International Journal of Health Technology and Innovation **RESEARCH ARTICLE**
An Official Journal of **Kalam Institute of Health Technology** | ISSN - **2583-8547**

Comparative Analysis of Backpropagation and Genetic Algorithms in Neural Network Training

Ayoub Hazrati, Shannon Kariuki, Ricardo Silva*

Department of Computer Science, Villanova University, United States

Received: 08th August, 2024; Revised: 24th September, 2024; Accepted: 28th November, 2024; Available Online: 05th December, 2024

ABSTRACT

The exploration of artificial neural networks (ANNs) has seen significant advancements, yet the optimal approach for training these networks remains a topic of debate. This study investigates the efficacy and computational efficiency of two prominent optimization techniques, backpropagation and genetic algorithms (GAs), in training traditional neural networks. The research conducted three experiments: the first involved training a single-layer neural network using a simple mathematical function; the second utilized the diabetes dataset for regression analysis; and the third applied the iris dataset for multi-class classification. The networks were trained using Google Colab, leveraging generative AI tools to expedite the experimentation process. Results indicate that backpropagation consistently achieved lower mean square error (MSE) in shorter training times compared to GAs, especially in high-dimensional data. GAs demonstrated robustness in escaping local minima, making them suitable for complex, noisy datasets where backpropagation might converge prematurely. The study concludes that while backpropagation is preferable for tasks requiring precision and speed, genetic algorithms offer valuable advantages in explorative scenarios, highlighting the importance of task-specific algorithm selection in neural network training.

Keywords: Artificial neural networks, Backpropagation, Genetic algorithms, Supervised learning, Optimization techniques.

International Journal of Health Technology and Innovation (2024)

How to cite this article: Hazrati A, Kariuki S, Silva R. Comparative Analysis of Backpropagation and Genetic Algorithms in Neural Network Training. International Journal of Health Technology and Innovation. 2024;3(3):18-25.

Doi: 10.60142/ijhti.v3i03.04

Source of support: Nil.

Conflict of interest: None

INTRODUCTION

The quest to unravel the mysteries of the brain's inner workings traces its origins back to Santiago Ramon y Cajal's groundbreaking publication in 1894.[1] Born in Navarra, Spain, in 1852, Cajal possessed exceptional artistic talent and chose to embark on a journey into the field of medicine, guided by his father's expertise. By 1883, Cajal had risen to the position of a professor of anatomy at the University of Valencia, publishing works on microanatomy, with a particular focus on the intricate nervous system. Collaboration with Camillo Golgi earned them the Nobel Prize in Physiology or Medicine in 1906.

Cajal's intricate illustrations, combined with the emergence of electrophysiology, a branch of physiology dedicated to exploring the electrical properties of biological cells and tissues, laid the groundwork for the development of various models. These models aimed to elucidate how individual neurons respond to stimuli and how neural networks have the capacity to learn. These early endeavors mark the inception of ANN. It is imperative to acknowledge, however, that despite the impressive computational power achieved by contemporary ANN models, they still pale in comparison to the immense complexity of the human brain.

ANNs are computational models inspired by the structure and function of the human brain. They consist of interconnected nodes, called neurons, organized in layers. Neurons are the fundamental building blocks of natural neural networks (NNN). Neurons come in various types, but the archetypal neuron is a specialized cell designed to process and transmit information. The neuron comprises three primary parts: Dendrites, Soma, and Axon, as shown in Fig. 1.

Dendrites serve as inputs, collecting information from external receptors or other neurons in the form of neurotransmitters. These neurotransmitters are small molecules that carry information and are stored in vesicles within the presynaptic cell's synaptic button. When a neurotransmitter is released from the presynaptic button, it crosses the synaptic cleft and binds to neurotransmitter receptors in the postsynaptic cell. This interaction can either

