OPTIMISATION OF NEURAL NETWORKS USING GENETIC ALGORITHM AND SYNTHETIC GRADIENTS

A DISSERTATION

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE AWARD OF DEGREE
OF

MASTER OF TECHNOLOGY
IN
SOFTWARE ENGINEERING

Submitted by:

VIJAY PANDEY
(2K16/SWE/20)

Under the supervision of:

DR. RAJNI JINDAL
(HOD CSE)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Bawana Road , Delhi – 110042

JUNE 2018

CANDIDATE'S DECLARATION

I, VIJAY PANDEY, 2K16/SWE/20 a student of M.TECH (Software Engineering) declare that the project Dissertation titled "OPTIMISATION OF NEURAL NETWORKS USING GENETIC ALGORITHM AND SYNTHETIC GRADIENTS" which is submitted by me to Department of Computer Science and Engineering, Delhi Technological University, Delhi in partial fulfilment of the requirement for the award of the degree of Master of Technology, is original and not copied from any source without proper citation. This work has not previously formed the basis for the award of any Degree, Diploma, Fellowship or other similar title or recognition.

Place: DTU, Delhi VIJAY PANDEY

Date: (2K16/SWE/20)

CERTIFICATE

I, hereby certify that the Project titled "OPTIMISATION OF NEURAL NETWORKS USING GENETIC ALGORITHM AND SYNTHETIC GRADIENTS" submitted by VIJAY PANDEY, Roll number: 2K16/SWE/20, Department of Computer Science and Engineering, Delhi Technological University, Delhi in partial fulfilment of the requirement for the award of the degree of Master of Technology, is a record of project work carried out by the student under my supervision. To the best of my knowledge, this work has not been submitted in part or full for any Degree or Diploma to this University or elsewhere.

Place: Delhi

Dr. RAJNI JINDAL

(HOD CSE)

Date:

SUPERVISOR

ACKNOWLEDGEMENT

I am very thankful to **Dr. Rajni Jindal** (Head of the Department, Computer Science Eng. Dept.) and all the faculty members of the Computer Science Engineering Dept. of DTU. They all provided immense support and guidance for the completion of the project undertaken by me. It is with their supervision that this work came into existence.

I would also like to express my gratitude to the university for providing the laboratories, infrastructure, test facilities and environment which allowed me to work without any obstructions.

I would also like to appreciate the support provided by our lab assistants, seniors and peer group who aided me with all the knowledge they had regarding various topics.

VIJAY PANDEY

M.TECH (SWE)

2K16/SWE/20

ABSTRACT

For the last few years machine learning algorithms have become increasingly popular for solving most of the real life as well as complex problems. Also the demand for tools automating such algorithms has been on a rise. However the difficult part of these machine learning algorithms is to identify the best model for solving the problem and also to identify the hyper parameters that play the most crucial part in getting the most efficient solution to the problem. The methods in present state of art are by using trial and error to identify the model and the hyper parameter.

Neural network are the state of the art for solving most real world problems including the complex problems. However constructing a neural network for solving a problem requires a huge task of defining the hyper parameters using trial and error which is very time consuming and inefficient. Also, training a neural network requires forward and backward passes. Back propagation algorithm used for updating the weights by propagating the error variables to the subsequent layers result in interlocking of the initial layers till the latter layers have back propagated the error signal. This also results in an increased time complexity of the model.

Genetic algorithms are inspired by the evolutionary process of organisms and are very effective in reducing the time complexity of computational algorithms. GA can be used for automating the task of feature selection of desired neural network in minimal time. Also, synthetic gradients can be used to do away with the problem of interlocking of layers caused by back propagation. Synthetic gradients decouple the layers of NN by introducing the model of future computation of error of the network graph. In this way errors can be computed independently without waiting for the error signals, thus reducing the time complexity.

Both the techniques are used to train and test MNIST and CIFAR-10 database. A Recurrent Neural Network model (RNN) is used as the model which can be further extended to any type of neural network including CNNs, LSTM and others.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ML	Machine Learning
MLP	Multi Layer Perceptron
CNN	Convolutional Neural Network
NN	Neural Network
GA	Genetic Algorithm
DNI	Decoupled Neural Interfaces
NIST	National Institute of Standards and Technology
MNIST	Modified National Institute of Standards and Technology
CIFAR	Canadian Institute For Advanced Research

CHAPTER 1

INTRODUCTION

1.1 OVERVIEW

In 2018, it's moderately simple to train neural networks, but it's still difficult to guess which network architectures and other hyper-parameters we should use – for example, number of neurons, how many layers, and which activation functions to use. In coming future, of course, neural networks will automatically learn how to construct themselves, without human intervention. Till then, the pace of developing application-optimized neural nets will remain limited by the time and expertise required to choose and refine hyper-parameters. This project is designed to help solve this problem, by constantly returning good hyper-parameters for particular datasets and classification problems. The code supports hyper-parameter discovery for MLPs (ie. fully connected networks) which can be further extended to other models as well.

Machine learning is the subfield of computer science that, according to Arthur Samuel in 1959, gives "computers the ability to learn without being explicitly programmed". Evolved from the study of pattern recognition and computational learning theory in artificial intelligence, machine learning explores the study and construction of algorithms that can learn from and make predictions on data – such algorithms overcome following strictly static program instructions by making data-driven predictions or decisions.

Between 2011 and 2015, the number of self-reported data scientists more than doubled. At the same time, machine learning has returned to the forefront of academia, business, and government as data scientists discover new applications for algorithms that automatically learn and create actionable insights from data. Owing to this growth, there has been a big demand for on-the-shelf tools that make machine learning more accessible, scalable, and flexible such that they can be applied across a wide variety of domains by non-experts. Unfortunately, the effective application of many machine learning tools typically requires expert knowledge of the tool and the problem domain, knowledge of the assumptions involved in the analysis, and/or the use of exhaustive brute force techniques. Inexperienced data scientists can easily spend most of their time exploring myriad pipeline configurations before settling on the best one.

In recent years, we have witnessed the development of intelligent systems in the field of evolutionary computation that consistently surprise us with their capabilities. In computer science, evolutionary computation is a family of algorithms for global optimization inspired by biological evolution, and the subfield of artificial intelligence and soft computing studying these algorithms. In technical terms, they are a family of population-based trial and error problem solvers with a metaheuristic or stochastic optimization character. Evolutionary computation techniques can produce highly optimized solutions in a wide range of problem settings, making them popular in computer science. Many variants and extensions exist, suited to more specific families of problems and data structures.

Before fitting a model of the data, the practitioner must prepare the data for modeling by performing an initial exploratory analysis (e.g., looking for missing or mislabeled data) and either correct or remove the missing records (i.e., data cleaning). Next, the practitioner may transform the data in some way to make it more suitable for modeling, e.g., by normalizing the features (i.e., feature preprocessing), removing features that are not useful for modeling (i.e., feature selection), and/or creating new features from the existing data (i.e., feature construction). Afterward, the practitioner must select a machine learning model to fit to the data (i.e., model selection) and choose the model parameters that allow the model to make the most accurate classification from the data (i.e., parameter optimization). Lastly, the practitioner must validate the model in some way to ensure that the model's predictions generalize to data sets that it was not fitted on (i.e., model validation).

1.2 PROBLEM STATEMENT

Building a good deep learning network includes a substantial amount of art to go with sound science. One way of finding the right hyper-parameters is using brute force trial and error: Try every possibility of sensible parameters, send them to your algorithm, go about your everyday routine, and come back when you got an answer.

A normally used neural architecture comprises largely of several convolutions, pooling, and completely connected layers. Many recent studies emphasis on developing a unique neural architecture that attains higher classification accuracy, like ResNet, GoogleNet and DensNet. Despite their success, designing neural architectures is still a hard task since many design constraints exist, like the depth of a net, the kind and parameters of every layer, and the connectivity of the layers. State-of-the-art CNN architectures have become deep and complex, which put forward that a significant number of design constraints should be tuned to understand the best performance for a given dataset. Therefore, trial-and-error or adept knowledge is required when users build suitable architectures for their target datasets. In lieu of this situation, automatic design approaches for neural architectures are extremely beneficial. Neural network architecture design can be seen as the model selection problem in terms of machine learning. The straight-forward method is to deal with construction design as a hyper-parameter optimization problem, improving hyper-parameters, such as the total number of layers and neurons, using black-box optimization techniques.

Evolutionary computation has been generally connected to outlining neural network structures. There are two kinds of encoding plans for network portrayal: direct and indirect coding. Direct coding speaks to the number and availability of neurons directly as the genotype, while indirect coding speaks to a generation run for network models. Albeit all customary methodologies advance the number and availability of low-level neurons, present day neural network structures for profound learning have numerous units and different kinds of units, e.g., convolution, pooling, and standardization. Upgrading such a significant number of parameters in a sensible measure of computational time might be troublesome. In this manner, the utilization of profoundly effective modules as a base unit is promising.

Additionally if you think about any layer or module in a neural network, it must be updated once all the other modules of the network have been executed, and gradients have been back-propagated to it. After Layer 1 has handled the information, it must be refreshed after the output activations have been propagated through whatever remains of the network, created a loss, and the error gradients back-propagated through each layer until Layer 1 has come. This grouping of tasks implies that Layer 1 needs to sit tight for the forwards and backwards calculation of Layer 2 and Layer 3 preceding it can update. Layer 1 is locked, coupled, to whatever remains of the network.

This is definitely a problem. Clearly for a simple feed-forward network we don’t require to fear about this problem. But consider a composite system of numerous networks, acting in multiple environments at asynchronous and irregular timescales. Or a large distributed network ranging over multiple devices. Sometimes needing all modules in a system to wait for all other modules to execute and back-propagate gradients is excessively time consuming or sometimes intractable.

1.3 OBJECTIVE

Here, we try to advance upon the brute force algorithm by applying a genetic algorithm to develop a network with the objective of achieving ideal hyper parameters in a portion the time of a brute force search. We also use synthetic gradients instead of the customary back-propagation algorithm to do away with the problem of update locking hence improving upon time complexity in the designing process. We use the synthetic gradients method to help decouple the neural interfaces so that they become independent of each other.

