ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΜΕ ΕΦΑΡΜΟΓΕΣ ΣΤΗ ΒΙΟΙΑΤΡΙΚΗ

Genetic Algorithms for low and full rank Convolutional Neural Network Training

Fani Tzina

A THESIS
Presented to University of Thessaly
in
Partial Fulfillment of the Requirements for the
Degree of Computer Science and Biomedical Informatics
2023

Supervisor of Thesis
Konstantinos Delibasis,
Associate Professor,
Department of Computer Science and Biomedical Informatics,

Με ατομική μου ευθύνη και γνωρίζοντας τις κυρώσεις [1], που προβλέπονται από τις διατάξεις της παρ. 6 του άρθρου 22 του Ν. 1599/1986, δηλώνω ότι:

1. *Δεν παραθέτω κομμάτια βιβλίων ή άρθρων ή εργασιών άλλων αυτολεξεί **χωρίς να τα περικλείω σε εισαγωγικά** και χωρίς να αναφέρω το συγγραφέα, τη χρονολογία, τη σελίδα. Η αυτολεξεί παράθεση χωρίς εισαγωγικά χωρίς αναφορά στην πηγή, είναι λογοκλοπή. Πέραν της αυτολεξεί παράθεσης, λογοκλοπή θεωρείται και η παράφραση εδαφίων από έργα άλλων, συμπεριλαμβανομένων και έργων συμφοιτητών μου, καθώς και η παράθεση στοιχείων που άλλοι συνέλεξαν ή επεξεργάσθηκαν, χωρίς αναφορά στην πηγή. Αναφέρω πάντοτε με πληρότητα την πηγή κάτω από τον πίνακα ή σχέδιο, όπως στα παραθέματα.*
2. *Δέχομαι ότι η αυτολεξεί **παράθεση χωρίς εισαγωγικά**, ακόμα κι αν συνοδεύεται από αναφορά στην πηγή σε κάποιο άλλο σημείο του κειμένου ή στο τέλος του, είναι αντιγραφή. Η αναφορά στην πηγή στο τέλος π.χ. μιας παραγράφου ή μιας σελίδας, δεν δικαιολογεί συρραφή εδαφίων έργου άλλου συγγραφέα, έστω και παραφρασμένων, και παρουσίασή τους ως δική μου εργασία.*
3. *Δέχομαι ότι υπάρχει επίσης περιορισμός στο μέγεθος και στη συχνότητα των παραθεμάτων που μπορώ να εντάξω στην εργασία μου εντός εισαγωγικών. Κάθε μεγάλο παράθεμα (π.χ. σε πίνακα ή πλαίσιο, κλπ), προϋποθέτει ειδικές ρυθμίσεις, και όταν δημοσιεύεται προϋποθέτει την άδεια του συγγραφέα ή του εκδότη. Το ίδιο και οι πίνακες και τα σχέδια*
4. *Δέχομαι όλες τις συνέπειες σε περίπτωση λογοκλοπής ή αντιγραφής.*

Ημερομηνία: ……/…../20……

Ο – Η Δηλ.

(Υπογραφή)

(1) «Όποιος εν γνώσει του δηλώνει ψευδή γεγονότα ή αρνείται ή αποκρύπτει τα αληθινά με έγγραφη υπεύθυνη δήλωση του άρθρου 8 παρ. 4 Ν. 1599/1986 τιμωρείται με φυλάκιση τουλάχιστον τριών μηνών. Εάν ο υπαίτιος αυτών των πράξεων σκόπευε να προσπορίσει στον εαυτόν του ή σε άλλον περιουσιακό όφελος βλάπτοντας τρίτον ή σκόπευε να βλάψει άλλον, τιμωρείται με κάθειρξη μέχρι 10 ετών.

Contents

Abstract

This work investigates the use of Genetic Algorithms (GAs) to train Convolutional Neural Networks (CNNs) in order to improve network's generalization. Experiments were conducted to evaluate the effect of varying training set sizes, random subsets of the training set per individual, and the rank of the network tensor of the convolutional layer on the test accuracy of the CNN. For this purpose, we constructed a very simple CNN with one convolutional layer and two fully connected layers. We used the MNIST dataset for experimentation. In detail, first we experimented with the influence of narrowing the training subset for each individual of the population, by comparing the training accuracy with the validation/test accuracy during the evolution. The results showed that test accuracy decreased from 74% to 67.4% and 64.2% when the training set was decreased from 5000 to 200 and 100 images, respectively. We also investigated the case of constructing the weight tensors of the convolutional layer with lower rank and the results have shown that the rank of the network tensor had no statistically significant effect on the test accuracy of the designed CNN. Finally, applying the hybridization of GAs with BP by fine tuning the best chromosome of the generations with BP caused quick loss of genetic diversity, resulting in overfitting, which was evident in the results, achieving 100% train accuracy and very poor test accuracy. It is important to mention that no overfitting was observed during the GA-training of the CNN, even after 400 generations. These results suggest that GAs can be used to train CNNs with improved generalization.

1. Introduction

Due to the technological advancements that have occurred in the field of artificial intelligence and computer information technology over the past few years, the number of new methods and techniques that have been developed to solve pattern recognition problems have increased. One of these is the Artificial Neural Network (ANN), in the context of deep learning. The ANN concept is based on relevant biological activities of the human brain. Currently, neural networks are performing well in many areas such as facial recognition, market forecasting, and risk assessment. ANN is a term used collectively for traditional architectures, such as Multilayered Perceptrons -MLPs-, as well as for deep learning ones, such as convolutional neural networks (CNNs). Backpropagation is the algorithm that dominates the training of ANNs, with many variations of the optimization function. Backpropagation (BP) involves feedforward of the input training pattern, error computation, and adjustment of synaptic weights. BP is a gradient descent optimization method; thus, it is essentially a local optimization, despite variations like the introduction of momentum. Currently, this method is applied in many fields such as predictive decision-making, orbital placement, and system planning. The applicability and good performance of network models in many fields make it a hotspot studied by relevant experts and scholars.

Genetic Algorithms (GAs) on the other hand is a global optimization that uses population dynamics, instead of gradient information. The main drawback is the slow convergence, however its ability to escape local optima of the fitness function landscape is a great advantage.
A CNN training by GAs (GA-CNN) is designed to be able to evolve a population of CNNs encoded into chromosomes, in order to better solve a given problem. This ability to adapt and change over time is what makes this type of global optimization so powerful. The aim of this work is to delve into the combination of GAs and BP for CNN training.

Aims and Contributions

More specifically, we attempt to investigate the role of genetic algorithms in training CNNs for image classification, in terms of accuracy, compared to backpropagation. GA is a type of artificial intelligence technique that uses evolutionary algorithms in order to evolve solutions for a given problem. GA can be used to find the optimal weights for a convolutional neural network (CNN) by creating a population of individuals (that contain the network's weights) and then using evolutionary operators to generate new weights from the existing ones. The best weights from the population are then selected and used for the CNN. This process can be repeated multiple times to find the optimal set of weights that leads to the highest accuracy.
The main contributions of this project are reflected in the following two levels: an optimized Genetic Convolution Neural Network (GA-CNN) algorithm and a hybrid genetic-backpropagation algorithm (GA-BPNN) are proposed and applied for the Digit and Type of

Clothing Recognition. The GA-CNN algorithm combines the genetic algorithm with the CNN model, allowing effective training to be achieved by optimizing the CNN model. To evaluate the performance of the GA-CNN algorithm, the accuracy of the digit and type of clothing recognition tasks are compared to the traditional backpropagation algorithm. More specific issues are being investigated:

- the degree of overfitting by the GA-based training, by comparing the training accuracy with the validation/test accuracy during the evolution
- the effect of lowering the rank of the weights of the convolutional layer, in terms of the achieved classification accuracy
- the effect of narrowing the training subset, globally for all individuals of the population
- the influence of narrowing the training subset, for each individual of the population
- the role of hybridizing GAs with BP and the effect of the frequency of this hybridization

The results are produced for a small dataset and presented and discussed in the following sections.

2. Background

2.1 Convolutional Neural Networks

Feedforward Neural Networks can be used to solve any kind of regression or classification problems but lacks in the field of computer vision as the number of parameters to optimize is very high. In fully connected layers also ANNs cannot identify the objects in each image. Due to these reasons, ANN is not recommended for identifying the object in an image. The usage of Convolution Neural Networks in the field of Image classification has achieved remarkable success in recent years.

Convolutional Neural Network (CNN) is specialized for image recognition. The CNN architecture can be divided into two sections: Feature learning and Classification section. The input image enters the feature extraction network. The extracted feature signals enter the Classification Neural Network. This section operates based on the features of the image and generates the output. The feature extraction neural network consists of the piles of the convolutional layer and pooling layer pairs. The convolutional layer, as its name implies, converts the image using the convolution operation - it can be thought of as a collection of digital filters. The pooling layer combines the neighboring pixels into a single pixel; therefore, the pooling layer reduces the dimension of the image. As the primary concern of the CNN is the image, the operations of the convolutional and pooling layers are conceptually in a two-dimensional plane. This is one of the differences between CNN and other neural networks.

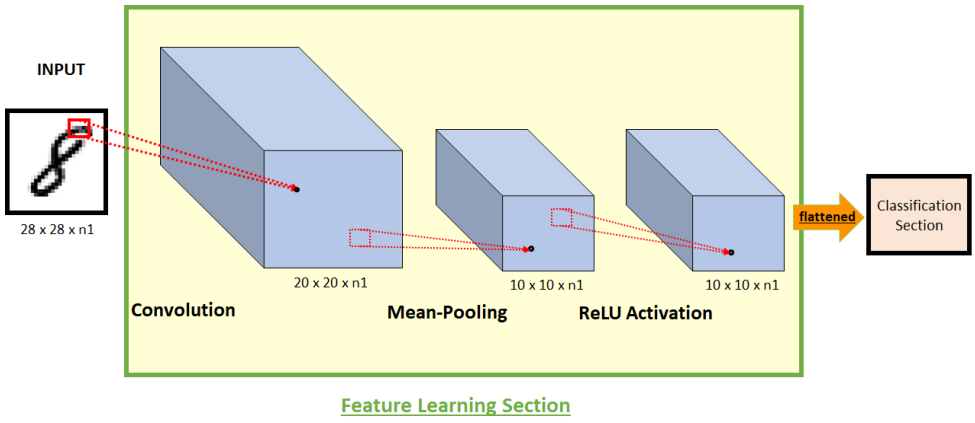


Figure 1: The architecture of a convolution neural network with 1 hidden layer

The convolutional layer generates feature maps from images. The working principle of this layer is different from other neural network layers. It does not employ connection weights and a weighted sum. Instead, it contains filters that convert images. These filters are called Convolutional Filters. The number of Feature maps and the number of convolutional filters is

the same. That means, if there are four convolution layers, then it will generate four feature maps.

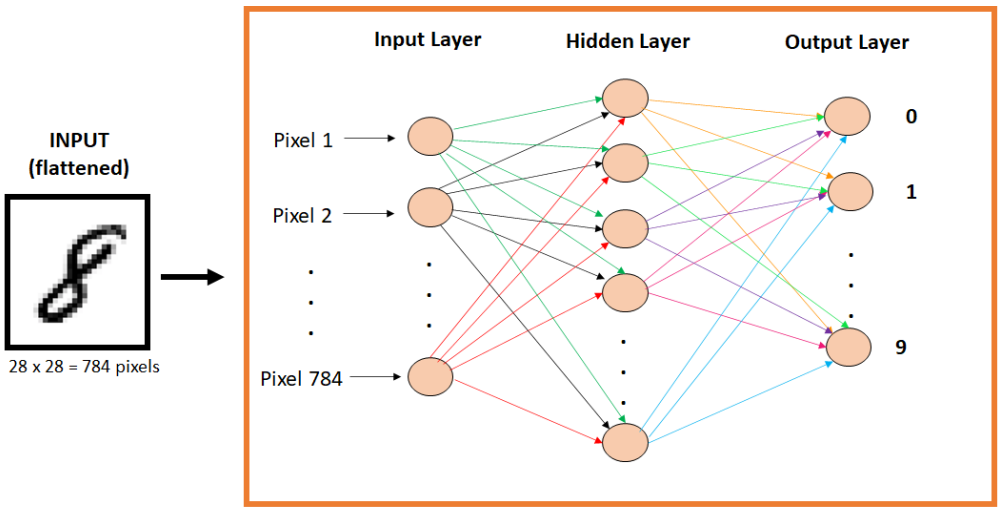


Figure 2: Classification Neural network

2.2 Neural Network training using Backpropagation

The backpropagation algorithm is used to train a neural network. By comparing desired outputs to achieved system outputs, the systems are tuned by adjusting connection weights to narrow the difference between the two as much as possible.