**Author for Correspondence:* ricardo.silva@villanova.edu*

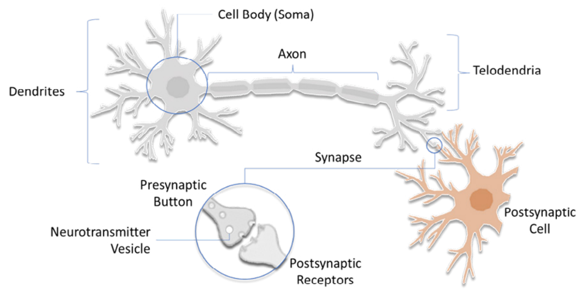


Fig. 1: Prototypical neural cell, comprised of: Dendrites, soma, and axon

excite (positive) or inhibit (negative) the postsynaptic cell, generating a partial response termed irritability. Notably, a single synapse's irritability alone is insufficient to trigger a full response in the receiving cell. Instead, it is the cumulative effect of tens to thousands of synapses, encompassing both excitatory and inhibitory signals, that is integrated in the neuron's soma.

If the combined influence of positive and negative signals from the presynaptic cells surpasses a threshold, the postsynaptic cell generates an electrical response known as an action potential (AP). This AP travels along the axon, branching into multiple terminals through a treelike structure called the telodendria. Each telodendron terminates in a synaptic button containing stored neurotransmitters. When the AP reaches the synaptic button, neurotransmitter is released into the synaptic cleft, where it is received by the postsynaptic cell.

Artificial Neural Networks (ANN)

The origins of ANNs can be traced back to the early 1940s, marked by the collaboration between Warren McCulloch, an American neurophysiologist and cybernetician at the University of Illinois at Chicago, and Walter Pitts, a self-taught logician and cognitive psychologist. Their groundbreaking work introduced the concept of the "McCulloch-Pitts neuron.[2]" This pioneering endeavor gave birth to the first mathematical model of a neural network.

In 1949, Canadian psychologist Donald O'Hebb introduced a seminal concept in the field of neural learning, proposing a fundamental hypothesis for biological neurons that would become widely recognized as Hebbian theory.[3] Hebb's theory posited that neural connections between neurons strengthen with activity. This principle can be summarized as "Neurons that fire together wire together," emphasizing the role of repeated and persistent neural activity in shaping the brain's connectivity. This concept is commonly referred to as Hebbian Learning and has played a significant role in the development of artificial neural networks and neurobiology.

In 1958, a significant milestone in the field of artificial neural networks was achieved with the development of the perceptron by Frank Rosenblatt, an American psychologist often referred to as the father of deep learning. The original perceptron was created as an electronic device and was initially simulated on an IBM 704 computer at Cornell Aeronautical Laboratory.[4]

Rosenblatt's groundbreaking work on perceptron's laid a strong foundation for subsequent developments in neural network theory, and its principles continue to exert a significant influence in the field of deep learning.[5] Remarkably, the perceptron, as shown in Fig. 2, with relatively minor modifications, remains at the core of modern ANNs. The perceptron operates on a relatively straightforward principle: it takes input signals and multiplies them by corresponding weights, which represent the strengths of synaptic connections. These weighted inputs are then aggregated, usually through addition, and the result is compared to a predefined threshold within the activation function. When this aggregated value surpasses the threshold, it triggers an activation signal, typically represented as a Digital 1. This fundamental concept of signal processing and decision-making within a neural network has endured and evolved into the complex and powerful deep learning models that underpin many of today's Artificial Intelligence (AI) applications.

Like natural neurons, a single perceptron possesses limited computational utility. Perceptron's are interconnected to form layers, with data typically flowing from left to right, As shown in the Fig. 3. The leftmost layer represents the input layer, where each neuron corresponds to the data to be analyzed, such as text or medical images. One or more hidden layers, where most of the ANN's processing capability resides, can exist. On the far right is the output layer, which typically represents desired outcomes, like identifying whether an image features a human or an animal or determining the language of a document. Present-day deep learning systems may comprise thousands of hidden layers, each containing numerous neurons.