1.4 ORGANISATION OF THESIS

The following chapters further elucidate the optimization algorithm and outcomes. Chapter 2 clarifies the necessary research work done in the field of neural networks and evolutionary computation methods used in the optimizer. Chapter 3 gives a short description of the terminologies used and a few basic concepts in the related area. Chapter 4 discusses the overall methodology associated with genetic algorithm optimizer. Chapter 5 explains how the endless search space is abridged to allow for a substantial speed up for the optimization process. Also results and analysis have been explained. Chapter 6 ends with concluding notes and future research efforts.

CHAPTER 2

LITERATURE SURVEY

Neural networks and the genetic algorithm are equally powerful tools modeled upon natural phenomena. Neural networks are modeled after the brain; which is extremely parallel, and gives many advantages while solving pattern recognition and classification tasks. The GA is built on the model of evolution and survival of the fittest and had been used to solve a large number of optimization problems. Neural networks give many advantages in a range of applications, but are unsuccessful if they are not correctly designed. There are many varieties when designing a neural network (NN) but a bad selection of any one parameter can leave the NN useless. There have been many attempts to produce formulas or directions for designing the organization of a NN. In[6], a formula was developed to generate a range of the desired number of hidden neurons, h, based on the number of inputs and outputs of a NN:

$$h = \sqrt{n + m} + (1 \sim 10)$$

where n is the number of inputs, m is the number of outputs (square root result is rounded up). The formula yields a range, that is not particularly beneficial because several systems need to be trained and evaluated to find the best one. Additional problem with this method is that it assumes the number of inputs and outputs is directly correlated with the complexity of the problem, which is not always the case. Furthermore, this formula only applies to a network with one hidden layer. Since there are no well-defined procedures for selecting the parameters of a NN for a given application, finding the best parameters can be a case of trial and error. There are many papers in which the authors arbitrarily choose the number of hidden layer neurons, activation function, and number of hidden layers. In[10], networks were trained with 3 to 12 hidden neurons and found that 9 was optimal for that specific problem. The GA had to be run 10 times, one for each of the network architectures. Since selecting NN parameters is more of an art than a science, it is an ideal problem for the GA. The GA has been used in numerous different ways to select the architecture, prune, and train neural networks. In [4], a simple encoding scheme was used to optimize a multi-layer NN. The encoding scheme consisted of the number of neurons per layer, which is a key parameter of a neural network. Having too few neurons does not allow the neural network to reach an acceptably low error, while having too many neurons limits the NN's ability to generalize. Another important design consideration is deciding how many connections should exist between network layers. In [5], a genetic algorithm was used to determine the ideal amount of connectivity in a feed-forward network. The three choices were 30%, 70%, or 100% (fully-connected). In general, it is beneficial to minimize the size of a NN to decrease learning time and allow for better generalization. A common process known as pruning is applied to neural networks after they have already been trained. Pruning a NN involves removing any unnecessary weighted synapses. A GA was used to prune a trained network. The genome consisted of one bit for each of the synapses in the network, with a '1' represented keeping the synapse, while a '0' represented removing the synapse. Each individual in the population represented a version of the original trained network with some of the synapses pruned (the ones with a gene of '0'). The GA was performed to find a pruned version of the trained network that had acceptable error. Even though pruning reduces the size of a network, it requires a previously trained network. The algorithm developed in this research optimizes for size and error at the same time, finding a solution with minimum error and minimum number of neurons.

Another critical design decision, which is application-specific, is the selection of the activation function. Depending on the problem at hand, the selection of the correct activation function allows for faster learning and potentially a more accurate NN. In [8], a GA was used to determine which of several activation functions (linear, logsig, and tansig) were ideal for a breast cancer diagnosis application.

Neural networks with feedback loops have also been enhanced with GA generated primary weights. A GA was used to optimize the initial weights of an Elman Recurrent Network (ERN)so as to produce short-term load forecasting model for a power system. ERNs have an input layer, middle layer, continue layer, and output layer, as shown in Figure 2.1.

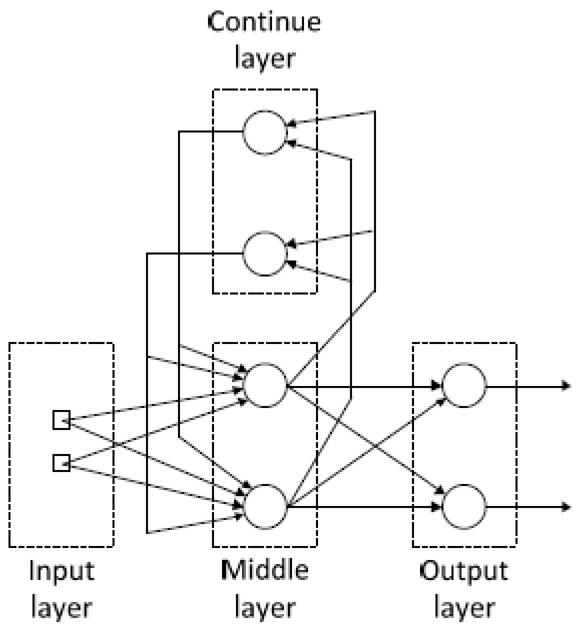


Fig2.1-The Structure of an Elman Recurrent Network

Genetic algorithms have also been used in the training process of neural networks, as an alternative to the back-propagation algorithm. In [10] and [12], genes represented encoded weight values, with one gene for each synapse in the neural network. It is shown in [13] that training a network using only the back-propagation algorithm takes more CPU cycles than training using only GA, but in the long run back-propagation will reach a more precise solution. The Improved Genetic Algorithm (IGA) was used to train a NN and shown to be superior to using a simple genetic algorithm to find initial values of a back propagation neural network. Each weight was encoded using a real number instead of a binary number, which avoided lack of accuracy inherent in binary encoding. Crossover was only performed on a random number of genes instead of all of them, and mutation was performed on a random digit within a weight's real number. Since the genes weren't binary, the mutation performed a "reverse significance of 9" operation (for example 3 mutates to 6, 4 mutates to 5, and so on). The XOR problem was studied, and the IGA was shown to be both faster and produce lower error.

Historically, machine learning automation research (autoML for short) has primarily focused on optimizing subsets of the pipeline[3]. For example, grid search is the most commonly-used form of hyperparameter optimization that applies brute force search to explore a broad range of model parameters in order to discover the parameter set that allows for the best model fit. Recent research has shown that randomly evaluating parameter sets within the grid search often discovers the ideal parameter set more efficiently than exhaustive search, which shows promise for intelligent search in the hyper-parameter space[4]. Bayesian optimization of model hyperparameters, in particular, has been effective in this realm and has even outperformed manual hyperparameter tuning by expert practitioners. Another focus of autoML research has been feature construction. One recent example of automated feature construction is the “Data Science Machine,” which automatically constructs features from relational databases via deep feature synthesis. In their work, Kanter et al. demonstrated the crucial role of automated feature construction in machine learning pipelines by entering their Data Science Machine in three machine learning competitions and achieving expert-level performance in all of them.

In 2015, Zutty et al.[6] similarly demonstrated an autoML system using genetic programming (GP) to optimize machine learning pipelines, and found that GP is capable of designing better pipelines than humans for one supervised classification task. As such, GP shows considerable promise in the autoML domain. All of these findings point to one take-away message: Intelligent systems are capable of automatically designing portions of machine learning pipelines, which can make machine learning more accessible and save practitioners considerable amounts of time by automating one of the most laborious parts of machine learning.

Haruna et al.[12] demonstrated how hybridization of two or more of these techniques eliminates such constraints and leads to a better solution. As a result of hybridization, many efficient intelligent systems are currently being designed. Recent studies that hybridized CI techniques in the search for optimal or near optimal solutions include, but are not limited to: genetic algorithm (GA), particle swarm optimization and ant colony optimization hybridization; fuzzy logic and expert system integration in [13]. The hybridization of GA and particle swarm optimization is considered the most reliable and promising CI techniques. Recently, NNs have proven to be a powerful and appropriate practical tool for modeling highly complex and nonlinear systems. The GA and NNs are the two CI techniques presently receiving attention from computer scientists and engineers. This attention is attributed to recent advancements in understanding the nature and dynamic behavior of these techniques. Furthermore, it is realized that hybridization of these techniques can be applied to solve complex and challenging problems. They are also viewed as sophisticated tools for machine learning. The vast majority of literature applying NNs was found to heavily rely on the back-propagation gradient method algorithms developed by [14] and popularized in the artificial intelligence research community. GA is evolutionary algorithm that could be applied (1) for the selection of feature subsets as input variables for back-propagation NNs, (2) to simplify the topology of back-propagation NNs and (3) to minimize the time taken for learning [13]. Some major limitations attributed to NNs and GA are explained as follows. The NNs are highly sensitive to parameters [12] which can have a great influence on the NNs performance. Optimized NNs are mostly determined by labor intensive trial and error techniques which include destructive and constructive NN design [11].These techniques only search for a limited class of models and a significant amount of computational time is, thus, required. NNs are highly liable to over-fitting and different types of NN which are trained and tested on the same dataset can yield different results. These irregularities are responsible for undermining the robustness of the NN. GA performance is affected by the following: population size, parent selection, crossover rate, mutation rate, and the number of generations [15].The selection of suitable GA parameter values is through cumbersome trial and error which takes a long time [15] since there is no specific systematic framework for choosing the optimal values of these parameters. Similar to the selection of GA parameter values, the design of an NN is specific to the problem domain. The most valuable way to determine the initial GA parameters is to refer to the literature with a description of a similar problem and to adopt the parameter values of that problem [14].

In summary, the papers mentioned above studied genetic algorithms that were lacking in several ways:

1. They do not allow flexibility of the number of hidden layers and neurons.

2. They do not optimize for size.

3. They have very large genomes and therefore search spaces.

The GA developed in this thesis addresses all of these issues. It searches through networks with one through four hidden layers, with anywhere from one to eight neurons in each. Also, the developed algorithm optimizes for error and size concurrently.

CHAPTER 3

BACKGROUND INFORMATION

This chapter provides a basic explanation about how neural networks, backpropagation and genetic algorithms work.

3.1. Neural Network Basics

Neural networks are easier to understand if they are broken down into their core components. This section explains the basics of neural networks. First, the nature of the neuron is explored (the elementary unit of a neural network). Next, the different ways in which neurons can be connected are shown. Finally, neural network training (the way in which a neural network learns) is examined.