As referred to in [Joseph Tarigan, 2017] and [Phil Kim, 2017], the main feature of the backpropagation algorithm is the iterative and recursive method of calculating and updating the weights based on the error rate of the previous epoch. In detail, the backpropagation algorithm works by first calculating the error of the network, then propagating the error backwards through the network to adjust the weights of each layer, and finally updating the weights to reduce the error. The process is repeated until the error is minimized. This process is known as gradient descent (Fig.3).

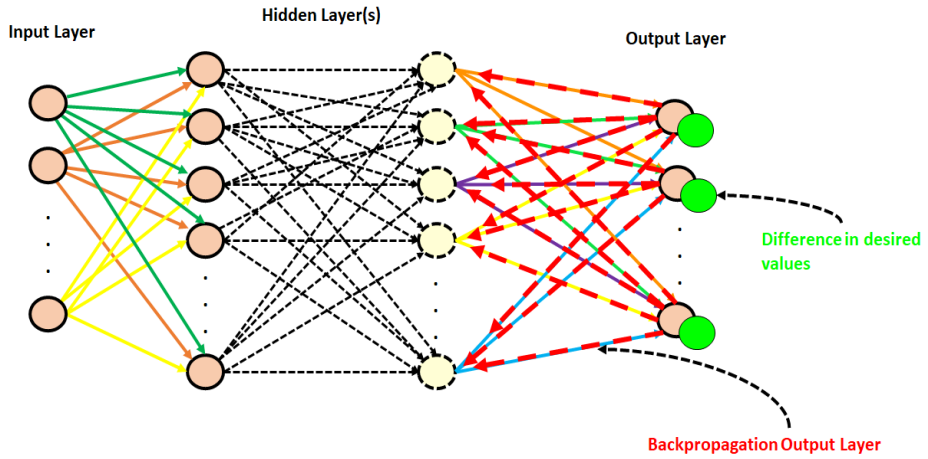


Figure 3: Training of CNN with backpropagation

The speed of the training process and the accuracy of image recognition is affected by two variables: learning rate and momentum rate. The learning rate indicates the step size towards the minimum value of the loss function when following the gradient descent, whereas the momentum weight considers previous weight changes when updating the current weights. Furthermore, the optimal learning rate and momentum rate values for one backpropagation neural network topology may not be optimal for another backpropagation neural network topology. This is because each topology is specific to its domain of use. These two variables also influence the tendency of the backpropagation process to fall into the local minimum, making the artificial neural network's detection performance suboptimal.

It also must be mentioned that the number of neurons in the hidden layer also affects the speed of the backpropagation process and its accuracy. As expected, the more neurons, the more information can be stored, but this affects the speed of the training process. A suboptimal number of neurons reduces the ability to generalize, whereas too many hidden neurons lead to overfitting problems. In both cases, the artificial neural network yields sub-optimal performance.

2.3 Neural Network training using Genetic Algorithm

Evolutionary algorithms have been used for long in the field of image analysis. Artificial immune system is an algorithm of this family that has been proposed for classification [K. Delibasis et. al., 2008] and finding corresponding points in images [K. Delibasis et. al., 2011]. The main representative of evolutionary algorithms is Genetic Algorithms (GAs). GAs is a global optimization technique that is known to optimize multidimensional landscape functions without the use of derivative information [Goldberg, 1988]. Genetic Algorithm is a heuristic search algorithm that is inspired by the biological evolution analogy of crossing over the fittest chromosomes to generate superior offspring. This algorithm works by applying random changes to current chromosomes (solutions) to create new ones. There are five

phases considered in a genetic algorithm: Initialization of the population, Fitness function, and Genetic Operators (Selection, Crossover, and Mutation). Each population is made of a number of chromosomes (individuals). The fitness function is used to evaluate these individuals and to select the best parameters for the genetic operators. Early use of GAs has been reported in the field of image processing and filter design [K. Delibasis et. al., 1997] and 3D image analysis [K. Delibasis et. al., 1994], [K. Delibasis et. al., 1996], [Undrill, 1997]. Several advancements have been made to the method of GAs. In [Khalid Jebari et. al. 2013] parent selection operators are investigated in the context of the 0-1 Knapsack Problem (KP), a well-known combinatorial optimization problem that has been studied for many years. Many researchers [Khalid Jebari et. al., 2013] are comparing different parameters and methods to find the optimal combination for the best results.

Another population-based optimization technique is Differential Evolution (DE) that uses mutation and crossover operations to generate new solutions from existing ones. In detail, DE is an evolutionary algorithm that differs from standard genetic algorithms in several ways. Instead of relying on probability distributions for mutation, DE uses unit vectors and distance and directional information to create a mutation operator. This means that instead of crossover being applied before mutation, DE applies mutation before crossover. Additionally, DE does not use a probability distribution to create the mutation operator, as standard genetic algorithms do, but instead uses unit vectors. This makes DE a unique algorithm that is usually more efficient than standard genetic algorithms.

[David J. Montana and Lawrence Davis] have used a number of statistical tests to measure the accuracy of the parameters and the GA performance, such as parameter determines with what probability each individual is chosen as a parent (“Parent Scalar”); parameters that select a number of the weakest individuals and performs mutation to them in order to get fitter, and they use two different subsets of weights in their network.

Convolutional Neural Networks in the field of Image classification have shown remarkable success in recent years. Automating the design of CNNs is required to help some users having limited domain knowledge to fine-tune the architecture for achieving desired performance and accuracy. The usage of different evolutionary methods such as Genetic Algorithms helps in simplifying and automating the architecture of CNNs and improving their performance. [Yanan Sun et. al., 2020] propose an automatic CNN architecture design method by using genetic algorithms. This method is advantageous in that it does not require users to have domain knowledge of CNNs. Instead, it can still provide a promising CNN architecture for given images. To validate the proposed algorithm, it is tested on widely used benchmark image classification datasets. It is compared to the state-of-the-art peer competitors, which include eight manually designed CNNs, seven automatic and manual tuning, and five automatic CNN architecture design algorithms. The experimental results show that the proposed algorithm outperforms the existing automatic CNN architecture design algorithms in terms of classification accuracy, parameter numbers, and computational resources. Furthermore, the proposed algorithm is comparable to the best one from manually designed

and automatic and manually tuning CNNs in terms of classification accuracy, while consuming much less computational resources.

[Hanxiao Liu, 2018] provides evidence that evolutionary algorithms can be used to discover high-performance architectures (such as gradient descent algorithms), and that they can be employed effectively in a variety of tasks and domains.

In [Fei Yin et. al., 2011] the GA is used to search for the optimal architecture from a predefined range of architectures. Firstly, the initial population of the GA is generated using a random selection from the predefined range of architectures. Subsequently, the GA will produce a few generations to optimize the architecture of Deep Neural Network (DNN) sub-models. The fitness of each architecture is measured based on the predictive performance of the DNN sub-models over the corresponding group of datasets. Lastly, the best architecture will be selected as the optimal architecture for each group of datasets. The training error of each DNN model is obtained and treated as the fitness value of the objective function in GA optimization. Such procedure is repeated until the preset optimization criteria is met.

[Junxi Zhang, Shiru Qu, 2021] explore the optimization of the adaptive genetic algorithm (AGA) in the backpropagation (BP) neural network (BPNN), applied to shallow Multi-Layer Perceptron (MLP) model, in the context of traffic flow prediction. The corresponding optimized BPNN includes 18 inputs and 1 output. A single hidden layer is adopted for BPNN in this study. The AGA is performed for 200 generations. The results of this study show that the optimized adaptive GA in the BP neural network (OAGA-BPNN) algorithm has better optimization performance than the AGA-BPNN algorithm. The OAGA-BPNN algorithm uses the crossover rate and mutation rate to optimize the parameters of the GA algorithm so that the algorithm can be better used for solving problems. The optimized algorithm has a better optimization effect and shorter calculation time. The average optimization time of the OAGA-BPNN algorithm is significantly reduced, and the average error of the optimized result is also reduced.

[Er. Jasmeen Gill, 2010] use GAs to train a shallow MLP with one hidden layer to predict weather parameters. They use populations up to 90 individuals evolving for typically 100 generations. The GAs and BP are compared in terms of accuracy and efficiency. The performance of the models is evaluated on a real-time weather forecasting dataset. The results indicate that the NN model based on GA has better accuracy and efficiency compared to the BPNN model based on gradient descent.

[Han-Xiong Huang, 2014] tried to improve the classic BP algorithm by initially training using fewer epochs and training using fewer learning samples (since data collection is costly and time-consuming in their application domain: plastic object design). Then the trained BPNN model is passed into a GA that searches in the feasible region to optimize the model. The trained BPNN model is used as the fitness function of the GA. This process is iterated until a satisfactory solution is found. The results show that the approach is effective, and the prediction accuracy of the BPNN model is improved after the optimization.

[Parsa Esfahanian et. al., 2019] proposes the use of GAs for CNN training. They introduced a new encoding scheme for the weights of the CNN, which achieves better accuracy and faster convergence than the traditional encoding. In addition, this approach was found to be more robust to the choice of hyperparameters. However, the reported accuracy of the constructed CNN was below 0.45 in the MNIST and CIFR-10 dataset.

[Lingxi Xie et. al., 2017] discusses the possibility of automatically learning deep network structures. We note that due to the exponentially increasing number of possible network structures with the number of layers, a genetic algorithm is employed to efficiently traverse this large search space. We first propose an encoding method to represent each network structure in a fixed-length binary string, which is then used to initialize the genetic algorithm. In each generation, standard genetic operations are applied to the population of individuals, such as selection, mutation, and crossover, in order to eliminate weak individuals and generate more competitive ones. The competitiveness of each individual is identified through its recognition accuracy, which is obtained by training the network from scratch and evaluating it on a validation set. We also propose an approach to speed up the training process by utilizing the information from previous generations. Experimental results demonstrate that our automated deep network design approach is able to outperform handcrafted networks in both classification and regression tasks.

[Felipe Petroski Such et. al., 2018] tests the performance of a simple GA on hard deep reinforcement learning (RL) benchmarks. This work demonstrates that a simple, gradient-free, population-based genetic algorithm can scale to very large neural networks and solve challenging deep reinforcement learning problems. It also shows that by combining DNNs with neuroevolutionary methods such as novelty search, performance can be improved on deceptive or sparse reward tasks. Finally, it shows that the Deep GA is faster than popular gradient-based algorithms, and enables a state-of-the-art, up to 10,000-fold compact encoding technique.

[Amin Dastanpour (2016)] proposes a model that consists of three layers of ANN, i.e., the input layer, the hidden layer, and the output layer. The input layer consists of 41 parameters which are extracted from the KDD CPU 99 dataset. The hidden layer consists of 15 neurons using the sigmoid activation function and the output layer consists of 2 neurons using the softmax activation function. The weights are optimized using the genetic algorithm (GA). The GA algorithm is used to optimize the weights of the ANN in order to improve the accuracy of the model. The experimental results show that the proposed model achieves better accuracy and detection rate compared to the ANN model without GA optimization. The proposed model also has a better false alarm rate compared to the ANN model. The results show that the proposed model is effective for intrusion detection.

2.4 Low-rank matrix and tensor analysis

Low-rank matrix and tensor analysis have applications in many areas, such as machine learning, image processing, natural language processing, and data mining. This subject is a branch of mathematics that deals with the analysis and decomposition of low-rank matrices and tensors. The goal of the analysis is to find an efficient representation of the underlying structure of a matrix or tensor. For instance, low-rank matrix and tensor decompositions have been used to extract features from images and audio signals, analyze text documents and detect patterns in large datasets.

The Low-Multilinear Rank Approximation (LMRA) is a form of matrix factorization that aims to approximate a given matrix by a low-rank matrix. This is done by decomposing the matrix into a product of two or more lower-rank matrices, each of which has fewer columns or rows than the original matrix. By doing this, the total number of elements in the matrix is reduced, resulting in a more efficient representation of the matrix and a reduced computational complexity. The goal is to minimize the reconstruction error, i.e. the difference between the original matrix and the reconstructed matrix. This can be achieved by using different algorithms such as Singular Value Decomposition (SVD), and Tensor Decomposition. In the proposed algorithm, we investigated a higher-order extension of SVD, called “Tucker Decomposition” (Fig. 4b).