Training Artificial Neural Networks

The process of training artificial neural networks is a fundamental concept in machine learning that involves refining and optimizing the network's parameters to perform a specific task effectively. One crucial milestone in the development of training methods for neural networks was the introduction of backpropagation of errors, which was pioneered by Paul Werbos, an American social scientist and machine learning innovator, in his 1974 dissertation.[6]

Backpropagation can be likened to solving inverse problems. It works by iteratively adjusting the network's internal parameters, such as weights and biases, based on the discrepancies, or errors, between the network's predictions and the desired or target outputs. The goal is to minimize these

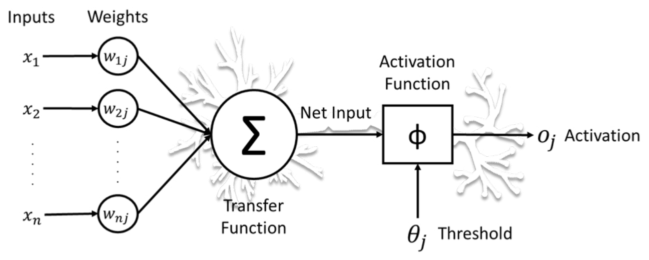


Fig. 2: The perceptron processes input signals and activates if the sum exceeds a threshold.

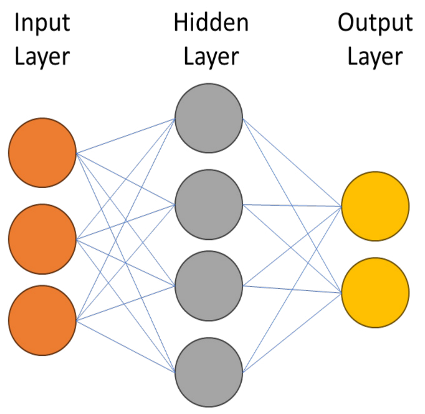


Fig. 3: Artificial Neural Network consists of an input layer that receives data, hidden layers that process and output layer that produces the result

errors, effectively training the network to make more accurate predictions over time.

For example, in a supervised learning scenario where the objective is to train an artificial neural network to distinguish between human and non-human images, a training set of labeled images is used. This training set comprises images that have been previously evaluated by human experts to determine whether each image depicts a human (positive class) or a non-human (negative class). To ensure effective training, it is essential to have a balanced training set, which includes an equal number of human and non-human images. The backpropagation algorithm then adjusts the network's parameters during training to reduce the errors in its predictions, gradually improving its ability to correctly classify new, unseen images. The objective is to determine the precise weight required for each connection to confidently classify human images.

Once the network is initialized, training data is passed through the model from input to predicted output in a feedforward step. Initially, the network's accuracy is low. Accuracy is typically assessed by computing the squared difference between predictions and targets. Backpropagation is responsible for calculating the error (loss function) concerning the network's weights for each input-output example. Derivatives of the error are found, and weights are adjusted based on these derivatives, gradually approaching the configuration that minimizes error. This iterative process is known as "backpropagation." After multiple iterations, the network becomes potentially trained and can be tested.

Testing involves using a new set of images, not part of the training data but meeting the same criteria. The system's accuracy is evaluated, and if it meets expectations, it is considered "trained" and can be applied to real data. If it falls short of expectations, additional training is required, possibly involving an expanded testing dataset. The advantage is that subsequent training does not begin with random weights, making convergence faster. Once an ANN is trained for a specific task, it can only perform that task and requires new training for different functions, such as distinguishing dogs from cats using a similar architecture but distinct training data.

Remarkably, the backpropagation algorithm used in training ANNs relies on the chain rule, a fundamental concept in calculus that was first formulated by the renowned German mathematician, philosopher, scientist, and diplomat Gottfried Wilhelm Leibniz in 1673. Leibniz is often referred to as "The Last Universal Genius" due to his significant contributions to various fields of knowledge.

The chain rule, as formulated by Leibniz, states that if y is a differentiable function of u, and u is a differentiable function of x, then y is a differentiable function of x, and the derivative of y with respect to x can be calculated as:

This rule is a fundamental tool in calculus for finding the derivatives of composite functions, and it plays a crucial role in various branches of mathematics and science, including calculus, physics, and machine learning.