3.1.1. The Neuron

Neural networks are made from as few as one to as many as hundreds of elementary units called neurons. As shown in figure 3.1, each neuron is made up of the following: inputs, synaptic weights, a bias, a summing junction, a local induced field, an activation function, and a single output. Each of the following need to be examined in order to understand how a neuron produces an output from its arbitrary inputs:

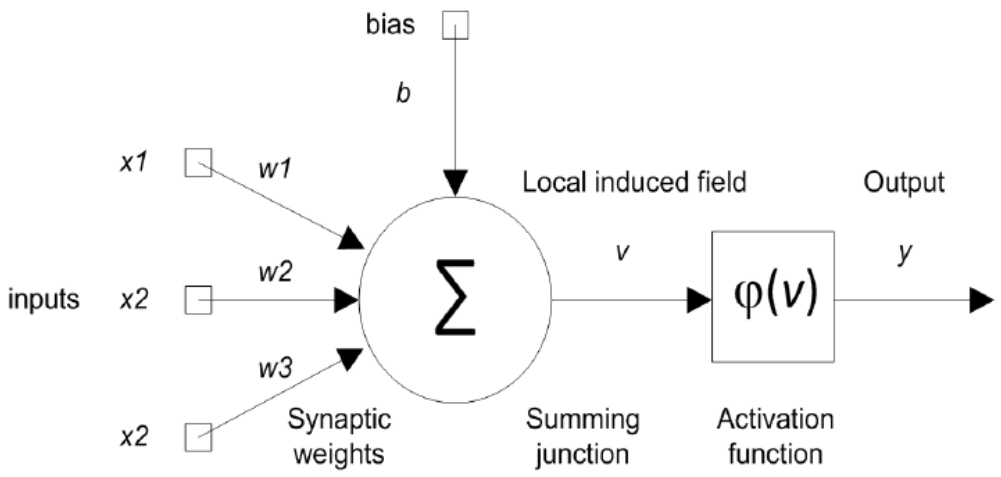


Fig 3.1 – The components of neuron

- **Inputs**

A neuron can have an arbitrary number of inputs (at least one). In figure 2.1, the three inputs are represented as *x1*, *x2*, and *x3*. Each neuron input can take on a positive,zero, or negative value. Since there is little restriction on the input values, neurons can be used in a broad range of applications. The outputs of several neurons can be connected to the inputs of another neuron, allowing the formation of multi-neuron networks.

- **Synaptic weights(s):**

Each neuron input has a corresponding synaptic weight (or simply weight) which can also take on any positive, zero, or negative value. In figure 3.1, the three synaptic weights are *w1*, *w2*, and *w3*. The synaptic weights are used to scale the inputs as they enter the summing junction.

- **Bias, Summing Junction, and Induced Local Field:**

The summing junction has at least two inputs (one for the bias and one for each of the neuron inputs). The bias (commonly represented as *b*) can be thought of as the weight for an input of unity. The summing junction therefore produces an output equal to the sum of the bias and all of the weighted inputs. The summing junction's output is called the induced local field, *v*, and is defined in equation 3.1 for the general case of neuron *k*.

$$\boldsymbol{V_k = b_k + \sum_{j=1}^{m_k} w_{kj} x_{kj}} \qquad \text{…eq.3.1}$$

Where

vk = induced local field of neuron *k*	*bk* = bias of neuron *k*
mk = number of inputs of neuron *k*	*wkj* = weight of synapse *j* of neuron *k*
xkj = input *j* of neuron *k*	

- **Activation Function and Output:**

As shown in figure 2.1, *v* is the input of the activation function φ(*v*). There are several different kinds of activation functions, each of which is beneficial in different situations. The unity between all activation functions is that they produce the neuron output y, which is restricted to values with a maximum value of 1 and a minimum value of either 0 or -1 (depending on the application). There are three basic activation functions that range from 0 to 1: threshold, piecewise-linear, and sigmoid. Another commonly used function, known as the hyperbolic tangent, is a modified version of the sigmoid having a range of -1 to 1. Equations 3.2 – 3.5 explicitly define them.

1. Threshold function: $\varphi(v) = \begin{cases} 1 \; if \; v \geq 0 \\ 0 \; if \; v < 0 \end{cases}$ …eq.3.2

2. Piecewise-linear function: $\varphi(v) = \begin{cases} 1 \; if \; v \geq 1/2 \\ v \; \; if \; 1/2 > v > -1/2 \\ 0 \; if \; v \leq -1/2 \end{cases}$ ...eq.3.3

3. Sigmoid function: $\varphi(v) = \frac{1}{1+\exp(-av)}$ …eq.3.4

4. Tanh function: $\varphi(v) = \tanh(av)$ …eq.3.5

3.1.2 The Back-Propagation Algorithm:

The back-propagation algorithm is used to change the synaptic weights throughout a neural network so as to minimize error. It is an iterative process, which changes the network one training examle at a time. During each iteration the error signal journeys backwards through the network, starting at the output neurons and finishing at the input synapses. The derivation of the back-propagation algorithm is not shown. Instead, a few important equations are discussed so that the effects of the learning rate and activation function can be shown.

The correction for a weight is defined as

$$\Delta w_{ji} = \eta \delta_j y_i \quad \text{...eq.3.6}$$

where η is the learning rate, Δw_{ji} is the change in weight connecting neuron *i* in layer L to neuron *j* in layer L+1, δ_j is the local gradient, and y_i is the output of neuron *i*. The

learning rate affects how much a weight will change based on the error and can be chosen to be any real number. The local gradient is the error signal that travels backwards through the network and is based on activation function, as well as whether the neuron in question is an output and non-output neuron. Only the correction functions for sigmoid and tanh activation functions are shown. For output neurons using the sigmoid activation function

$$\delta_j = a\big(d - O_j\big)O_j(1 - O_j) \quad \text{…eq.3.7}$$

where a is the sigmoid parameter defined in equation, dj is desired output, and O_j is the actual output. The correction function for output neurons using the sigmoid activation function is therefore

$$\Delta w_{ji} = a\eta\big(d - O_j\big)O_j\big(1 - O_j\big)y_i \quad \text{…eq. 3.8}$$

For non-output neurons using the sigmoid activation function

$$\delta_j = ay_j\big(1 - y_j\big)\sum_k \delta_k w_{k,j} \qquad \text{…eq.3.9}$$

where y_j is the output of neuron j, δ_k is the local gradient of neuron k in the next layer, and $w_{k,j}$ is the weight connecting neuron j with each of the neurons in the next layer. The correction function for non-output neurons using the sigmoid activation function is
therefore

$$\Delta w_{ji} = a\eta y_{j(1-y_j)} y_i \sum_k \delta_k w_{k,j} \qquad \text{…eq. 3.10}$$

3.1.3 The Network Architecture:

As mentioned in section 3.1.1, the output of a neuron can be connected to the input of another neuron. In fact, one neuron's output can be connected to any number of other neurons' inputs, allowing numerous possible ways of combining neurons to form a neural network. Neural networks are commonly organized in a layered fashion, in which neurons are organized in the form of layers. There are three kinds of layers: input layer, hidden layers, and output layer. The input layer is made up of the input nodes of the neural network. The output layer consists of neurons which produce the outputs of the network. All layers which do not produce outputs, but instead produce intermediate signals used as inputs to other neurons, are considered hidden layers. Figure 2.3 shows a simple neural network, which consists of an input layer and an output layer (circles represent neurons and arrows represent synapses). In this network, three neurons process the three inputs to produce three outputs.

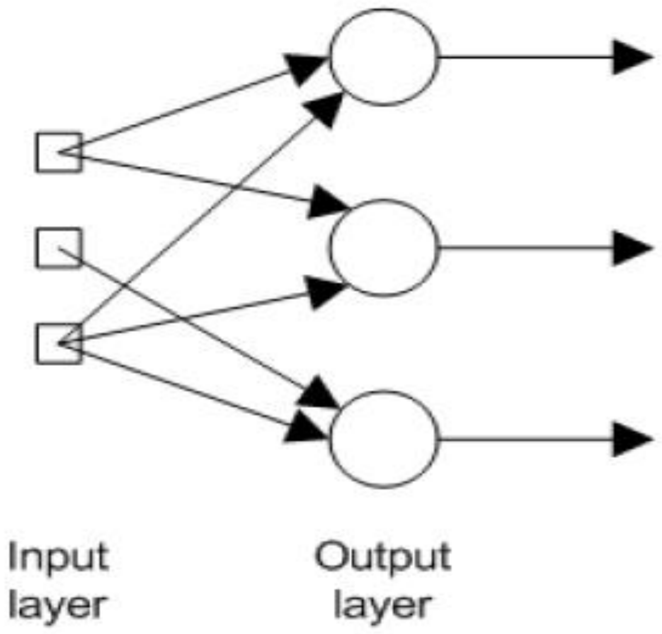


Fig 3.2- A Simple Feedforward neural network

Some complex tasks require architectures that contain multiple hidden layers.

3.1.4 Convolution Neural Networks:

Neural networks are limited with images due to the fact they required objects in a similar portion of the image to be classified. The image has to be generalized before it is passed to the fully connected layers. Most image processing is performed using convolutional kernels. Yann Lecun came up with the idea of convolutional layers to use a way of using feature extraction before the fully connected layers.

Each convolutional layers can have many filters per layer. The maximum amount of filter is M. Each convolutional kernel is k, and kernel can differ dramatically in size as well as the stride length. The step length controls how far each convolutional kernel achieves the convolution operation which changes the dimensions of the output. The _ denotes the 2D convolutional operator. The function u is the activation function which can vary dramatically in each application. The output, as the nodal layers before is yj .

$$y_j = u\left(b_j + \sum_{i=1}^{m} k_{ij} * x_i\right) \quad \text{...eq. 3.11}$$

Convolutional layers prove to be very beneficial in many applications. Feature extraction allow for networks to grow in depth without over-fitting to the data. The features extracted from convolutional layers are often referred to as feature maps. These feature maps allow the convolutional neural network to be more generalized and reducing the amount of diversity required for the training process.

3.1.5 Pooling

Convolutional layers are known for their feature extraction and feature detection. Although these layers do perform well they require significant computation. When convolutional layers do not detect any of the desired features, the output becomes mostly zeros. Pooling allows for selected values to be passed to the next layer while leaving the unnecessary values behind. There are a few pooling methods such max pooling, average pooling, and minimum pooling. Max pooling takes the largest value from the pooling area, average takes the average from the area and minimum takes the minimum. Most modern applications use max pooling. Max pooling passes the largest value in the given area removing most of the sparse data thus increasing generalization and computation efficiency.