SVD is graphically described in Fig. 4a. This method works by decomposing a matrix X as a product of table A (with orthogonal columns), a diagonal matrix Σ and B, the transpose of another matrix with orthogonal columns.

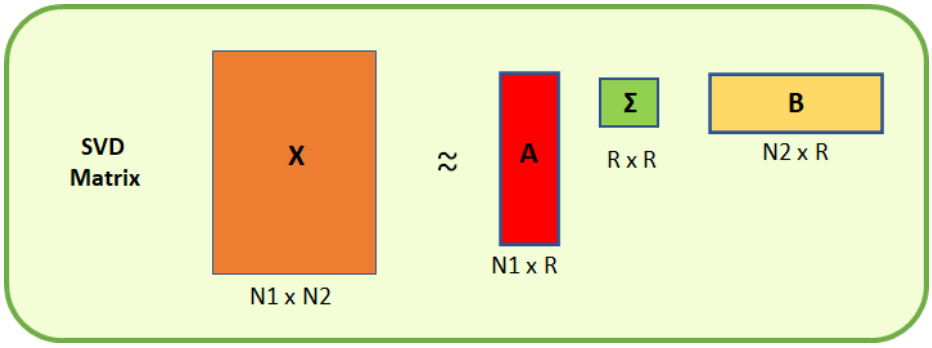


Figure 4a: SVD is a technique used for the reduction or compression of data

On the other hand, Tucker decomposition is essentially a higher-order extension of SVD, where we find the best rank approximation of it (Fig. 4b). Practically, we can decompose a 3-way tensor into:

- Core Tensor (G): dense tensor of interactions between factor matrices,
- Factor matrices (A, B, C): each one of them represents a different core scaling along each mode.

More details are described by Tamara G. Kolda et. al. (2009), Laurent Sorber et. al. (2014), Alex P. da Silva et. al., and in section 3.5.2.2.

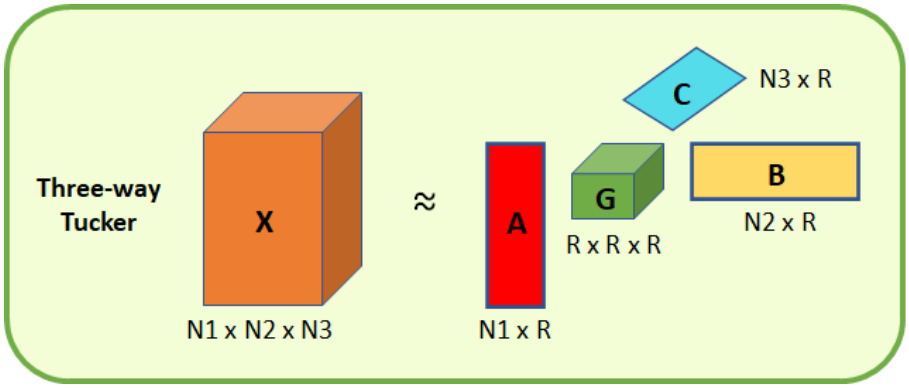


Figure 4b: Tucker Decomposition for a 3-way tucker

3. Methodology

3.1 The available dataset

Here, we implement the neural network using $MNIST$ database. It contains 70,000 images, from which 60,000 are often used for training and 10,000 for validation. Each of these images is a 28×28 pixel binary image, as seen in Fig. 5. However, training a neural network using 60,000 images is time-consuming. In our case, we describe the process using 6,000 images, where 5,000 images are used for training and 1,000 for validation.

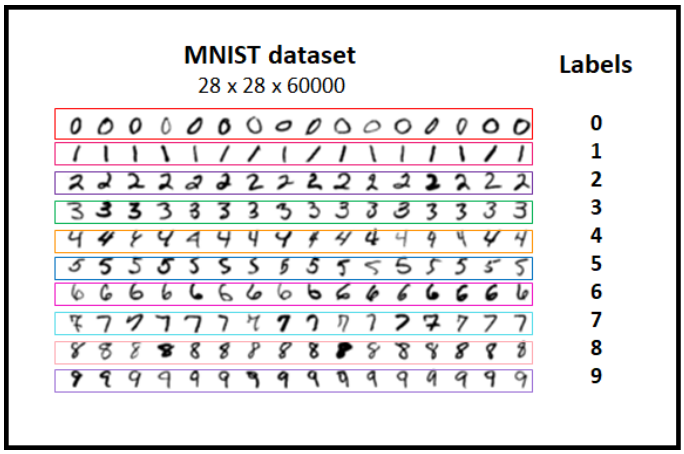


Figure 5: Samples of MNIST and Fashion MNIST dataset

3.2 Convolutional Neural Network architectures and Parameterization of Basic CNN functions

We used a CNN containing one Convolutional layer, ReLU and Softmax activation functions, and one Mean Pooling layer. Our CNN's architecture consists of an input layer, one hidden layer, and the output layer. The input layer has $784\ neurons$, including $784\ input\ nodes\ (28{\times}28)$ representing the pixel count of the input image, and $0\ bias\ neurons$. We use a single convolution layer with $20,\ (9{\times}9)$ filters. Each output of the convolution filter (20×20×20) will be passed through the ReLU and Pool function; we use 2 submatrices for Pooling Layer. That was the Feature Learning Section of our CNN. The outcome of this section is a 10×10×20 matrix which is flattened and passed into the Classification Section of our neural network. In detail, the flattened matrix passes through a ReLU function and then through the 10×1 output layer of our CNN, with Softmax as the activation function which leads to a matrix corresponding to the probabilities of each class of the dataset.

3.2.1 Weight Dimensions

In $Fig.\ 6$, the implemented architecture of the neural network is shown. There are many layers, but only three of them contain weight matrices. Hence, these three layers require training.

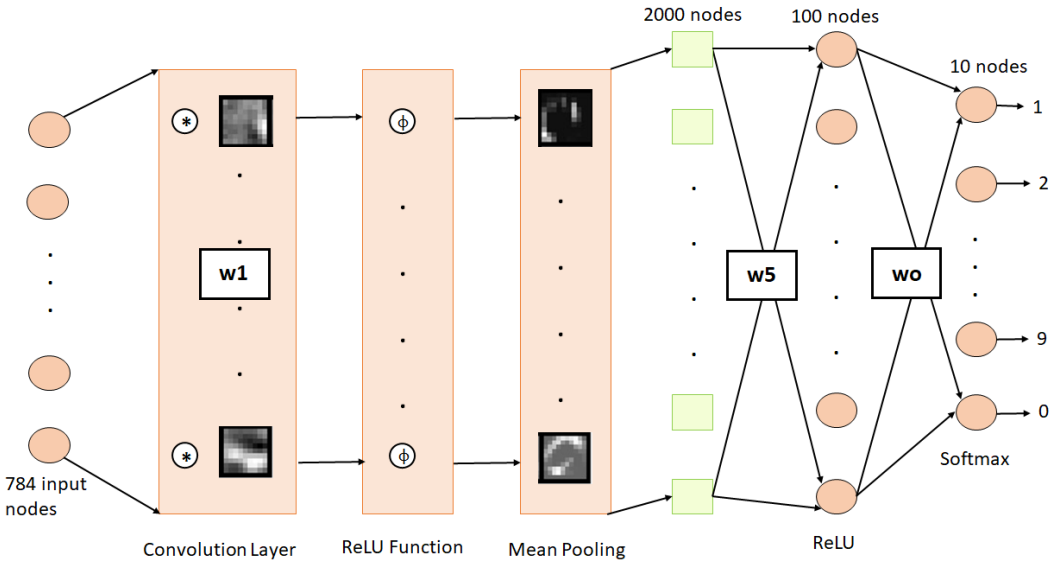


Figure 6: The architecture of our CNN

As seen above, $W1$ is the weight of the convolutional layer: it is used by the convolution filters for image processing· $W5$ and Wo contain the connection weights of the classification layers. Note that, in the first fully-connected (FC) layer, there are 2000 green-squared nodes, which represent the layer that transforms a $2D$ image into a vector (flattening) and does not participate in the training process. According to that, $W1$ represents the $20,\ (9\times9)$ filters and has length equal to $9 \times 9 \times 20$ -1620- weights· W5 connects the flattened outcome of the Feature learning section with the ReLU activation function of the Classification section and so its length is equal to 100×2000. Finally, Wo is the weight matrix that connects the output layer (10 neurons) to the previous FC layer. Hence, the dimension of Wo is 10×100.

3.2.2 Activation Functions

The layer between the convolution filter and feature map is the Activation function. In CNN, a commonly used activation function is Rectified Linear Unit, in short, ReLU function. The ReLU function is used to calculate the activation values in our CNN. Mathematically, it is expressed as

$$ReLU(z) = \begin{cases} 0, \text{ for z=0,} \\ z, \text{ for } z > 0 \end{cases} \quad (1)$$

where z : the input value of each neuron and $ReLU(z)$: is the feature map of input images.

In summary, ReLU is a nonlinear function -or piecewise linear function- that will output the input directly if it is positive, otherwise, it will output zero, as seen on $Fig.\ 7$.

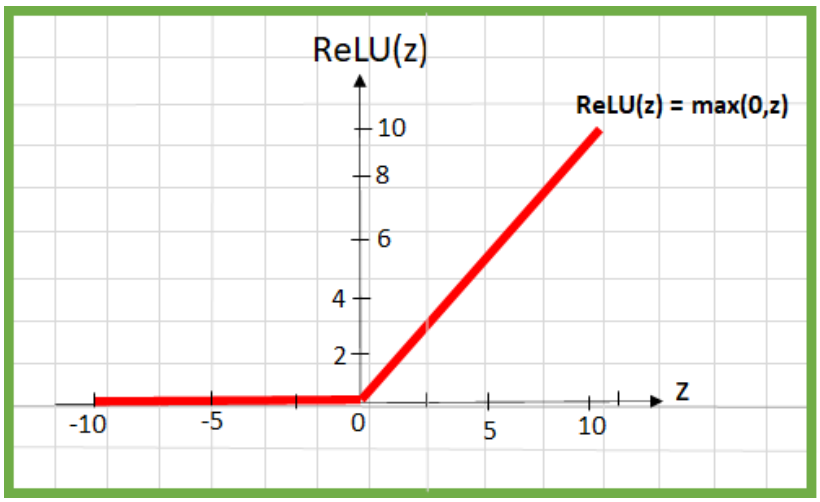


Figure 7: ReLU is used in the hidden layer to avoid the vanishing gradient problem and better computation performance.

Note that the activation function is an integral part of a neural network. Without an activation function, a neural network is a simple linear regression model. This means the activation function gives non-linearity to the neural network.

The Softmax function is an activation function before the output layer and returns the probability of each class. It can be mathematically expressed as

$$Softmax(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \quad (2)$$

where z_i: the values from the neurons of the output layer, k: number of classes on the output layer (here, $k = 10$), e^{z_i}: the standard exponential function for every input of softmax function (z_i). The nominator is positive, with low value for negative and great value for positive z_i, but it is still not fixed in the range [0, 1]. The normalization term $\sum_{j=1}^{k} e^{z_j}$ guarantees that the Softmax output values will all add up to 1 and fall within the range (0, 1), forming a valid probability distribution.

3.3 CNN training using Backpropagation (BP)

A simple version of BP for a CNN with only one convolutional layer and two fully connected (FC) layers has been implemented in this work, based on [Phil Kim, 2017]. Note that we define $\alpha = 0.01$ as the learning rate of training, which determines the speed at which learning takes place. The momentum coefficient is set $\beta = 0.95$. Using the mini-batch method, the mini-batch size was set as $batch = 100$, we select a subset of data with which the neural network is trained on these selected data. More mathematical quotations about BP are described by [Phil Kim, 2017].

We have not implemented early stopping, instead we allow the BP to evolve for a number of N epochs and study the evolution of train accuracy and test accuracy.

3.4 Genetic Operators

Genetic operators (also known as genetic variation operators) are functions used in genetic algorithms to manipulate the chromosomes of the population in the evolutionary process. They are used to produce new and different chromosomes from the existing ones in the population. Common genetic operators include Evaluation of chromosomes, Selection, Crossover, and Mutation.