The utilization of the chain rule in backpropagation allows neural networks to efficiently compute gradients, which are essential for adjusting the network's parameters during training. By propagating errors backward through the network and applying the chain rule, backpropagation enables the network to learn and improve its performance over time, making it a foundational technique in the field of ANN and deep learning.

$$\frac{dy}{dx} = \frac{dy}{du} \times \frac{du}{dx}$$

Genetic Algorithms

GAs are inspired by the process of natural selection and genetics, introduced by John Holland in the 1970s.[7] In the context of GAs, we begin with a population, which consists of a set of potential solutions to a given optimization problem. Each solution within this population is represented as a chromosome, which can be encoded as a string of bits, numbers, or characters, depending on the specific problem being addressed.

A chromosome is further divided into genes, each representing a particular variable or attribute of the solution. The overall quality or performance of these solutions is evaluated using a fitness function. This function assigns a fitness score to each individual based on how well it solves the problem at hand, in the same way that accuracy determines the efficacy of back propagation.

The selection process is critical as it determines which individuals are chosen to reproduce and pass their genetic material to the next generation. This selection is typically biased towards individuals with higher fitness scores, akin to selecting the best candidates for breeding in natural evolution.

Following selection, crossover (or recombination) is employed to combine the genetic information of two parents to produce new offspring. This genetic operator mimics biological reproduction, creating new solutions that inherit traits from both parents, thus fostering genetic diversity and innovation within the population.

To ensure the population does not converge prematurely to suboptimal solutions, a mutation operator is introduced. Mutation involves making random alterations to some genes in the chromosomes, thereby maintaining genetic diversity and enabling the exploration of new areas in the solution space.

The Genetic Algorithm operates through an iterative process, beginning with the initialization phase. During this phase, an initial population of potential solutions is generated randomly, establishing a diverse foundation for the evolutionary process.

In the subsequent evaluation step, each individual in the population is assessed using the fitness function. This evaluation determines the fitness score of each solution, providing a basis for comparison and selection.

The selection phase follows, wherein individuals are chosen based on their fitness scores to serve as parents for the next generation. The selection process is designed to favor individuals with higher fitness, ensuring that advantageous traits are propagated.

Next, the crossover operator is applied. This step involves combining pairs of selected parents to produce new offspring, thereby introducing new combinations of genetic material into the population. Crossover promotes the exchange of beneficial traits and enhances the overall adaptability of the population.

To prevent genetic stagnation and encourage exploration of the solution space, the mutation operator is employed. Mutation introduces random changes to some genes in the offspring, ensuring genetic diversity is maintained and novel solutions are explored.

After generating the new offspring, the replacement phase takes place. During this phase, the new population is formed by replacing some or all of the old individuals with the newly created ones. This iterative cycle of evaluation, selection, crossover, mutation, and replacement continues until a predetermined stopping criterion is met, such as achieving a satisfactory solution or reaching a maximum number of generations.

Through this evolutionary process, the Genetic Algorithm incrementally refines the population of solutions, progressively converging towards an optimal or near-optimal solution to the problem. The principles of natural selection, recombination, and mutation are thus harnessed to explore and exploit the solution space effectively.

While there are multiple instances of research exploring the combination of genetic algorithms and neural networks,[8] particularly in areas like neural network architecture optimization,[9] direct application of genetic algorithms to train the weights of a neural network is not common and for a very long time has been assumed not to be an efficient approach.[10]

The primary reason for this is the complexity of the neural network weight space. Genetic algorithms, while effective in discrete search spaces, struggle to efficiently explore the continuous, high-dimensional space of neural network weights. Backpropagation, on the other hand, is specifically designed for this task.[8,10]

Purpose

High performance computing and computation as a service has made available immense amounts of computational capacity that had previously been missing. The goal of this project is to investigate the difference of two optimization approaches, (backpropagation and genetic algorithms) in terms of performance and running time on various traditional artificial neural networks. For this purpose, we trained traditional neural networks with genetic algorithms and backpropagation for classification and regression on simulated datasets and publicly available datasets. For implementation of these experiments, we used google Collab which empowered with various Gen AI tools such as Collab AI and Gemini. Using these tools helps us to do more experiments in faster time.