3.1.6 Synthetic Gradients:

The synthetic gradient model takes in the activations from a module and produces what it predicts will be the error gradients - the gradient of the loss of the network with respect to the activations. In a simple feed-forward network example, if we have a synthetic gradient model we can do the following:

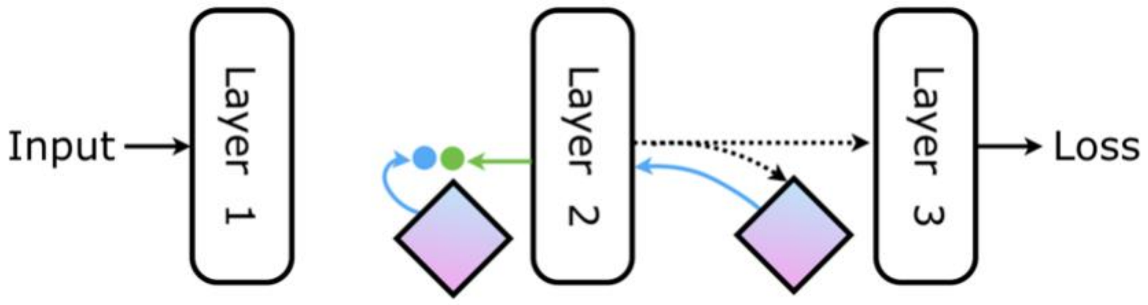


Fig. 3.3- Working of synthetic gradient model

use the synthetic gradients (blue) to update Layer 1 before the rest of the network has even been executed. The synthetic gradient model itself is trained to regress target gradients - these target gradients could be the true gradients back-propagated from the loss or other synthetic gradients which have been back-propagated from a further downstream synthetic gradient model.

Quite surprisingly, the synthetic gradient models can be very simple. For feed-forward nets, we actually found out that even a single linear layer works well as a synthetic gradient model. Consequently it is both very easy to train and so produces synthetic gradients rapidly.

It's important to recognise that DNI doesn't magically allow networks to train without true gradient information. The true gradient information does percolate backwards through the network, but just slower and over many training iterations, through the losses of the synthetic gradient models. The synthetic gradient models approximate and smooth over the absence of true gradients.

3.2 Evolutionary Computation:

Evolutionary computation is the method of applying the concept of evolution to optimize a solution or solve problems. Evolutionary computation is a method of machine learning that is proven to be effective to many applications and also highly scalable. If a problem solution can be reduced to a series of values then the problem is a potential candidate for evolutionary computation.

3.2.1 Chromosomes

Every potential solution to a problem is known to as a chromosome. How each chromosome is defined dramatically affects how well the evolutionary computation performs. The chromosome definition could dramatically reduce the search space either removing redundant or incompatible solutions. Two problems are widely used as examples for evolutionary computation are the traveling salesman and the knapsack problem. The traveling salesman problem consist of a salesman who has to go to many cities and end back home. While there is many possible solutions to the problem the salesman wants to know what path will get him home the fastest. The solution of the problem would be a list of cities in the order in which he would visit. Therefore, to improve the speed of optimization, the number of potential solutions needs to be minimized. An example of this minimization for the traveling salesman problem can be, not allowing solutions with repeats of the same city to be generated. The knapsack problem is another problem which is fairly common. The problem is defined as, if the knapsack has a limited amount of space, fill it with the greatest value. There will be many items with different sizes and values. The task is to fit the most items with the highest value in the bag without overfilling it. The chromosome would then be the items which go into the knapsack after each item is assigned a unique identifier.

3.2.2 Crossover Techniques

In every evolutionary computation techniques there is a task called crossover. This is the combination of chromosomes to produce new chromosomes. These new chromosomes are referred to as children and the chromosomes who generated the child is referred to as a parent. Most crossover techniques have two parents producing one or two children, but this varies dramatically on each application. A few popular techniques are n-point crossover, arithmetic crossover, and uniform crossover.

The first discussed technique is the n-point crossover. The value n can range to many values, but the most common of which is a single point crossover. There will be two parents to produce two children. First it picks a point at random then each parent is broken in half from that point. Once the parents are split each half from the parents are swapped with the other half from the other parent. This produces two unique children. The value n for the n-point crossover points signifies the number of break points for each chromosome. The simplest variation of this crossover is a single point crossover. A single point crossover is when there is only a single point of separation from each parent. Figure 3.3 shows a visual depiction of a single point crossover.

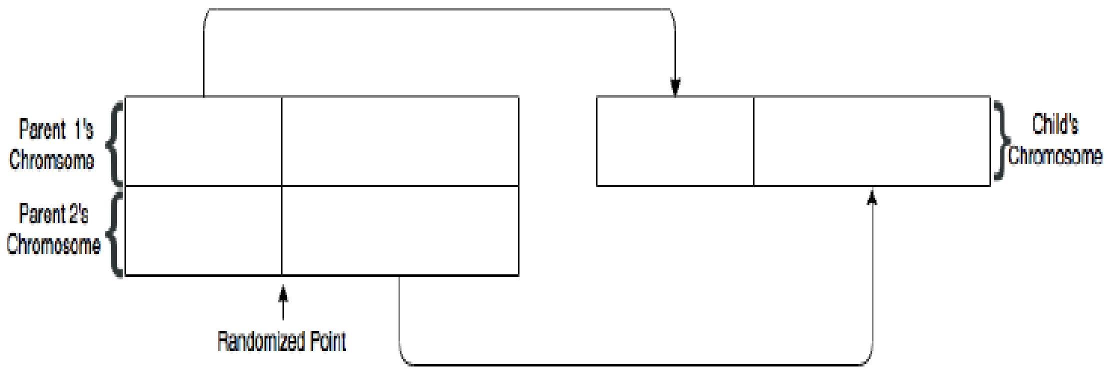


Fig 3.4 Single Point Crossover Technique

Another popular one is arithmetic crossover. The most common of which is taking two parents to produce a single child. This crossover adds the two parents together then dividing by two. The parents create the child by averaging themselves together. This method prompts much faster convergence, thus lacks exploitation.

The uniform crossover technique takes two parents and generates two children like the n point crossover. However instead of declaring a certain number of break points half of each parent will be picked at random. Yet, any part that does not make it to the first child is the part of the second child. This allows for exactly half of both parents to exist with each child.

3.2.3 Exploration and Exploitation

Exploration and exploitation is the biggest problem any search optimization technique run into. A specifically scenario is with infinite search spaces. This search space could then only contain a singular global minimum. This infinite search space could also have many local minimum so it becomes difficult to determine when the algorithm has found the global minimum. This is where the concept of exploration vs exploitation is important. Exploration is the process of trying samples scattered through a search space. Exploitation is the process of trying samples near well performing solutions. The best method is to explore the search space and once enough information is gathered the algorithm should start to exploit. When preforming the crossover techniques this usually promotes more exploration than exploitation. Another method exists to control the concept of exploration vs. exploitation The common technique for exploring the search space by randomizing the children created from crossover. This method is referred to as mutation. The amount of mutation can be controlled by limiting how often a gene mutates. This probability of mutation if refereed to as a mutation rate.

In most applications it is best to start with a higher mutation rate and lower it slowly after each generation. This is the primary technique to ensure adequate exploration vs. exploitation.

3.2.4 Selection

The selection process is another key aspect of of evolutionary computation. This step chooses which members of the populace to keep with each generation. Common selection processes are

elitist selection, roulette wheel selection, and tourney selection. The selection process is important since it is an additional method in controlling exploration vs. exploitation.
Elitist selection is the most common and the simplest selection technique. The method is to only keep the strongest chromosomes. This makes sure that no valuable chromosomes are lost. However, this highly promotes exploration over exploitation. This could to very fast convergence, but to a poor solution.
Roulette wheel selection creates a probability distribution when selecting which chromosomes stay in the populace. The chromosomes with higher scores are weighted so they are selected more often. This is a compromise between exploration vs. exploitation.
There are some instances where it is hard to quantitatively evaluate how a chromosome is performing. An example of this would be if evolutionary computation is being used to optimize a robot to play soccer. A robot by itself is hard to objectively assign a performance value. Instead of comparing a single value each chromosome will compete against each other. The winners are chosen by hosting a tournament against the other chromosomes with each generation. The top winners of the tournament are then kept in the populace.

The idea behind GA´s is to extract optimization strategies nature uses successfully - known as Darwinian Evolution - and transform them for application in mathematical optimization theory to find the global optimum in a defined phase space.

One could imagine a population of individual explorers sent into the optimization *phase-space*. Each explorer is defined by its genes, what means, its position inside the phase-space is coded in his genes. Every explorer has the duty to find a value of the quality of his position in the phase space. (Consider the phase-space being a number of variables in some technological process, the value of quality of any position in the phase space - in other words: any set of the variables - can be expressed by the yield of the desired chemical product.) Then the struggle of "life" begins. The three fundamental principles are

1. Selection
2. Mating/Crossover
3. Mutation

Only explorers (= *genes*) sitting on the best places will reproduce and *create a new population*. This is performed in the second step (*Mating/Crossover*). The "hope" behind this part of the algorithm is, that "good" sections of two parents will be recombined to yet better fitting children. In fact, many of the created children will not be successful (as in biological evolution), but a few children will indeed fulfill this hope. These "good" sections are named in some publications as building blocks.

Now there seems a problem. Repeating these steps, no *new* area will be explored. The two former steps would only feat the already known regions in the phase space, which could lead to untimely convergence of the algorithm with the result of missing the global optimum by exploiting some local optimum. The third step - the Mutation confirms the necessary accidental effects. One can visualize the new population being varied up a little bit to bring some new information into this set of genes. Off course this has to happen in a well-balanced way!

Whereas in biolog*y* a gene is labelled as a macro-molecule with four different bases to code the genetic information, a gene in genetic algorithms is usually defined as a bitstring (a sequence of b 1´s and 0´s).