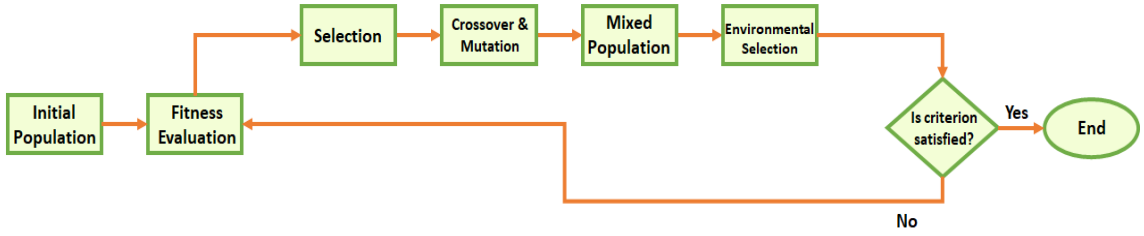


Figure 8: The flowchart of Genetic Algorithm

3.4.1 Calculation of chromosomes' fitness

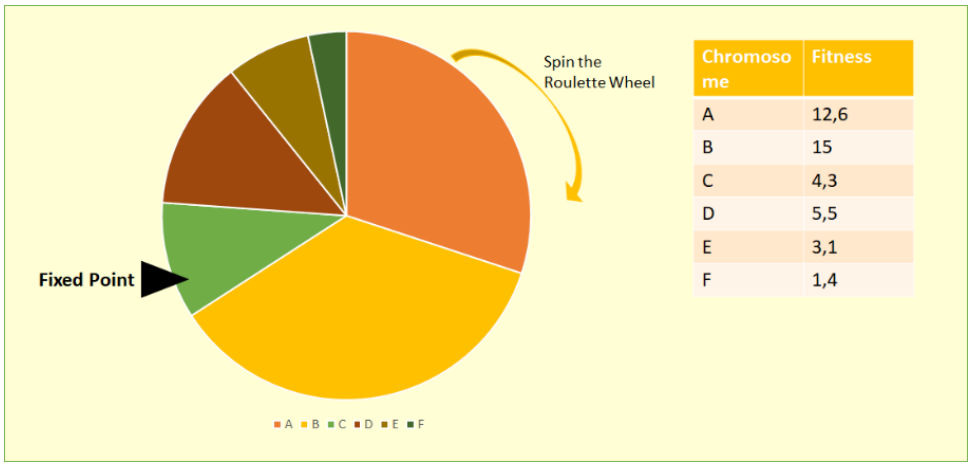
The fitness function is the most computational but indispensable element of GAs. It is a measure of the quality of a solution. Essentially, GA follows the fitness function in its search for the optimal solution. The fitness function helps to decide which individuals can progress to the subsequent generation of solutions. Each GA operator is designed to increase the fitness of the population. Thus, assigning fitness to each chromosome (individual) is of paramount importance for GAs. In the case of CNN training using GAs, the fitness function is defined in subsection $3.5.2$.

3.4.2 Selection Operator

After calculating the fitness of every chromosome in the population, selection operator (or parent selection) is used to determine which of the individuals in the population will be chosen to reproduce and create the offspring that form the next generations.

GA is based on Darwin's theory of "Survival of the fittest ". Our goal in this operator is to pick the fittest chromosomes to pass their good (fit) genes to the next generation and so, evolve a fitter population after each iteration. The probability of choosing an individual depends directly on their fitness evaluation: fitter individuals have a higher chance of being chosen and propagating their features to the next generation.

The Roulette Wheel Selection is used as the parent selection mechanism. The wheel is divided into n_C portions, where n_C is the number of individuals in the population; each portion of the wheel represents the fitness value of each chromosome. A fixed point ξ is chosen randomly on the wheel and we spin the roulette wheel 2 times to get our couple of parents. The portion that comes in front of this point is chosen as the parent. This fixed point is a random number between 0 and the sum of fitness for the whole population. More formally, let f_i be the fitness of individual i, with $i = 1, 2, \ldots, n_C$. Then an individual is selected with a probability according to its fitness as follows: a random number $\xi \sim \cup(0, \sum_{j=1}^{n_c} f_i\ )$ is generated. For a given value of ξ, the individual k is selected such that $\xi \leq \sum_{j=1}^{k} f_j\ (3)$ and $\xi > \sum_{j=1}^{k+1} f_j\ (4)$. The concept of selection is graphically depicted in $Fig.\ 9$.



Chromosome	Fitness
A	12,6
B	15
C	4,3
D	5,5
E	3,1
F	1,4

Figure 9: Roulette Wheel Selection

As seen above, the fitter an individual is, the bigger portion they have on the wheel, and they are more likely to land in front of the fixed point when the wheel is rotated.

3.4.3 Crossover

Crossover is a genetic operator used to vary the genotype of chromosomes from one generation to the next. In other words, the crossover is the reproduction in our population. Two chromosomes ("parents") are selected using the roulette wheel, from the mating pool to crossover and produce offspring.

Here, we use Uniform Crossover. This operator takes two parents as input and creates two children by randomly selecting genes from either parent. Each gene has an equal probability of being selected from either parent. This allows for a more balanced combination of genetic material from the two parents. A random variable generated according to the Uniform distribution $U(0,1)$, called $alpha$, has the same length as parents do, and shows us which parts of each parent will be passed to the offspring. Mathematically:

$$child_1 = alpha \times \text{parent}_1 + (1 - \text{alpha}) \times parent_2 \quad (5)$$

$$child_2 = alpha \times \text{parent}_2 + (1 - \text{alpha}) \times parent_1 \quad (6)$$

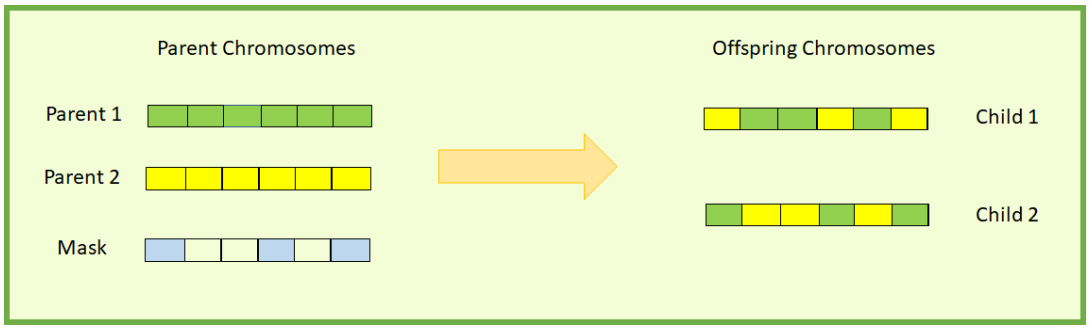


Figure 10: Uniform Crossover where "Mask" illustrates the variable alpha

3.4.4 Mutation

Mutation Operation is defined as a random tweak in the genotype of chromosomes to get a new individual. Its purpose is to introduce and maintain diversity in the population and is usually applied with a low probability per gene (p_m), usually set to 0.001. For each gene, a random number is generated according to the Uniform distribution, $U(0,1)$ and if it exceeds p_m then the gene is mutated. In our proposed algorithm, we choose to apply mutation only to children. A gene with a value a is mutated to a new value $a \leftarrow a + \sigma\xi, \ \xi \sim \mathcal{N}(0,1) \quad (7)$.

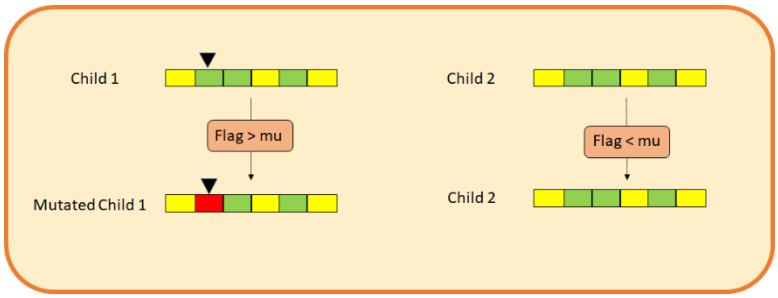


Figure 11: Mutation operator

3.5 Genetic Algorithm implementation for CNN training

Using a genetic algorithm to optimize a convolutional neural network (CNN) involves creating a fitness function that measures the performance of the CNN and then applying genetic operators (selection, crossover, and mutation) to the fitness function to optimize the network. Hence, fitness function should measure the accuracy of the network on a dataset, and the genetic operators should be applied to the weights and parameters of the network in order to optimize them. The goal of the optimization process is to find the set of weights for given GA settings (number of iterations/populations/genes, mutation percentage, etc.) that maximize the accuracy of the CNN on the dataset. After the optimization process is complete, the resulting CNN should be able to generalize better to unseen data (test set) and achieve higher accuracy overall.

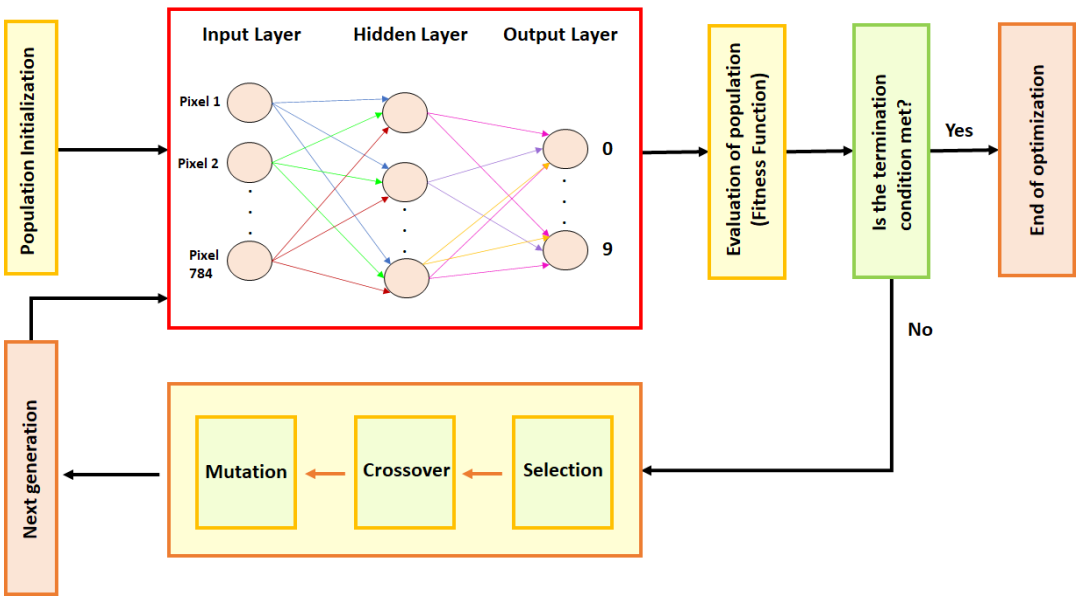


Figure 12: Genetic CNN

3.5.1 CNN representation and Population Initialization

The CNN we implemented in the proposed work is described in 3.2.

Population Initialization

This process begins with a set of individuals (chromosomes) which is called a population. Each chromosome is a solution to the problem we want to solve and is practically characterized by a set of variables known as genes. So, each chromosome encodes the CNN. Therefore, when it comes to training a CNN using GAs, chromosomes represent the weights of our CNN. The weights are initialized randomly as follows:

$w_1 \sim \cup(-2,+2)$, $w_5 \sim \cup(-0.05,+0.05)$ and $w_o \sim \cup(-0.2,+0.2)$.

In the case of low-rank convolutional tensor, the relevant initialization is modified:

$w_1 \sim \mathcal{N}\ (0,1)$, where $\cup$ and $\mathcal{N}$ stand for the uniform and normal distribution.