Methods

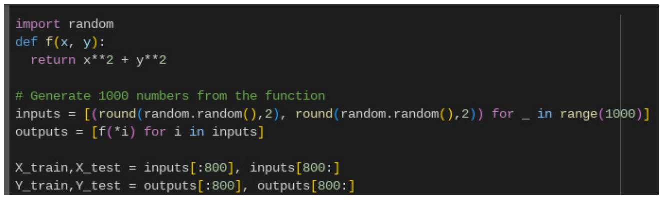
We investigate the effects of backpropagation vs genetic algorithms for fine tuning traditional neural networks. Three different experiments were performed, in the first we generated sample data from simple differentiable mathematical functions. For the second and third experiments, we used publicly available datasets with some level of noise, to compare efficacy of training and convergence.

Simple Mathematical Function

For the first experiment we trained a simple one-layer neural network with sigmoid activation function, to solve a two variable mathematical function. This function is $f(x,y) = x^2 + y^2$ and we generated 1000 data points. We used 80% of the data for training the models and 20% for testing. The code for generating the data is shown in Fig. 1. The code for training neural network with genetic algorithm is shown in Fig. 4 and the code for training neural network with backpropagation is shown in Fig. 5 and with genetic algorithms in Fig. 6. In this example we choose not to use any deep learning packages and use only python math library, so the comparison of both approaches could be fair.

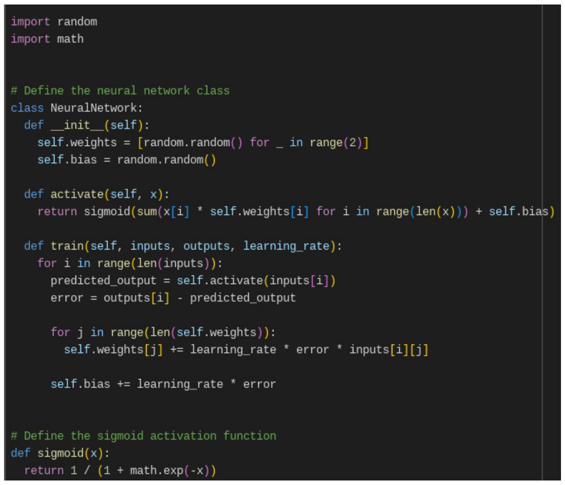
Experiment on Diabetes Dataset

The second experiment that we conducted was with the public diabetes dataset for Machine learning. The dataset has 10 features and one numeric output. Total number of observations in this dataset is 442 and we partitioned into train and testing data. The training size was 80% of the data. The full description of the dataset and the meaning of its features is documented in the sklearn documentation (https://scikit-learn.org/stable/datasets/toy_dataset.html).



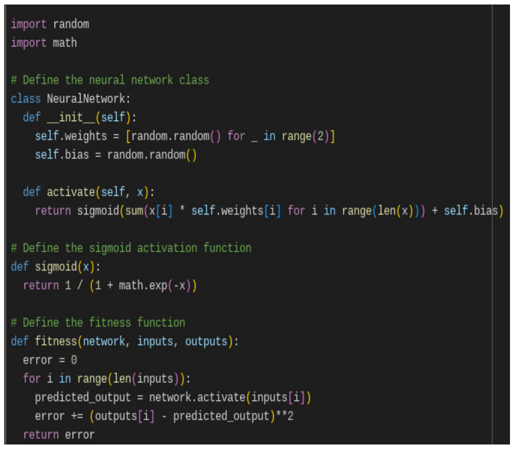
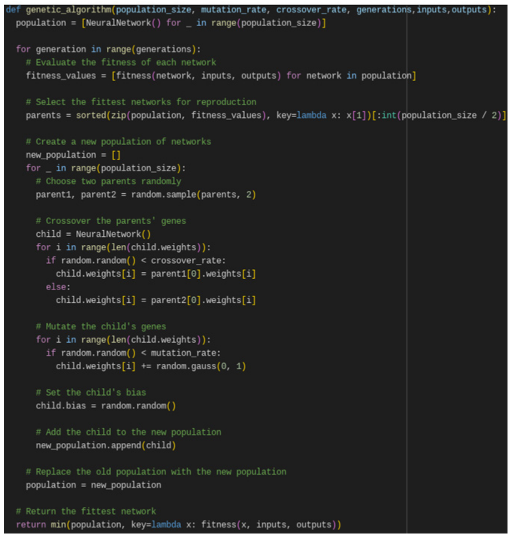
```
import random
def f(x, y):
  return x**2 + y**2

# Generate 1000 numbers from the function
inputs = [(round(random.random(),2), round(random.random(),2)) for _ in range(1000)]
outputs = [f(*i) for i in inputs]

X_train,X_test = inputs[:800], inputs[800:]
Y_train,Y_test = outputs[:800], outputs[800:]
```