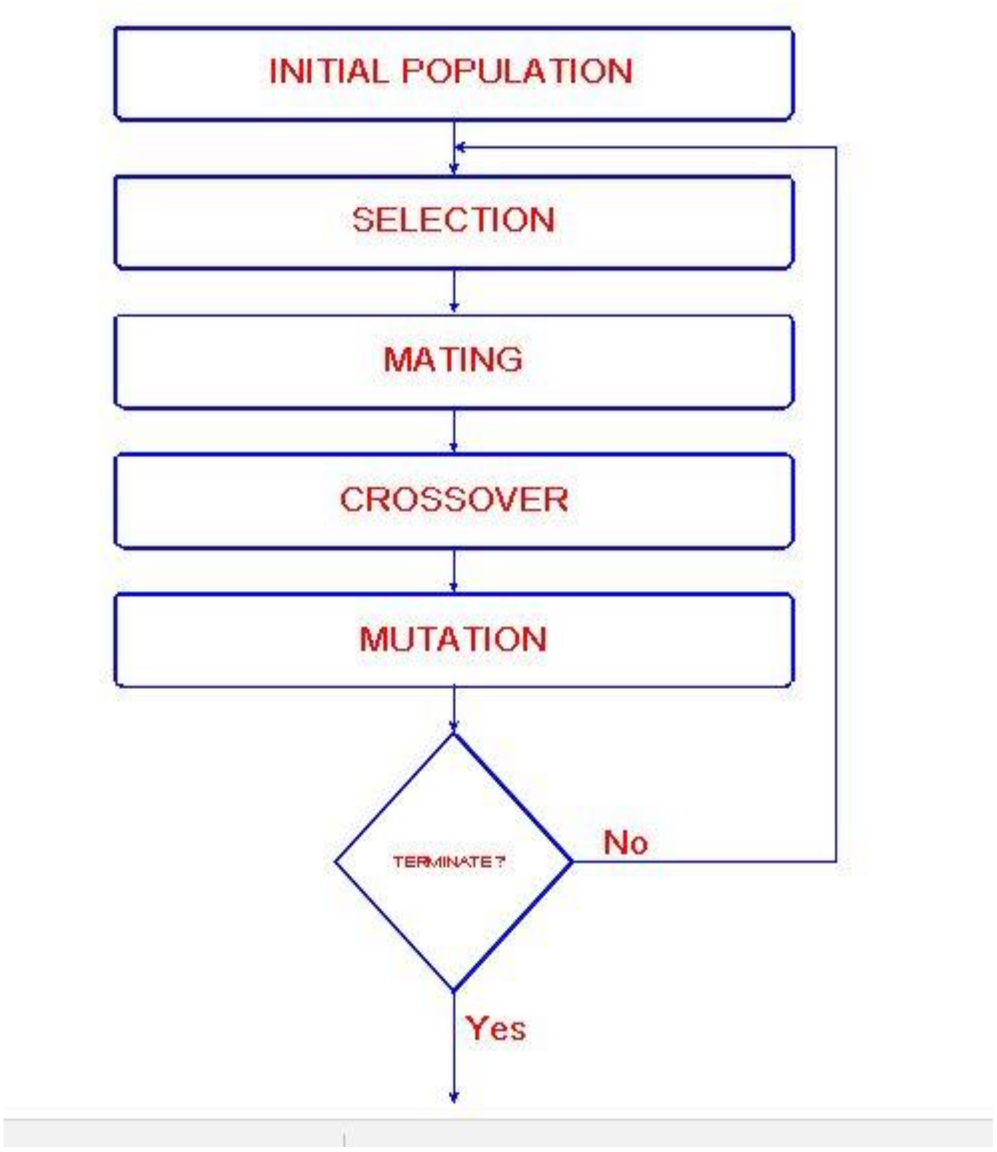


Fig3.5- The process of genetic algorithm

CHAPTER 4

SIMULATION DETAILS

Our simulation for automatic parameter tuning of neural network was based on its performance on a classification problem on CIFAR10 dataset.

4.1 Dataset

We'll use the relatively simple but not easy MNIST and CIFAR10 dataset for our experiment. These datasets gives us a big enough challenge that most parameters won't do well, while also being easy enough for an MLP to get a decent accuracy score. A convolutional neural network is certainly the better choice for a 10-class image classification problem like CIFAR10. But a fully connected network will do just fine for illustrating the effectiveness of using a genetic algorithm for hyper-parameter tuning.

The MNIST database comprises 60,000 training images and 10,000 testing images. Half of the training set and half of the test set were taken from NIST's training dataset, while the other half of the training set and the other half of the test set were taken from NIST's testing dataset.

The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.

The dataset is distributed into five training batches and one test batch, each with 10000 images. The test batch comprises exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5000 images from each class.

4.2 Process

- **Applying genetic algorithms to Neural Networks**

We'll attempt to evolve a fully connected network (MLP). Our goal is to find the best parameters for an image classification task.

We'll tune four parameters:

- Number of layers (or the network depth)
- Neurons per layer (or the network width)
- Dense layer activation function
- Network optimizer

The steps we'll take to evolve the network, similar to those described above, are:

1. Initialize *N* random networks to create our population.
2. Score each network. This takes some time: We have to train the weights of each network and then see how well it performs at classifying the test set. Since this will be an image classification task, we'll use classification accuracy as our fitness function.
3. Sort all the networks in our population by score (accuracy). We'll keep some percentage of the top networks to become part of the next generation and to breed children.
4. We'll also randomly keep a few of the non-top networks. This helps find potentially lucky combinations between worse-performers and top performers, and also helps keep us from getting stuck in a local maximum.
5. Now that we've decided which networks to keep, we randomly mutate some of the parameters on some of the networks.
6. Let's say we started with a population of 20 networks, we kept the top 25% (5 nets), randomly kept 3 more loser networks, and mutated a few of them. We let the other 12 networks die. In an effort to keep our population at 20 networks, we need to fill 12 open spots.

- **Breeding**

Breeding is where we take two members of a population and generate one or more child, where that child represents a combination of its parents.

In our neural network case, each child is a combination of a random assortment of parameters from its parents. For instance, one child might have the same number of layers as its mother and the rest of its parameters from its father. A second child of the same parents may have the opposite. You can see how this mirrors real-world biology and how it can lead to an optimized network quickly.

Also instead of the traditional back-propogation algorithm, we use the synthetic gradients to approximate error gradients and hence fast converging to real values.

4.3 Some python libraries used for the process

- **NumPy** – NumPy is the fundamental package for scientific computing with Python. It contains among other things:
  - a powerful N-dimensional array object
  - sophisticated (broadcasting) functions
  - tools for integrating C/C++ and Fortran code
  - useful linear algebra
  - Fourier transform, and random number capabilities

  Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data. Arbitrary data-types can be defined. This allows NumPy to seamlessly and speedily integrate with a wide variety of databases.

- **Scipy** – SciPy is a collection of mathematical algorithms and convenience functions built on the Numpy extension of Python. It adds significant power to the interactive Python session by providing the user with high-level commands and classes for manipulating and visualizing data. With SciPy an interactive Python session becomes a data-processing and system-prototyping environment rivaling systems such as MATLAB, IDL, Octave, R-Lab, and SciLab. GA1 – An incrementally learning algorithm that assesses attributes individually and orders them based on their relevance to the problem.

- **Scikit Learn**-Scikit-learn (formerly scikits.learn) is a free software machine learning library for the Python programming language.It features various classification, regression and clustering algorithms including support vector machines, random forests, gradient boosting, *k*-means and DBSCAN, and is designed to interoperate with the Python numerical and scientific libraries NumPy and SciPy.

- **TensorFlow**- TensorFlow is an open-source software library for dataflow programming across a range of tasks. It is a symbolic math library, and also used for machine learning applications such as neural networks. TensorFlow is an open source library for fast numerical computing. It was created and is maintained by Google and released under the Apache 2.0 open source license. The API is nominally for the Python programming language, although there is access to the underlying C++ API. Unlike other numerical libraries intended for use in Deep Learning like Theano, TensorFlow was designed for use both in research and development and in production systems, not least RankBrain in Google search and the fun DeepDream project. It can run on single CPU systems, GPUs as well as mobile devices and large scale distributed systems of hundreds of machines.

- **Keras** - Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. It was developed with a focus on enabling fast experimentation. Keras is a deep learning library that:
  Allows for easy and fast prototyping (through user friendliness, modularity, and extensibility), Supports both convolutional networks and recurrent networks, as well as combinations of the two and runs seamlessly on CPU and GPU.

CHAPTER 5

RESULTS AND ANALYSIS

5.1MNIST DATABASE

We’ll start by running the brute force algorithm on the relatively simple MNIST database to find the best network. That is, we’ll train and test every possible combination of network parameters we provided.

Top result using brute force achieved **98.33% accuracy** with the following parameters:

- **Layers**: 2
- **Neurons**: 512
- **Activation**: relu
- **Optimizer**: rmsprop

This took ***2hours 22 min* to run.**

SNO	OPTIMISER	NB_LAYERS	NB_NEURONS	ACTIVATOR	ACCURACY
1	rmsprop	2	128	Relu	97.97
2	rmsprop	2	128	Relu	97.90
3	rmsprop	2	128	Elu	98.12
4	Adam	3	128	Relu	98.10
5	rmsprop	2	512	Relu	98.33
6	Adam	2	512	Relu	98.23
7	rmsprop	3	128	Elu	98.14

Table 5.1-Result of Brute Force Algo on MNIST database

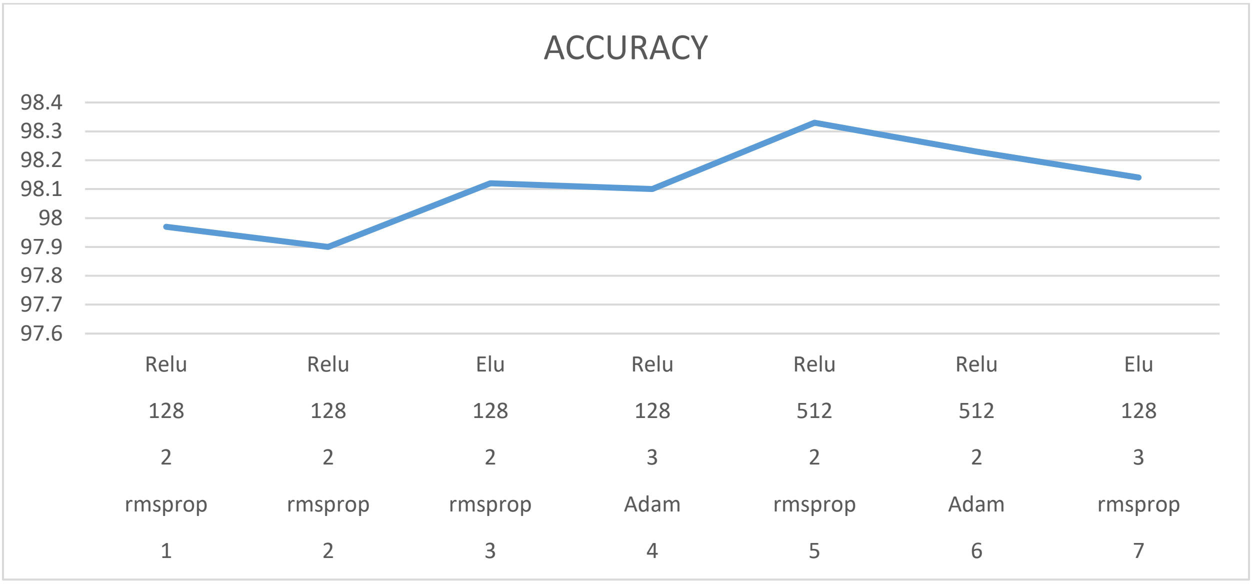


Fig 5.1-Accuracy measure of brute force on MNIST database

Now we use Genetic Algorithm to find the best parameter for the problem: Starting with a population of 15 randomly initialized networks, and we'll run it for 5 generations.