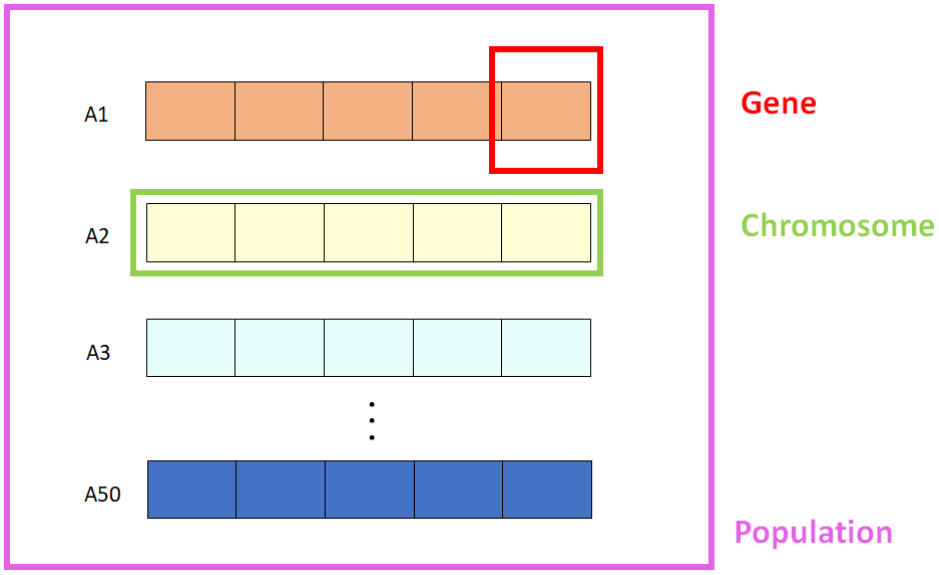


Figure 13: Genetic Algorithm terms

In Genetic Algorithm, the population size is an important parameter that directly influences the ability to search for an optimum solution in the search space. It is generally known that having a large population leads to the accuracy of getting an optimal solution. Here, we set the population size n_C equal to 50.

The following process is employed to generate the 1st generation: n_C individuals are generated, and their fitness is evaluated. The fittest 20% individuals are saved, and the rest

are discarded. The above steps are repeated 5 times, thus a total of n_C individuals are saved that are expected to have above random average fitness. These individuals serve as the 1st generation.

3.5.2 The structure of a Chromosome

3.5.2.1 Encoding Full-Rank Tensors

Here, the weights of all layers are structured as full-rank tensors that are initialized randomly, in order to form the chromosomes of each population.

We create uniformly distributed weight matrices that contain random numbers, where $W1$ has length equal to $9 \times 9 \times 20$, $W5's$ length is equal to 100×2000 and Wo has length equal to 10×100, as described in [$3.1.1$]. Then, these three weight matrices are flattened into one variable called the Chromosome or Individual of the population. According to the perspective of Full Rank matrices, the length of each chromosome is equal to

$$9 \times 9 \times 20 + 100 \times 2000 + 10 \times 100 = 202,620 \; genes \quad (8)$$

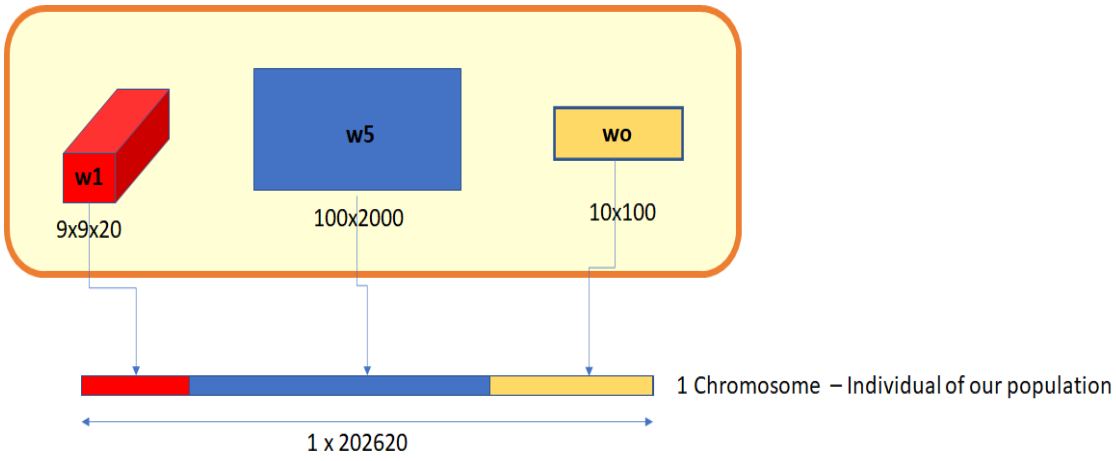


Figure 14: Structure of chromosome (Full Rank Version)

3.5.2.2 Encoding Low-rank Tensors

Here, as we know, matrix W1 captures the weights of convolution layers (size: $9 \times 9 \times 20$), but in this case, it is considered as a 3D Tensor that is decomposed to G (Core Tensor), A, B , and C (factor matrices). According to section 2.4, the sizes of these four matrices are shown below:

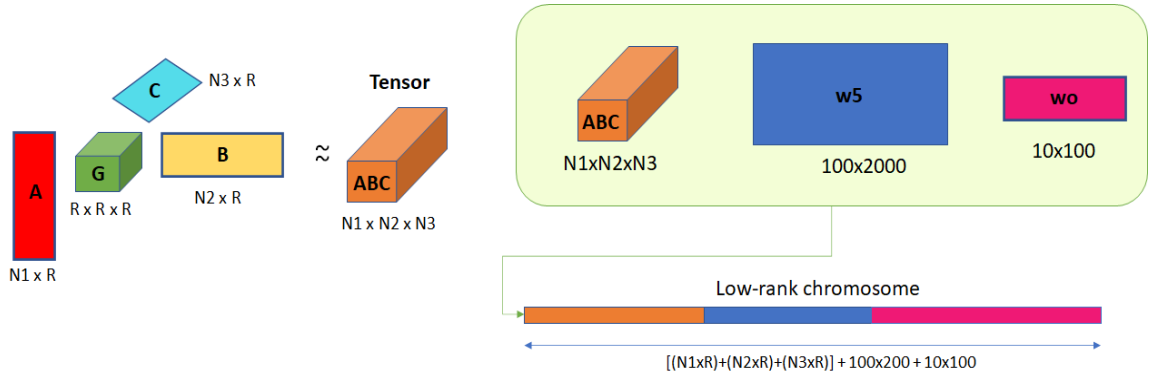


Figure 15: Chromosome initialization based on low-rank matrices

As seen in $Fig.\ 15$, W1 is a $3D$ tensor composed by $G = R{\times}\mathrm{R}\ {\times}R$, $A = 9{\times}\mathrm{R}$, $B = 9{\times}\mathrm{R}$ and $C = 20{\times}\mathrm{R}$, where R: the rank of W1 that we choose in every experiment. In the proposed algorithm, the core tensor is equal to 1 and so W1 is the outer product of our factor matrices. Eventually, the length of W1 is equal to $ABC = 9{\times}\mathrm{R} + 9{\times}R + 20{\times}\mathrm{R}\ \ (9)$. More details are described in Table 1.

Table 1: Total Genes in our CNN's weights for full-rank and low-rank

Rank	Total Genes (Size of chromosome)	Genes in convolutional Tensor (W1)	Genes in hidden layer (W5)	Genes in output layer (Wo)
5	201,190	190	100x2000	10x100
7	201,266	266	100x2000	10x100
11	201,418	418	100x2000	10x100
Full	202,620	1620	100x2000	10x100

3.5.3 Fitness Function

The most challenging yet essential concept of GAs is the fitness function. In the proposed work, fitness function is a measure of the fraction of correct image classifications for the training set that a given chromosome (individual) produces. The higher the value of the fitness function, the better the solution.

Initially, the available dataset of images is divided into training and testing subsets. It is important to mention that the training set is randomly shuffled so that the neural network will have a better generalization. The main steps for fitness calculation are as follows:

1. The chromosome is decoded and the corresponding CNN is constructed. The decoding is the inverse process of chromosome encoding that has been described in the subsection – above, both for the full-rank and the low-rank case.
2. The images of the training set are forward-passed into the constructed chromosome. The CNN's predictions are compared to the true labels of the images and the accuracy is calculated. The fitness is set equal to the accuracy.

The accuracy of the test subset is also calculated, solely for the purpose of measuring the CNN's ability to generalize (equivalently quantify the overfitting) and it is not used for the training process.

3.5.4 Parent Selection

The $Roulette - Wheel$ Selection is used as the parent selection mechanism, as described in paragraph $3.4.2$ - $"Selection"$.

3.5.5 Crossover, Mutation & Termination Condition

Here, we use Uniform Crossover. According to $3.4.3$, each gene of each offspring is a random linear combination of the corresponding genes of the two parents. This way, the resulting chromosomes will be more diverse and may contain better solutions than the original parents.

The mutation operator was implemented as follows. First, the genes of each chromosome that will be mutated are randomly selected. As referred in subsection $3.4.4$, the mutation is performed on each gene of the offsprings with a probability of $p_m = 0.001$.

The **termination condition** is the number of generations N_{iter} that is limited to 400, after which the parameters of the fittest individual will be used to build the final CNN.

Creation of new generation

The crossover between two good chromosomes (solutions) may not always yield a fitter solution. However, since parents are selected according to fitness, the probability of the child being fit is high. Once the offspring are created, they are joined with the existing individuals (parents) in the population, and then altogether get sorted in descending order by their fitness. Note that our population now consists of both parents and children. Our purpose is

to keep the fittest members and maintain the size of our population stable. Hence, the low-fitness chromosomes will be removed from the population at the end of each iteration and the remaining n_C chromosomes will form the new generation.

3.5.6 GA Hybridization

It is common practice to hybridize a global optimization method with a local one. The main strategy of this approach is that every few generations a number of the fitter individuals undergo local optimization and then are inserted back in the population. In detail, the BP is performed every X generations and fine-tunes the weights of the fittest Chromosome of the Xth generation; BP is allowed to repeat for a number of epochs using the training set. Once the best set of weights is found using BP, the fine-tuned chromosome is encoded and replaces the corresponding chromosome in the population of the current generation.

4. Results and Discussion

GA settings

For GA, we initialize a population containing 50 chromosomes (individuals). For the initial generation, we generate 5 populations in order to create the gene pool. Gene pool contains the 10 fittest chromosomes of these 5 populations. The Roulette-Wheel Selection during its first iteration (3.5.1). GA evolves for maximum number of $400\ iterations\ (generations)$, performing Evaluation (Fitness Function), Selection, Crossover, and Mutation, and at the end of each generation, our chromosomes are sorted according to their fitness value and moved to the next generation, while the chromosome with the highest fitness (accuracy) is saved to perform GA hybridization in $Experiment\ 4$. Furthermore, it is important to mention that Crossover is always performed to its input individuals, while the performance of Mutation depends on p_m. For more GA settings, see 3.5 and $Table\ 2$.

Table 2: Genetic settings

GA parameters	Value
Chromosomes (population size)	50
Genes (size of individual)	202,620 (Full-Rank), 201,190 (R=5), 201,266 (R=7), 201,418 (R=11)
Maximum iterations	400
Selection Type	Roulette Wheel
Crossover Type	Uniform
Mutation probability	0.001
Sigma (Mutation)	1

Training Images	[500, 1000, 2500, 5000]
Test Images	the last 1000 of 60,000

Using the above settings, we implemented four experiments.

1. Train CNN with GA using the training subsets, globally for all individuals of the population.

When training with GA, the parameters can be adjusted globally for all individuals of the population, which can further improve accuracy. In this experiment we investigated the effect of the size of the training set. Additionally, we executed the BP on the best individual of the last generation of the GAs. In $Table\ 3$, we can see that our experiment worked pretty well resulting in satisfactory validation accuracy with a mean percentage of 72%. The results of this experiment are shown below in figures $16a-\mathrm{d}$.

Table 3: Achieved accuracy of the GA-CNN for different size of the training set.