Fig. 4: Code used to generate the Sample Data for the Simple Mathematical Function

Code	Description
	For training neural network model with backpropagation, we only instantiate one neural network and update the weight of the network while passing through the data and correcting the weights so that the network output be close to the real outputs.

```python
import random
import math


# Define the neural network class
class NeuralNetwork:
  def __init__(self):
    self.weights = [random.random() for _ in range(2)]
    self.bias = random.random()

  def activate(self, x):
    return sigmoid(sum(x[i] * self.weights[i] for i in range(len(x))) + self.bias)

  def train(self, inputs, outputs, learning_rate):
    for i in range(len(inputs)):
      predicted_output = self.activate(inputs[i])
      error = outputs[i] - predicted_output

      for j in range(len(self.weights)):
        self.weights[j] += learning_rate * error * inputs[i][j]

      self.bias += learning_rate * error


# Define the sigmoid activation function
def sigmoid(x):
  return 1 / (1 + math.exp(-x))
```

Figure 5: Code used to train the Simple Mathematical Function Artificial Neural Network with Backpropagation

Code	Description
	This code creates a class for Neural Networks with sigmoid activation function. The fitness function calculates the squared error for each observation in the training/testing dataset and returns the total error. By setting the Neural Networks as a class we can instantiate many objects from the class and train them independently. This network is quite simple with only two inputs and a bias term.
	The genetic algorithm function trains the neural network by instantiating the number of individual neural networks in the population Neural Network(). It then applies the fitness function to each instance and calculates the total error. Finally, it sorts the instances by fitness values and picks the first 50% (we can change this for different experiments). During the training, it crosses over the network weight based on the value of a random number and mutates randomly with a little gaussian noise. Most of the code was generated by google collab AI with user optimization for the specific purpose.

```python
import random
import math

# Define the neural network class
class NeuralNetwork:
  def __init__(self):
    self.weights = [random.random() for _ in range(2)]
    self.bias = random.random()

  def activate(self, x):
    return sigmoid(sum(x[i] * self.weights[i] for i in range(len(x))) + self.bias)

# Define the sigmoid activation function
def sigmoid(x):
  return 1 / (1 + math.exp(-x))

# Define the fitness function
def fitness(network, inputs, outputs):
  error = 0
  for i in range(len(inputs)):
    predicted_output = network.activate(inputs[i])
    error += (outputs[i] - predicted_output)**2
  return error
```

```python
def genetic_algorithm(population_size, mutation_rate, crossover_rate, generations,inputs,outputs):
  population = [NeuralNetwork() for _ in range(population_size)]

  for generation in range(generations):
    # Evaluate the fitness of each network
    fitness_values = [fitness(network, inputs, outputs) for network in population]

    # Select the fittest networks for reproduction
    parents = sorted(zip(population, fitness_values), key=lambda x: x[1])[:int(population_size / 2)]

    # Create a new population of networks
    new_population = []
    for _ in range(population_size):
      # Choose two parents randomly
      parent1, parent2 = random.sample(parents, 2)

      # Crossover the parents' genes
      child = NeuralNetwork()
      for i in range(len(child.weights)):
        if random.random() < crossover_rate:
          child.weights[i] = parent1[0].weights[i]
        else:
          child.weights[i] = parent2[0].weights[i]

      # Mutate the child's genes
      for i in range(len(child.weights)):
        if random.random() < mutation_rate:
          child.weights[i] += random.gauss(0, 1)

      # Set the child's bias
      child.bias = random.random()

      # Add the child to the new population
      new_population.append(child)

    # Replace the old population with the new population
    population = new_population

  # Return the fittest network
  return min(population, key=lambda x: fitness(x, inputs, outputs))
```