Top result using GA achieved **99.02% accuracy** with the following parameters:

- **Layers**: 3
- **Neurons**: 512
- **Activation**: relu
- **Optimizer**: adam

However, the best thing is **this took just 1*hour 15 min* to run.**

GEN	OPTIMISER	NB_LAYERS	NB_NEURONS	ACTIVATOR	ACCURACY
1	Adam	3	128	elu	98.17
2	Adam	2	512	elu	98.12
3	Adam	2	128	elu	98.22

4	Adam	2	128	Relu	98.26
5	Adam	3	512	Relu	99.02

Table 5.2- Result of GA on MNIST database

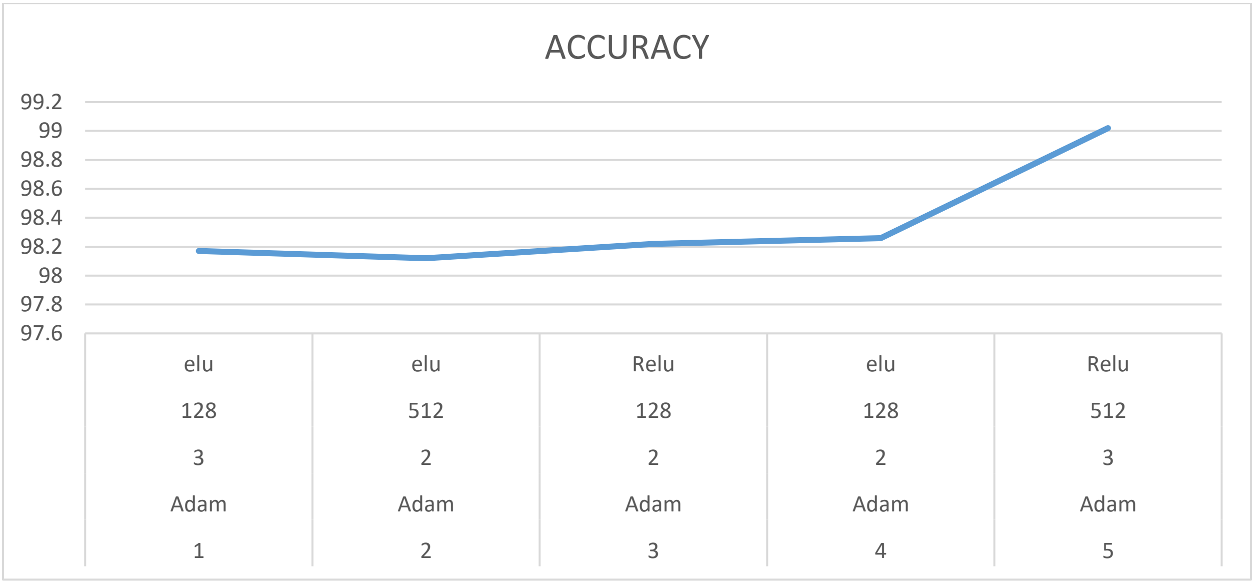


Fig 5.2-Accuracy measure of GA on MNIST database

5.2CIFAR10 DATABASE

Now we run the brute force algorithm on CIFAR10 database. Although a CNN architecture would have performed better for image classification task but the MLP would do just fine to demonstrate our purpose.

Top result using brute force achieved **51.53% accuracy** with the following parameters:

- **Layers**: 2
- **Neurons**: 512
- **Activation**: elu
- **Optimizer**: adam

This took ***4hours 05 min* to run.**

SNO	OPTIMISER	NB_LAYERS	NB_NEURONS	ACTIVATOR	ACCURACY
1	rmsprop	3	128	elu	49.46
2	adam	2	512	elu	51.53
3	adam	2	128	elu	48.71
4	rmsprop	2	512	Relu	36.38
5	rmsprop	2	128	elu	44.93
6	rmsprop	2	512	elu	10.00
7	adam	3	128	relu	46.03

Table 5.3-Result of brute force on CIFAR10 database

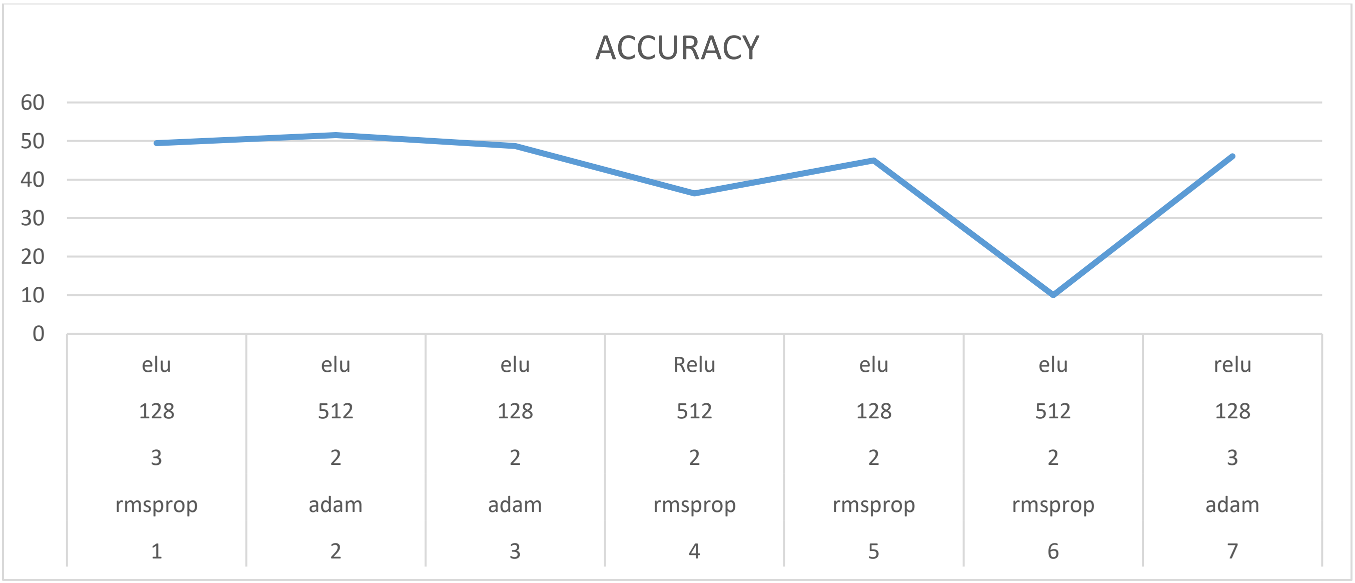


Fig 5.3-Accuracy measure of brute force on CIFAR10 database

Now we use Genetic Algorithm to find the best parameter for the problem: Starting with a population of 15 randomly initialized networks, and we'll run it for 5 generations.

Top result using GA achieved **52.15% accuracy** with the following parameters:

- **Layers**: 3
- **Neurons**: 512
- **Activation**: elu
- **Optimizer**: adam

However, the best thing is **this took just 1*hour 45 min* to run.**

GEN	OPTIMISER	NB_LAYERS	NB_NEURONS	ACTIVATOR	ACCURACY
1	Adam	3	128	elu	50.89
2	Adam	2	128	Relu	51.02
3	Adam	3	512	Relu	51.33
4	Adam	3	512	elu	51.41
5	Adam	3	512	elu	52.15