Dataset			Tensor rank	BP during GAs	Accuracy GA %		Accuracy % BP after GA	
Train	test	train / chrom.			Train	Test	Train	Test
1-5000	last 1000	all	full	Never	69.54	71.1	100	96.8
1-500	last 1000	all	full	Never	76	67.4	100	86.8
1-1000	last 1000	all	full	Never	73.10	75.6	100	93.1
1-2500	last 1000	all	full	Never	72.04	74	100	96.2

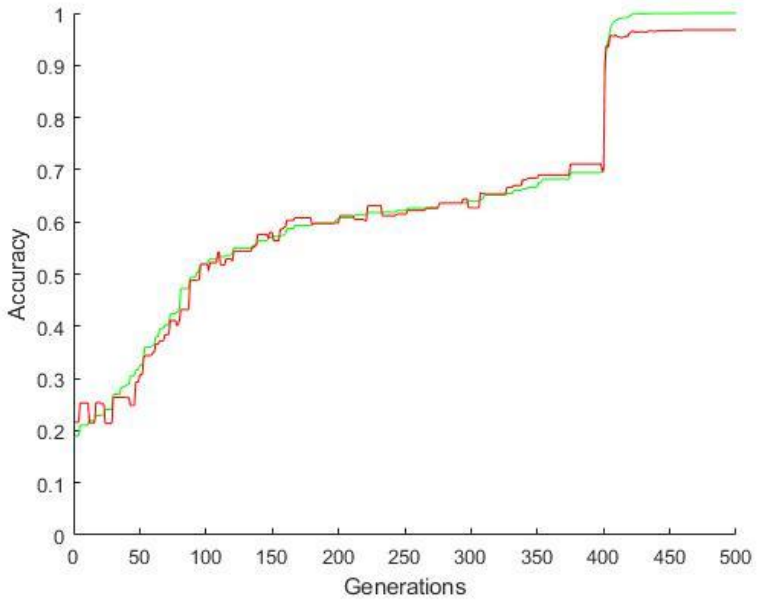


Figure 16a: Accuracy of our GA-CNN with 5000 training images. After 400 generations we performed BP to our model to upgrade the accuracy

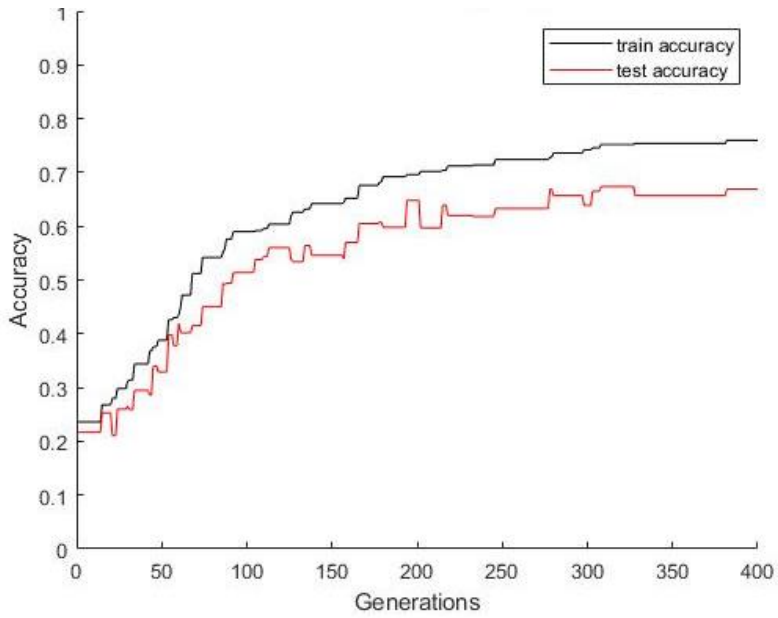


Figure 16b1: Accuracy of the proposed GA-CNN with 500 training images

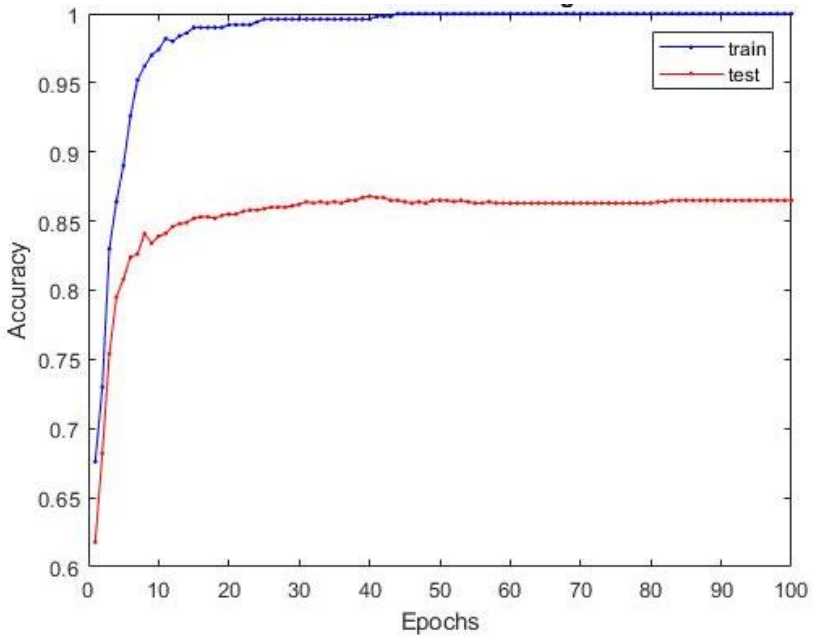


Figure 16b2: Accuracy of the proposed GA-CNN with 500 training images AFTER applying BP

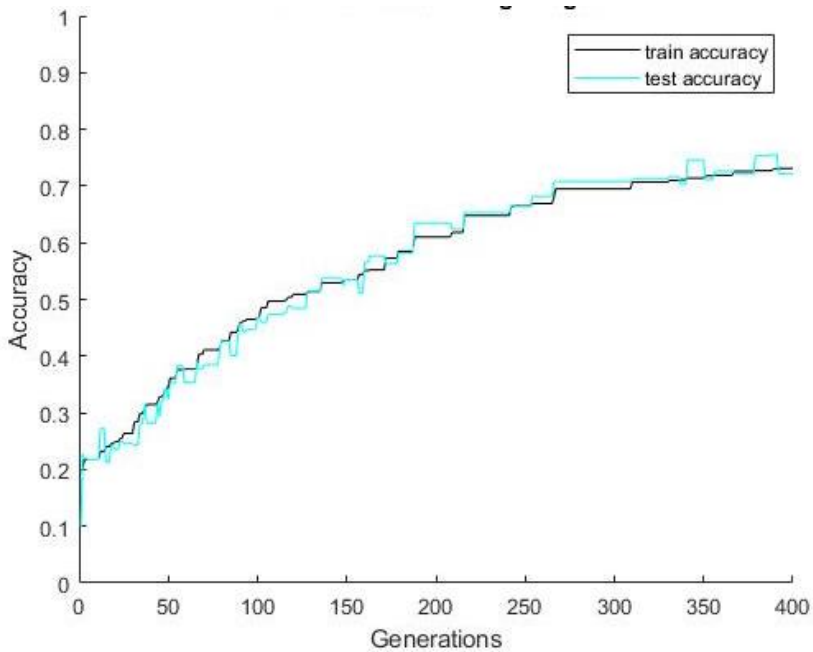


Figure 16c1: Accuracy of our GA-CNN with 1000 training images

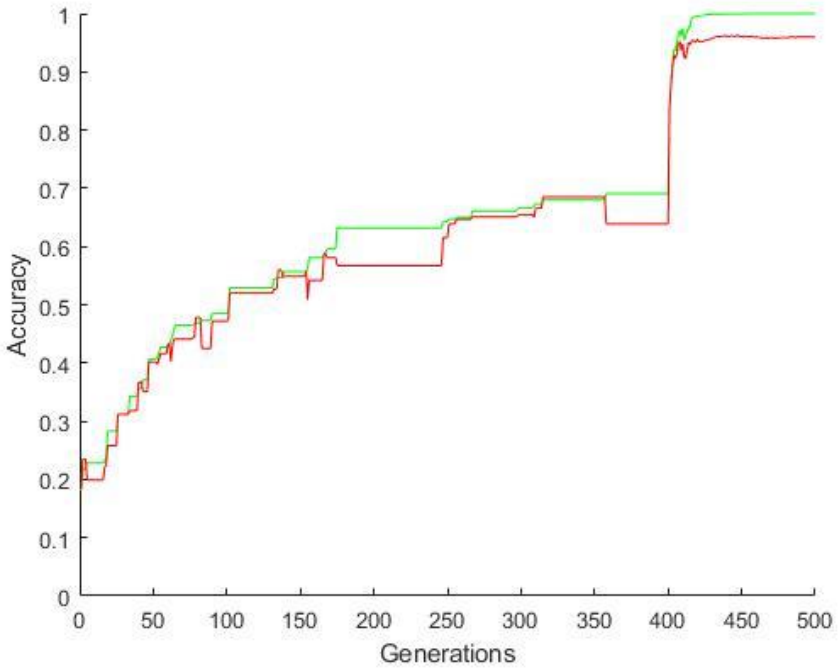


Figure 16c2: Accuracy of our GA-CNN with 1000 training images. After 400 generations we performed BP to our model to upgrade the accucacy

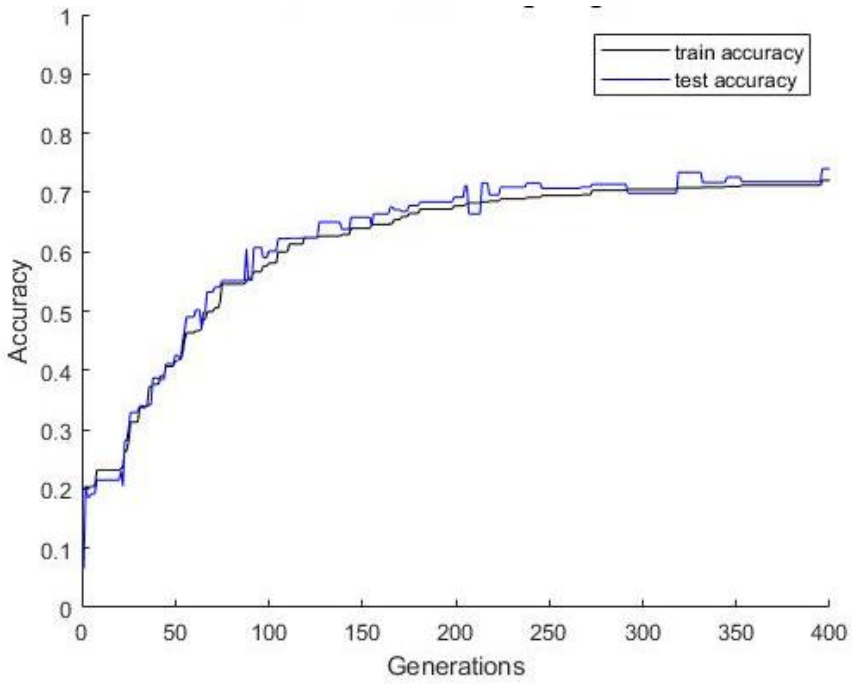


Figure 16d: Accuracy of our GA-CNN with 2500 training images

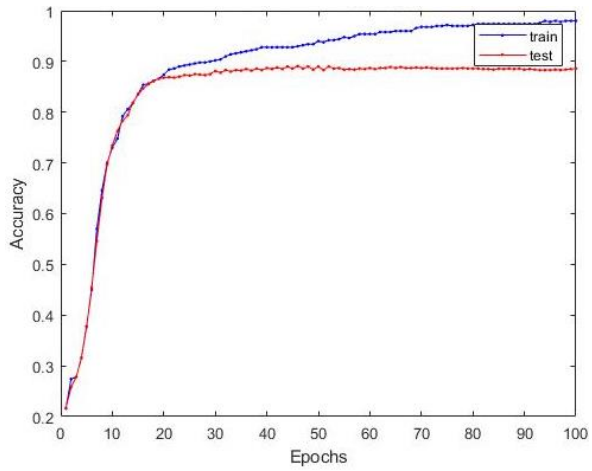
- Train our CNN with Backpropagation

In this chapter, we present the performance of our CNN when optimized with the traditional BP algorithm. The following figures ($17a - 17c$) work as our baseline in order to compare the efficiency of our proposed algorithm with the BPNN.