Fig. 6: Training Simple Mathematical Function Neural Network with Genetic Algorithms

We build a neural network to train with the diabetes dataset. The neural network has a hidden layer with 10 neurons and sigmoid activation functions and a final linear layer for the regression. We trained the neural networks with the Keras library for backpropagation and for training with the genetic algorithms, we implemented the same algorithm presented in Fig. 6.

Experiment with iris dataset

The third experiment was conducted for multi-class classification. The iris dataset was utilized to conduct this experiment. The iris dataset is a small dataset with 150 observations. There are 4 features in this dataset and one multi-class output. For more detail about this dataset please visit: https://archive.ics.uci.edu/dataset/53/iris.

RESULTS

We ran 10 experiments with different parameters for the Simple Mathematical Function, until a small Mean Square Error (MSE) was reached. The results of training with backpropagation is summarized in Table 1, while the results for the same network with genetic algorithms are presented in Table 2.

For the diabetes dataset, the results of the training of the neural network with backpropagation are shown in Table 3, while the results of training the same neural network with genetic algorithms are shown in Table 4.
Results for the Iris dataset documented in Tables 5 and 6.

DISCUSSION

On the two variable differentiable mathematical function, backpropagation performed significantly better than genetic algorithms in both error, and training time. For the Diabetes Dataset, we tested the model performance with different settings and even limiting the number of generations to 1000 the model performance was above 5000 in MSE test. For Diabetes Dataset experiments show that the genetic algorithm can reach convergence fast (Table 3 *vs* Table 4); however, more search is needed to fully evaluate the accuracy for real data. For the Iris dataset genetic algorithm outperformed backpropagation both from the accuracy and training time perspective.

Up to this point it has been assumed that genetic algorithms require more computational power than backpropagation and that the larger the size of the neural network, the worse this problem becomes. The general assumption is that genetic algorithm memory usage that for large networks might be considerable because it needs to instantiate a large population of similar networks in contrast to backpropagation that only needs one instance of neural network.

However, with the backpropagation we not only need to understand the model architecture, but we also need to understand the derivative across different network nodes. The larger the size of the network the more time it takes for the network to converge. Genetic algorithms, in the other hand, are

Table 1: Result of training simple mathematical function neural network with backpropagation

Experiment	*Learning Rate*	*Number of Epochs*	*Train MSE*	*Test MSE*	*Training Time (Seconds)*
1	0.1	10	0.058	.051	0.021
2	0.3	20	0.086	0.079	0.037
3	0.05	50	0.076	0.068	0.083
4	0.1	100	0.094	0.085	0.3
5	0.05	2000	0.1	0.093	3.4

Table 2: Result of training simple mathematical function neural network with genetic algorithms

Experiment	*Parameters (population size, mutation rate, crossover rate, generations)*	*Train MSE*	*Test MSE*	*Training Time (Seconds)*
1	(100,0.1,0.7,10)	0.133	0.126	1.27
2	(100,0.05,0.8,20)	0.13	0.122	2.38
3	(100,0.05,0.8,30)	0.1347	0.1262	4.89
4	(100,0.05,0.8,100)	0.1334	0.1264	12.8
5	(50,0.2,0.5,10)	0.1342	0.1258	0.67
6	(10,0.2,0.5,10)	0.1389	0.1285	0.13
7	(10,0.2,0.5,20)	0.14	0.1329	0.24
8	(10,0.5,0.5,5)	0.1555	0.1487	0.07
9	(10,0.01,0.5,5)	0.1449	0.137	0.067
10	(10,0.01,0.5,50)	0.1286	0.1198	0.61