Table 5.4- Result of GA on Cifar10 database

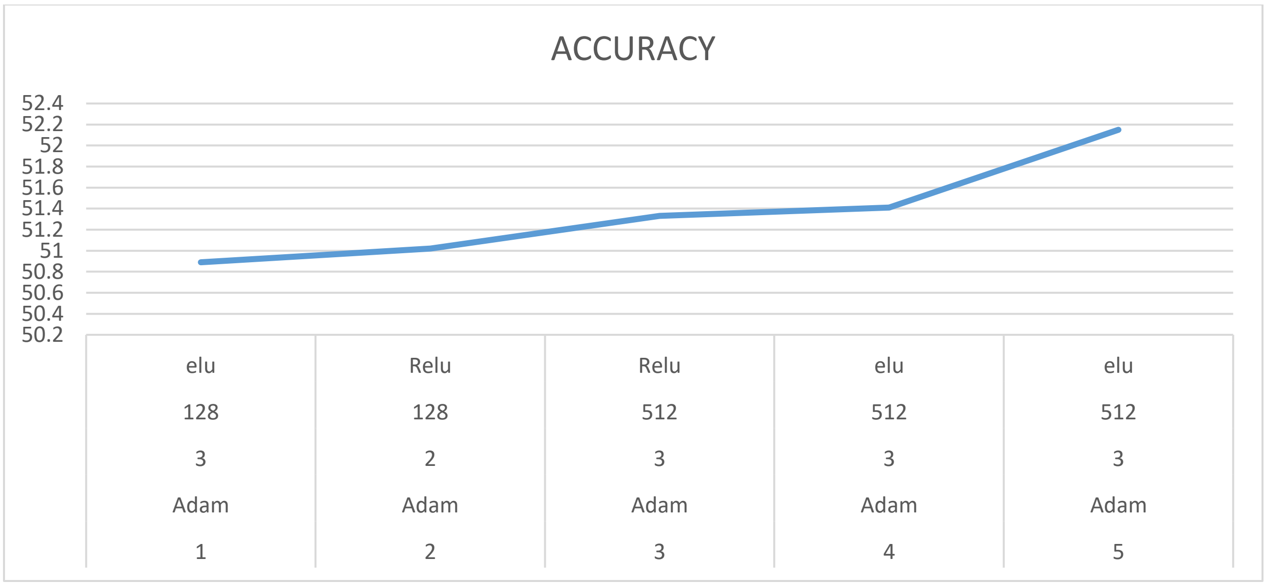


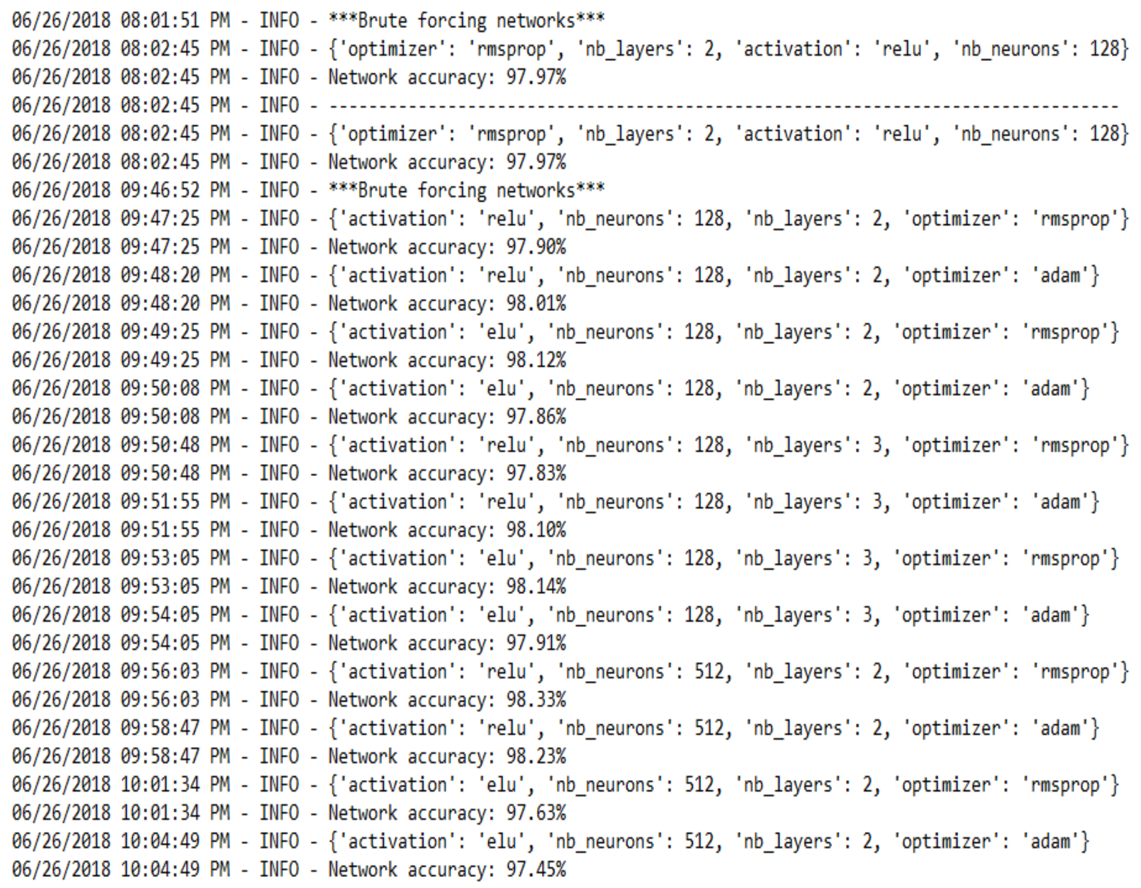
Fig.5.4-Accuracy measure of GA on CIFAR10 database

5.3 Analysis

The genetic algorithm gave us the same result in almost less than half the time taken by brute force. And it's likely that as the parameter complexity increases, the genetic algorithm provides exponential speed benefit.

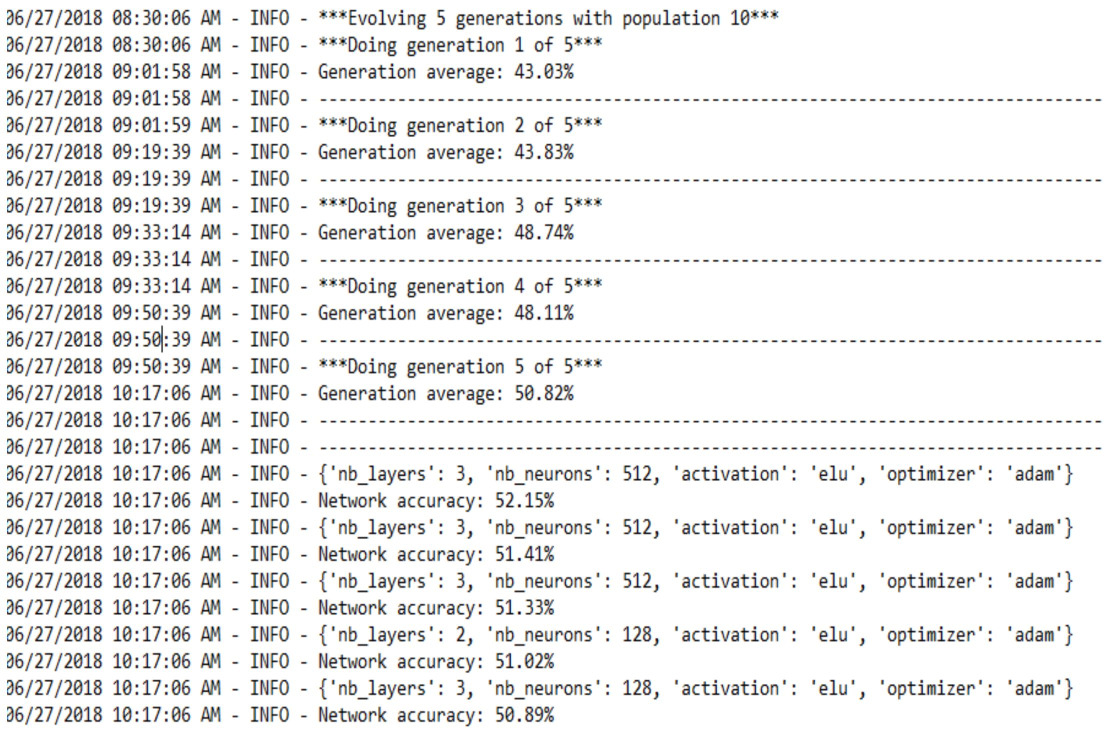
On the MNIST database the accuracy measured was very high due to its simplicity in design. Here the GA outperformed brute force in terms of accuracy and time both in just 5 generations.

On the CIFAR10 database also we could achieve lesser accuracy due to image classification problem which can be more effectively done using a CNN architecture. However here also we demonstrated how GA outperformed brute force in just 5 generations. With more number of generations and added parameter possibilities we can expect more accuracy using GA.



```
06/26/2018 08:01:51 PM - INFO - ***Brute forcing networks***
06/26/2018 08:02:45 PM - INFO - {'optimizer': 'rmsprop', 'nb_layers': 2, 'activation': 'relu', 'nb_neurons': 128}
06/26/2018 08:02:45 PM - INFO - Network accuracy: 97.97%
06/26/2018 08:02:45 PM - INFO - --------------------------------------------------------------------------------
06/26/2018 08:02:45 PM - INFO - {'optimizer': 'rmsprop', 'nb_layers': 2, 'activation': 'relu', 'nb_neurons': 128}
06/26/2018 08:02:45 PM - INFO - Network accuracy: 97.97%
06/26/2018 09:46:52 PM - INFO - ***Brute forcing networks***
06/26/2018 09:47:25 PM - INFO - {'activation': 'relu', 'nb_neurons': 128, 'nb_layers': 2, 'optimizer': 'rmsprop'}
06/26/2018 09:47:25 PM - INFO - Network accuracy: 97.90%
06/26/2018 09:48:20 PM - INFO - {'activation': 'relu', 'nb_neurons': 128, 'nb_layers': 2, 'optimizer': 'adam'}
06/26/2018 09:48:20 PM - INFO - Network accuracy: 98.01%
06/26/2018 09:49:25 PM - INFO - {'activation': 'elu', 'nb_neurons': 128, 'nb_layers': 2, 'optimizer': 'rmsprop'}
06/26/2018 09:49:25 PM - INFO - Network accuracy: 98.12%
06/26/2018 09:50:08 PM - INFO - {'activation': 'elu', 'nb_neurons': 128, 'nb_layers': 2, 'optimizer': 'adam'}
06/26/2018 09:50:08 PM - INFO - Network accuracy: 97.86%
06/26/2018 09:50:48 PM - INFO - {'activation': 'relu', 'nb_neurons': 128, 'nb_layers': 3, 'optimizer': 'rmsprop'}
06/26/2018 09:50:48 PM - INFO - Network accuracy: 97.83%
06/26/2018 09:51:55 PM - INFO - {'activation': 'relu', 'nb_neurons': 128, 'nb_layers': 3, 'optimizer': 'adam'}
06/26/2018 09:51:55 PM - INFO - Network accuracy: 98.10%
06/26/2018 09:53:05 PM - INFO - {'activation': 'elu', 'nb_neurons': 128, 'nb_layers': 3, 'optimizer': 'rmsprop'}
06/26/2018 09:53:05 PM - INFO - Network accuracy: 98.14%
06/26/2018 09:54:05 PM - INFO - {'activation': 'elu', 'nb_neurons': 128, 'nb_layers': 3, 'optimizer': 'adam'}
06/26/2018 09:54:05 PM - INFO - Network accuracy: 97.91%
06/26/2018 09:56:03 PM - INFO - {'activation': 'relu', 'nb_neurons': 512, 'nb_layers': 2, 'optimizer': 'rmsprop'}
06/26/2018 09:56:03 PM - INFO - Network accuracy: 98.33%
06/26/2018 09:58:47 PM - INFO - {'activation': 'relu', 'nb_neurons': 512, 'nb_layers': 2, 'optimizer': 'adam'}
06/26/2018 09:58:47 PM - INFO - Network accuracy: 98.23%
06/26/2018 10:01:34 PM - INFO - {'activation': 'elu', 'nb_neurons': 512, 'nb_layers': 2, 'optimizer': 'rmsprop'}
06/26/2018 10:01:34 PM - INFO - Network accuracy: 97.63%
06/26/2018 10:04:49 PM - INFO - {'activation': 'elu', 'nb_neurons': 512, 'nb_layers': 2, 'optimizer': 'adam'}
06/26/2018 10:04:49 PM - INFO - Network accuracy: 97.45%
```