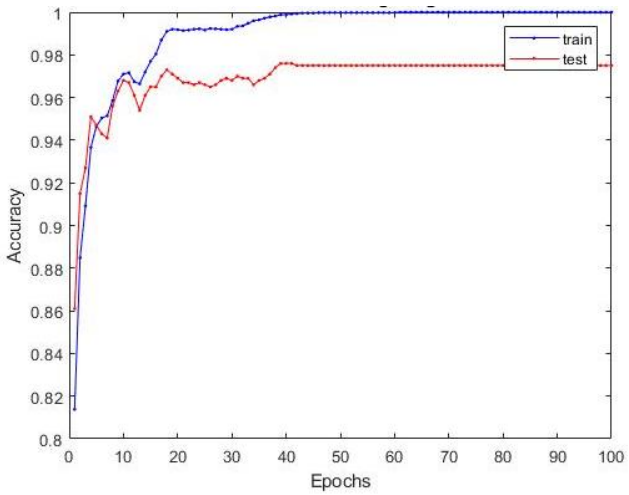
Table 4: CNN training and test accuracy of the traditional BP

Dataset			Tensor rank	Epochs	Accuracy %	
train	Test	train / chrom			Train	Test
1-500	last 1000	All	full	100	98	89.1
1-5000	last 1000	All	full	50	99.99	97.4
1-5000	last 1000	All	full	100	100	97.6

Backpropagation is a quick, easy and efficient algorithm to train CNNs. In this experiment, we tried to evaluate BP's performance by using two different sizes of training sets and by increasing the number of epochs. According to $Table\ 4$, BP worked very well: the weights of our CNN are well optimized, and our model becomes more accurate. Although, a large number of epochs can lead to fluctuations in backpropagation, as seen in Fig. 17b. Additionally, the longer the training period, the more likely it is that overfitting will occur, which can also cause fluctuations in backpropagation. However, in these experiments, not significant overfitting was observed, since the test accuracy did not deteriorate with the number of epochs.



(a): BP performed to 500 images with 100 epochs



(b): BP performed with 5000 training images and 100 epochs

Figure 17: Training and Test accuracy of BPNN

2. CNN training with GA using training subset for each individual.

Here, the training process of CNN with GA is the same as in $Experiment\ 1$, by using a random subset of the training set to calculate the fitness of each chromosome. The effect of the size of the random subset is investigated, using as few as 200 random images per fitness evaluation. During the training process, the same chromosome decoding process is applied to the updated chromosome to construct the updated CNN.

According to $Table\ 5$, reducing the size of the training subset can reduce the accuracy of the CNN model. We referred in 3.5.2 that the fitness function is the most challenging operation in GA-CNN, meaning that it is the most time-consuming/computationally expensive procedure since it evaluates the accuracy of our model. Hence, when we train our CNN with a smaller training set, the fitness function becomes quicker. But, in this case, our model may not be able to learn the features of the dataset well enough, resulting in a decrease in accuracy. As seen in $18a$, $18b$ and $18d$, the subset is reduced by $80 - 90\%$ where we can notice that the average validation accuracy is 57.66%, while in the other two cases where the percentage decrease of our dataset is smaller, the average validation accuracy is 67.35%. More details in $Table\ 5$.

Table 5: GA-CNN Settings for Experiment 2

Dataset			Tensor rank	BP during GAs	Accuracy	
Train	Test	Train/chrom.			Train	Test
1-5000	Last 1000	all	full	Never	69.54	71.1
1-2500	Last 1000	1000	full	Never	69.1	68.5
1-5000	Last 1000	200	full	Never	51	43.7
1-5000	Last 1000	500	full	Never	68.2	64.2
1-5000	Last 1000	1000	full	Never	68.9	65.1
1-5000	Last 1000	2500	full	Never	69.2	69.6

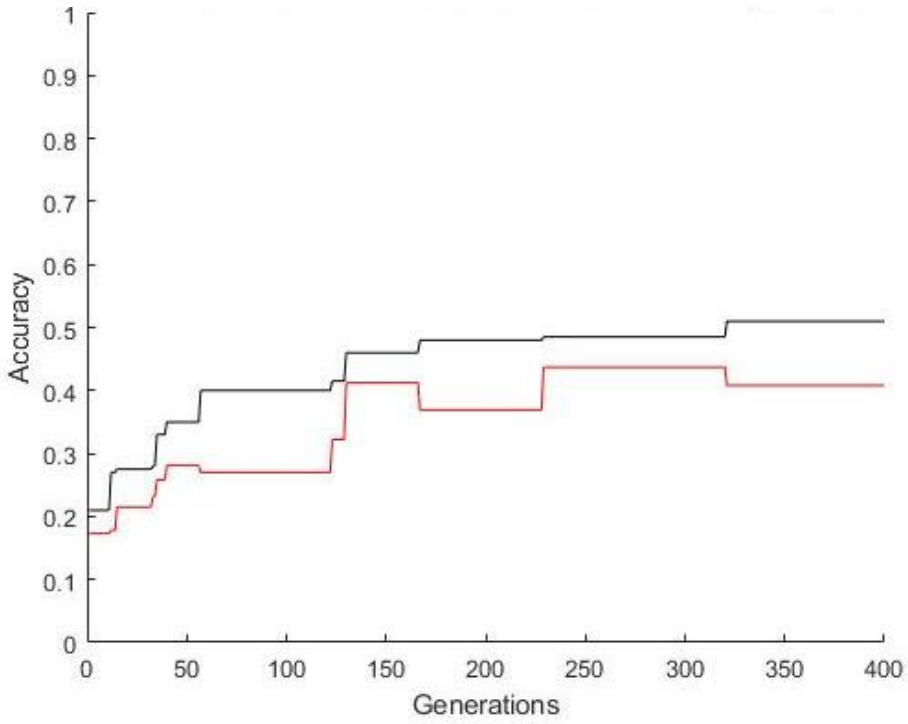


Figure 18a: Train and Test accuracy of our CNN optimized with 200 training subset (initial size: 5000)

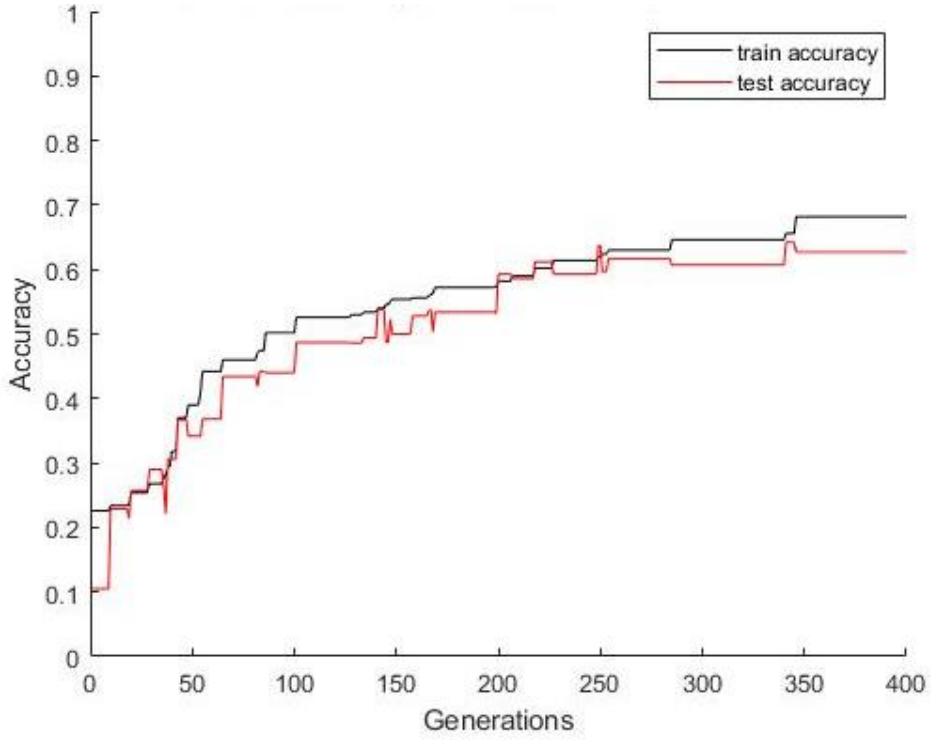


Figure 18b: Train and Test accuracy of our CNN optimized with 500 training subset (initial size: 5000)

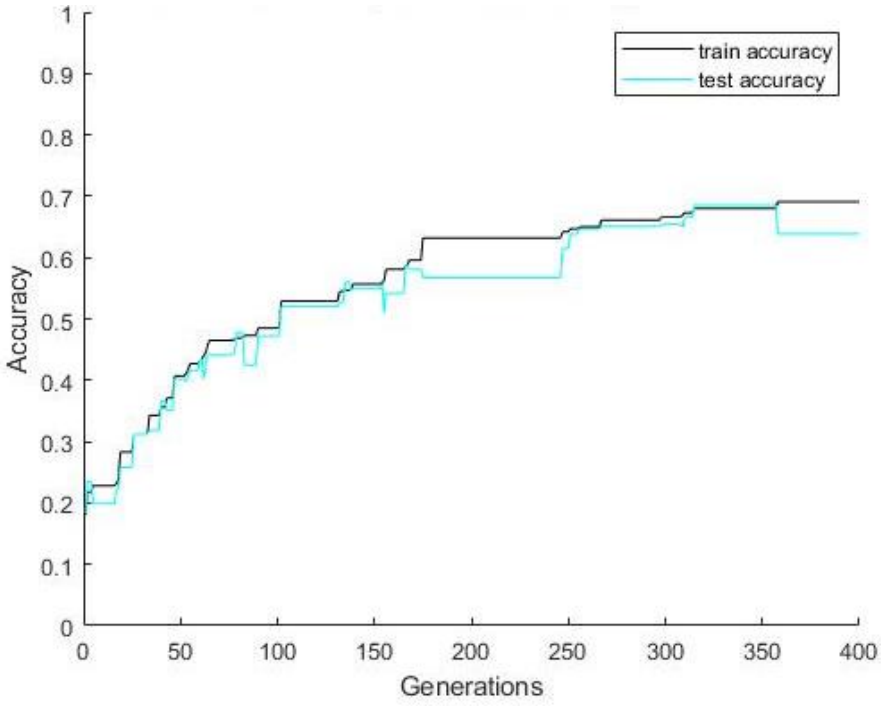


Figure 18c: Train and Test accuracy of our CNN optimized with 1000 training subset (initial size: 5000)

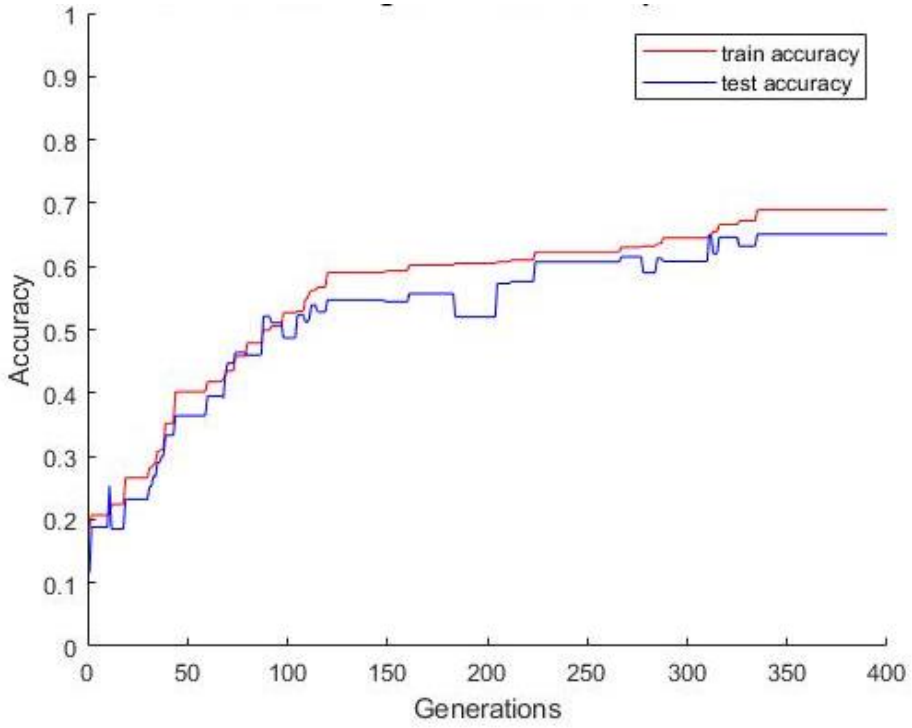


Figure 18d: Train and Test accuracy of our CNN optimized with 1000 training subset (initial size: 2500)