Table 3: Result of training diabetes dataset neural network with back propagation

Experiment	*Learning Rate*	*Number of Epochs*	*Train MSE*	*Test MSE*	*Training Time (Seconds)*
1	0.05	20	16371	17271	10.72
2	0.05	100	6300	5823	49.83
3	0.1	50	31117	33597	41.31
4	0.01	200	6576	5505	103
5	0.001	300	2619	2751	202

Table 4: Result of training diabetes dataset neural network with genetic algorithms

Experiment	*Parameters (population size, mutation rate, crossover rate, generations)*	*Train MSE*	*Test MSE*	*Training Time (Seconds)*
1	(10,0.1,0.5,10)	28035	24972	0.34
2	(100,0.1,0.5,50)	11738	9825	2.08
3	(100,0.05,0.5,100)	21907	19189	3.83
4	(100,0.05,0.5,500)	45664	41781	21.46
5	(200,0.1,0.5,100)	6076	5361	9.07
6	(200,0.2,0.5,500)	6102	5301	202

Table 5: Result of training iris dataset neural network with backpropagation

Experiment	*Learning Rate*	*Number of Epochs*	*Train Accuracy*	*Test Accuracy*	*Training Time (Seconds)*
1	0.05	20	67	63	2.3
2	0.05	100	67	63	4.5
3	0.1	50	79	76	4.4
4	0.01	1000	94.1	93.3	41
5	0.02	5000	98.4	100	203

Table 6: Result of training iris dataset neural network with genetic algorithms

Experiment	*Parameters (population size, mutation rate, crossover rate, generations)*	*Train Accuracy*	*Test Accuracy*	*Training Time (Seconds)*
1	(10,0.1,0.5,10)	65	70	0.08
2	(100,0.1,0.5,50)	66	70	3
3	(100,0.05,0.5,100)	65	70	6
4	(100,0.05,0.5,500)	32	36	18
5	(100,0.1,0.5,1000)	97.5	96.6	37.7
6	(200,0.01,0.5,2000)	98.5	100	138

architecture independent and the iterative process is identical, independent of the size of the network.

CONCLUSION

The effects of optimization approaches on neural network training were investigated, in terms of model test accuracy and training time. The article compared genetic algorithms with the traditional backpropagation methodology. Genetic algorithms can be implemented for any model architecture. In terms of accuracy and training time, our experiments show that genetic algorithms can provide good convergence, even faster than backpropagation.

There is not enough information to conclude that genetic algorithms can outperform backpropagation or vice versa. The choice genetic algorithms vs backpropagation depends on the complexity of the problem. If the network is complex and obtaining derivation formula is not straightforward, we do think genetic algorithms are a proper alternative and further research would be needed to validate this approach.

REFERENCES

1. Cajal SR. Texture of the Nervous System of Man and the Vertebrates. Wien: Springer-Verlag; 1999.
2. McCulloch WS, Pitts W. A logical calculus of the ideas immanent

in nervous activity. Bull Math Biophys. 1943;5:115–33.

3. Hebb D. The Organization of Behavior: A Neuropsychological Theory. New ed. Psychology Press; 2002.
4. Rosenblatt F. Cornell Aeronautical Laboratory. Report no. VG-1196-G-8; 1962.
5. Rosenblatt F. Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms. Spartan Books; 1962.
6. Werbos P. The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting. New York: John Wiley & Sons; 1994.
7. Holland J. Adaptation in Natural and Artificial Systems. University of Michigan Press; 1975.
8. Yao X. Evolving artificial neural networks. Proc IEEE. 1999;87(9):1423–47.
9. Stanley KM, Miikkulainen R. Evolving neural networks through augmenting topologies. Evol Comput. 2002;10(2):99–127.
10. Whitley D, Starkweather T, Bogart C. Genetic algorithms and neural networks: optimizing connections and connectivity. Parallel Comput. 1990;14(3):347–61.