Fig.5.5-A snapshot of the brute force log on MNIST database



```
06/27/2018 08:30:06 AM - INFO - ***Evolving 5 generations with population 10***
06/27/2018 08:30:06 AM - INFO - ***Doing generation 1 of 5***
06/27/2018 09:01:58 AM - INFO - Generation average: 43.03%
06/27/2018 09:01:58 AM - INFO - --------------------------------------------------------------------------------
06/27/2018 09:01:59 AM - INFO - ***Doing generation 2 of 5***
06/27/2018 09:19:39 AM - INFO - Generation average: 43.83%
06/27/2018 09:19:39 AM - INFO - --------------------------------------------------------------------------------
06/27/2018 09:19:39 AM - INFO - ***Doing generation 3 of 5***
06/27/2018 09:33:14 AM - INFO - Generation average: 48.74%
06/27/2018 09:33:14 AM - INFO - --------------------------------------------------------------------------------
06/27/2018 09:33:14 AM - INFO - ***Doing generation 4 of 5***
06/27/2018 09:50:39 AM - INFO - Generation average: 48.11%
06/27/2018 09:50:39 AM - INFO - --------------------------------------------------------------------------------
06/27/2018 09:50:39 AM - INFO - ***Doing generation 5 of 5***
06/27/2018 10:17:06 AM - INFO - Generation average: 50.82%
06/27/2018 10:17:06 AM - INFO - --------------------------------------------------------------------------------
06/27/2018 10:17:06 AM - INFO - --------------------------------------------------------------------------------
06/27/2018 10:17:06 AM - INFO - {'nb_layers': 3, 'nb_neurons': 512, 'activation': 'elu', 'optimizer': 'adam'}
06/27/2018 10:17:06 AM - INFO - Network accuracy: 52.15%
06/27/2018 10:17:06 AM - INFO - {'nb_layers': 3, 'nb_neurons': 512, 'activation': 'elu', 'optimizer': 'adam'}
06/27/2018 10:17:06 AM - INFO - Network accuracy: 51.41%
06/27/2018 10:17:06 AM - INFO - {'nb_layers': 3, 'nb_neurons': 512, 'activation': 'elu', 'optimizer': 'adam'}
06/27/2018 10:17:06 AM - INFO - Network accuracy: 51.33%
06/27/2018 10:17:06 AM - INFO - {'nb_layers': 2, 'nb_neurons': 128, 'activation': 'elu', 'optimizer': 'adam'}
06/27/2018 10:17:06 AM - INFO - Network accuracy: 51.02%
06/27/2018 10:17:06 AM - INFO - {'nb_layers': 3, 'nb_neurons': 128, 'activation': 'elu', 'optimizer': 'adam'}
06/27/2018 10:17:06 AM - INFO - Network accuracy: 50.89%
```

Fig. 5.6- A snapshot of GA on CIFAR10 database

CHAPTER 6

CONCLUSION

We have accomplished a number of things with our work on using genetic algorithms to train feedforward networks. In the field of genetic algorithms, we have demonstrated a real-world application of a genetic algorithm to a large and complex problem. We have also shown how adding domain specific knowledge into the genetic algorithm can enhance its performance. In the terms of neural networks, we have introduced a new type of training algorithm, the synthetic gradient which on our data outperforms the backpropagation algorithm. The work described here only touches the surface of the potential for using genetic algorithms to train neural networks. In the realm of feedforward networks, there are a host of other operators with which one might experiment. Perhaps most promising are ones which include backpropagation as all or part of their operation. Another problem is how to modify the genetic algorithm so that it deals with a stream of continually changing training data instead of fixed training data. This requires modifying the genetic algorithm to handle a stochastic evaluation function. Finally, as a general-purpose optimization tool, genetic algorithms should be applicable to any type of neural network (and not just feedforward networks whose nodes have smooth transfer functions) for which an evaluation function can be derived. The existence of genetic algorithms for training could aid in the development of other types of neural networks.

This thesis proposes the use of a Genetic Algorithm to find the architecture of neural networks and the learning parameters associated for that network. This network is then customized to individual applications. The genetic algorithm reduces the computational complexity and increases the accuracy of the MLP neural network. However, the optimizer has to be better than a human optimizing a network to make it beneficial.

Test results shows that genetic algorithms can be used to optimize a neural network. In this example the genetic algorithm increases the accuracy and decreases amount of computation time by 50% due to increasing generalization. Future efforts contain a more flexible chromosome so the type of layers to the solution is not predetermined. This could be applied to many preexisting networks and additional types of layers to increase the accuracy, increase the computational efficiency, and decrease development time.

REFERENCES

[2] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. Genetic Programming: An Introduction. Morgan Kaufmann, San Meateo, CA, 1998.

[3] SJ. Bergstra and Y. Bengio. Random Search for Hyper-Parameter Optimization. Journal of Machine Learning Research, 13:281–305, 2012.

[4] Randal S. Olson, Ryan J. Urbanowicz, Peter C. Andrews, Nicole A. Lavender, La Creis Kidd, and Jason H. Moore (2016). Automating biomedical data science through tree-based pipeline optimization. *Applications of Evolutionary Computation*, pages 123-137.

[5] J. M. Kanter and K. Veeramachaneni. Deep Feature Synthesis: Towards Automating Data Science Endeavors. In Proceedings of the International Conference on Data Science and Advance Analytics. IEEE, 2015.

[6] J. Zutty et al. Multiple objective vector-based genetic programming using human-derived primitives. In Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation, GECCO ’15, pages 1127–1134, New York, NY, USA, 2015. ACM.

[7] E. M. Fredericks and B. H. Cheng. Exploring automated software composition with genetic programming. In Proceedings of the 15th Annual Conference Companion on Genetic and Evolutionary Computation, GECCO ’13 Companion, pages 1733–1734, New York, NY, USA, 2013. ACM.

[8]W. Wu, W. Guozhi, Z. Yuanmin and W. Hongling, "Genetic Algorithm Optimizing Neural Network for Short-Term Load Forecasting," *International Forum on Information Technology and Applications*, pp. 583-585, 2009

[9] Z. Chen, "Optimization of Neural Network Based on Improved Genetic Algorithm," *International Conference on Computational Intelligence and Software Engineering,* pp.1-3, 2009

[10] X. Chen, X. Xu, Y. Wang, and X. Zhong, “Found on Artificial Neural Network and Genetic Algorithm Design the ASSEL Roll Profile,” *Proceedings of the International Conference on Computer Science and Software Engineering,* Vol. 01, pp. 40-44, 2008

[11] C. Tang, Y. He and L. Yuan, “A Fault Diagnosis Method of Switch Current Based on Genetic Algorithm to Optimize the BP Neural Network,” *International Conference on Electric and Electronics,* Vol. 99, pp. 943-950, 2011

[12] S. Zeng, J. Li and L. Cui, “Cell Status Diagnosis for the Aluminum Production on BP Neural Network with Genetic Algorithm,” *Communications in Computer and Information Science,* Vol. 175, pp. 146-152, 2011

[13] S. Nie and B. Ye, "The Application of BP Neural Network Model of DNA-Based Genetic Algorithm to Monitor Cutting Tool Wear," *International Conference on Measuring Technology and Mechatronics Automation*, pp. 338-341, 2009

[14] W. Yinghua and X. Chang, "Using Genetic Artificial Neural Network to Model Dam Monitoring Data," *Second International Conference on Computer Modeling and Simulation,* pp. 3-7, 2010

[15] D. Sabo, “A Modified Iterative Pruning Algorithm for Neural Network Dimension Analysis”, A Thesis for California Polytechnic State University, October, 2007.

[16] X. Fu, P.E.R. Dale and S. Zhang, “Evolving Neural Network Using Variable String Genetic Algorithms (VGA) for Color Infrared Aerial Image Classification,” *Chinese Geographical Science,* Vol. 18(2), pp. 162-170, 2008

[17] P. Koehn, “Combining Genetic Algorithms and Neural Networks: The Encoding Problem,” University of Tennessee, Knoxville, 1994

[18] D. Whitley, T. Starkweather and C. Bogart, “Genetic Algorithms and Neural Networks: Optimizing Connections and Connectivity,” *Parallel Computing*, Vol. 14, pp. 347-361, 1990

[19] P. W. Munro, “Genetic Search for Optimal Representation in Neural Networks,” *Proceedings of the International Joint Conference on Neural Networks and Genetic Algorithms,* pp. 675-682, 1993

[20] D. Dasgupta and D. R. McGregor, “Designing Application-Specific Neural Networks using the Structured Genetic Algorithm,” *Proceedings of International Workshop on Combinations of Genetic Algorithms and Neural Networks,* pp. 87- 96, 1992

[21] J. M. Bishop and M. J. Bushnell, “Genetic Optimization of Neural Network Architectures for Colour Recipe Prediction,” *Proceedings of the International Joint Conference on Neural Networks and Genetic Algorithms,* pp. 719-725, 1993

[22] S. A. Harp and T. Samad, “Genetic Synthesis of Neural Network Architecture,” *Handbook of Genetic Algorithms*, pp. 202-221, 1991

[23] G. G. Yen and H. Lu, “Hierarchical Genetic Algorithm Based Neural Network Design,” *IEEE Symposium on Combinations of Evolutionary Computation and Neural Networks,* pp. 168-175, 2000

[24] V. Bevilacqua, G. Mastronardi, F. Menolascina, P. Pannarale and A. Pedone, “A Novel Multi-Objective Genetic Algorithm Approach to Artificial Neural Network Topology Optimisation: The Breast Cancer Classification Problem,” *International Joint Conference on Neural Networks,* pp. 1958-1965, 2006

[25]J. Cheng andQ.S. Li, Reliability analysis of structures using artificialneural network based genetic algorithms, Comput. Methods Appl. Mech. Engrg. 197, 3742-3750 (2008).

[26] S.M.R. Longhmanian, H. Jamaluddin, R. Ahmad, R. Yusof and M. Khalid, Structure optimization of neural network for dynamic system modeling using multi-objective genetic algorithm, Neural Comput. Appl. 21, 1281-1295 (2012).