obtained with a population size of 30 individuals, 40 generations and 5 crossovers per generation. Compared to the best results from the other experiments, we can see that the fittest individual has the highest number of mid layer neurons and training epochs, but an average learning rate and a relatively small mini-batch size.

The fittest individual found was also the best at classifying the hand-written digits, with 9549 images over 10000 correctly classified, and a total training time of 97 seconds.

6. DISCUSSION

The idea of tuning the network parameters with a genetic algorithm can be useful to provide an initial configuration to train the network, leaving the fine tuning to more advanced techniques. In this work we also focused on how fast the neural network is trained, using the total training time as a variable in the calculation of the fitness function. We did this because, thinking that the parameters found in the results can generalize to other types of neural networks and classification problems, it might be useful to have a network that is trained fast up to a certain level of precision and recall, and then can be fine tuned from there. Further experiments are needed to understand the best configuration of the genetic algorithm's parameters for this problem, trying different mutation probabilities and selection strategies might also improve the results.

7. REFERENCES

[1] L. Gazzola. Tuning network parameters through a genetic algorithm. https://github.com/lucaGazzola/neural-networks-and-deep-learning.

[2] J. H. Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[3] Y. LeCun, C. Cortes, and C. J. Burges. Mnist handwritten digits database. http://yann.lecun.com/exdb/mnist/.

[4] D. Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.

Experiment	Population Size	Generations	Crossovers	Mutation prob.	Execution Time	Fittest individual	Score
1	5	5	2	0.1	3182s	(37, 8, 22, 3.19)	707
2	5	10	2	0.1	6415s	(37, 6, 19, 4.39)	659
3	5	20	2	0.1	9298s	(38, 5, 18, 1.93)	750
4	10	10	2	0.1	10551s	(32, 8, 14, 0.9)	722
5	10	20	2	0.1	18874s	(36, 4, 5, 2.89)	621
6	10	30	2	0.1	26766s	(37, 9, 11, 3.39)	632
7	10	40	2	0.1	34223s	(28, 7, 13, 2.43)	612
8	20	30	4	0.1	67961s	(39, 8, 7, 3.51)	611
9	20	40	4	0.1	83212s	(27, 5, 16, 3.14)	604
10	30	40	5	0.1	131920s	(40, 10, 11, 3.12)	547

Table 1: Experiment results