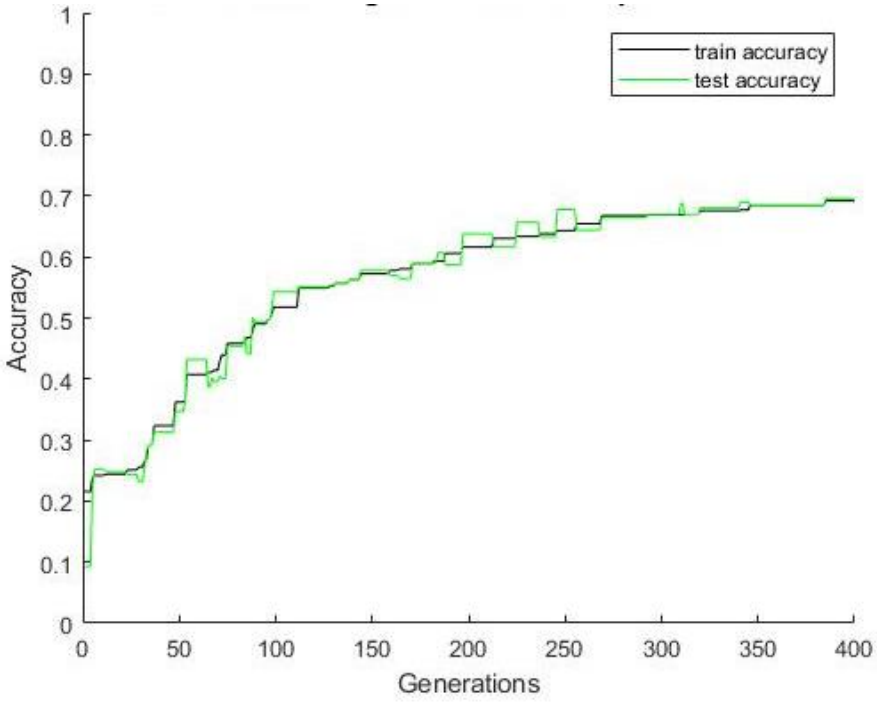


Figure 18e: Train and Test accuracy of our GA-CNN optimized with 2500 training subset (initial size: 5000)

3. Apply low-rank weights to the convolutional layer and then optimize our CNN only with GA.

Here, we tried to investigate the influence of low-rank matrices on our GA-CNN model. The training process is similar to the process described above ($Experiment1$), with the only difference being that some parts of our chromosomes are encoded according to $3.5.1.2$. What is remarkable in this experiment is the fact that our GA-CNN is optimized more effectively when it has 190/266/418 weights in the convolutional layer compared to when it has 1690 weights. This is due to the fact that having a lower number of weights allows the model to reduce the complexity of the model, thus making it easier to optimize and achieve better performance. Hence, low-rank weights can reduce the number of parameters in the network, resulting in faster training and improved model performance. Additionally, low-rank weights can help reduce overfitting, as they encourage the model to focus on important features and ignore irrelevant ones. In conclusion, the accuracy of the model can be improved by applying low-rank weights to the convolutional layer, as seen in Table 6.

According to 2.4 and $3.5.1.2$, the accuracy of the model can be improved by applying low-rank weights to the convolutional layer. Low-rank weights can reduce the number of parameters in the network, resulting in faster training and improved model performance. Additionally, low-rank weights can help reduce overfitting, as they encourage the model to focus on important features and ignore irrelevant ones. To apply low-rank weights to the convolutional layer, the genetic algorithm can be used to search for the best combination of weights that results in the highest accuracy. Once the optimal weights are found, they can be applied to the convolutional layer and the accuracy of the model can be evaluated.

Table 6: Low-rank GA-CNN training results

Dataset			Tensor rank	BP	Genes in convolutional Tensor (W1)	Accuracy %	
Train	test	train/ chrom				Train	Test
1-5000	last 1000	all	Full	never	190	69.54	71.1
1-5000	last 1000	all	7	never	266	70.90	73.60
1-5000	last 1000	all	5	never	418	72.32	73.70
1-5000	last 1000	all	11	never	1620	57.20	73.70

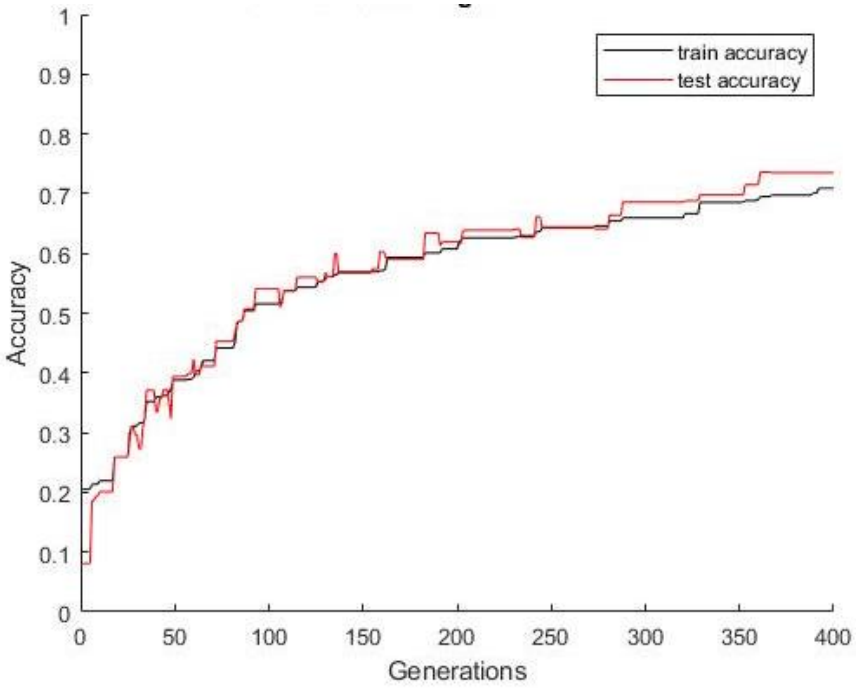


Figure 19a: Training and Validation accuracy of Low-rank GA-CNN (R=7)

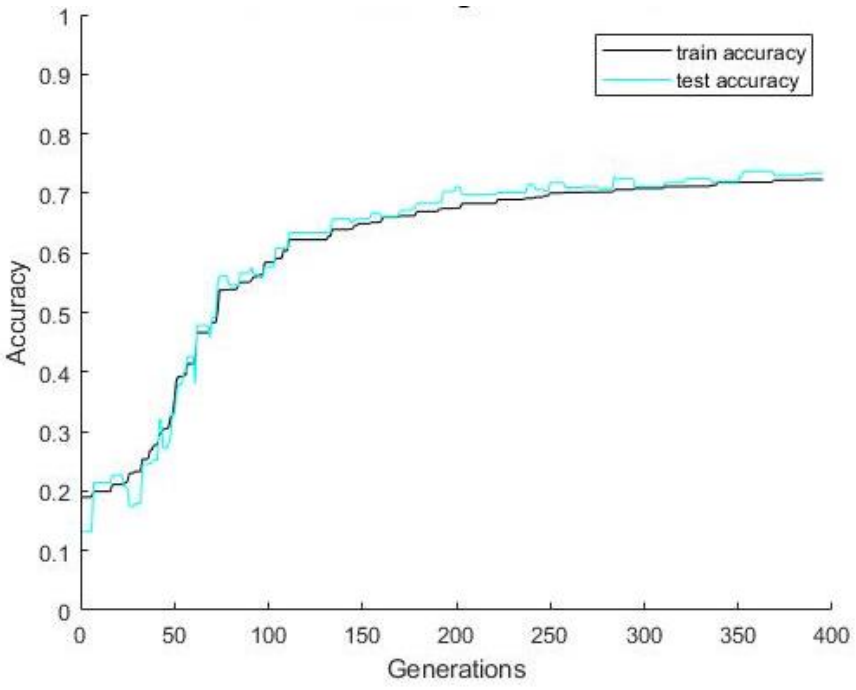


Figure 19b: Training and Validation accuracy of Low-rank GA-CNN (R=5)

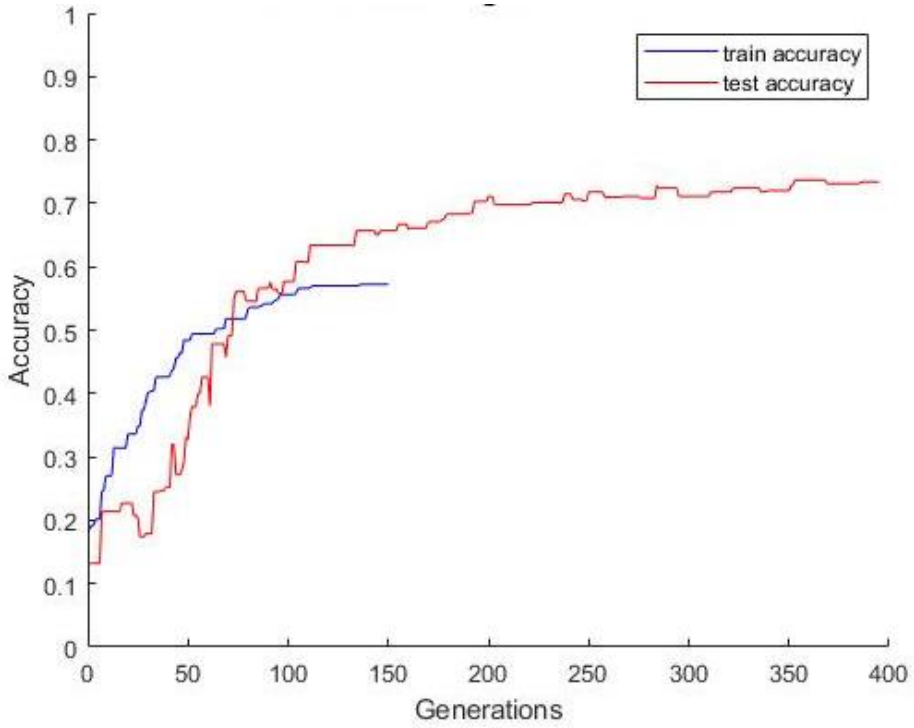


Figure 19c: Training and Validation accuracy of Low-rank GA-CNN (R=11)

4. Train CNN with a hybrid GA-BP algorithm.

The training process of CNN with the hybrid GA-BP algorithm is described in $3.5.5$. Further details about our experiment are shown in $Table\ 7$.

Table 7: BP settings for the GA-CNN model

BP parameters	Value
Epochs	[50, 100, 500]
Learning Rate	0.01
Momentum	0.95
Batch Method	Mini Batch Method
Batch size	100
Training Images per chromosome	[500, 1000, 2500, 5000]
Test Images	the last 1000 of 60,000
Number of updates of the weights	[5, 10, 25, 50]
Initialization of weights	[random, the fittest chromosome]
Hidden Layer	1
Activation Functions	ReLU, Softmax
Pooling	Mean pooling

The results from this experiment are remarkable since the hybrid GA-BP algorithm scored 100% in training accuracy while test accuracy did not overcome 20% no matter how many iterations we performed. This is due to the fact that genetic variability is lost because of BP. In other words, BP is used here to make our chromosomes optimal. After a number of iterations, more and more optimal individuals are added to the population. Genetic operators

see these optimal individuals as the goal-the type of chromosomes that the rest of the population should look like in order to get a high accuracy of our model, and eventually we reach a point where we have exactly the same individuals in the next generations.

Table 8: Train and Validation accuracy of the hybrid GA-BP algorithm

Dataset			Tensor rank	BP every gen	Accuracy %	
train	test	train per chrom			Train	Test
1-5000	last 1000	All	full	1	100	10.1
1-5000	last 1000	all	full	5	100	12.3
1-5000	last 1000	all	full	10	100	15.7
1-5000	last 1000	all	full	20	100	20

Based on our experiments, we can realize that GAs requires a large amount of data, iterations and time to train our CNN satisfactorily, while BP can quickly and accurately identify handwritten digits after a few training epochs. For example, when training our model with 5000 training images, GA-CNN has accuracy equal to 71.1% after 400 iterations and BPNN's accuracy is equal to 97.6 after only $100\ epochs$. After performing $Experiment\ 2$, we come to the conclusion that the accuracy of the model is also relatively consistent across different training datasets. In $Table\ 4$, we notice that whether the training subset is minor or almost half of the original training set $(\mathrm{Fig.}\ 18a\ and\ 18\mathrm{d})$, the accuracy of our model is almost equal to 66.85%.

During our research, we came across two surprising conclusions: training our GA-CNN model with low-rank weights leads to better optimization results in contrast with full-rank weights. Here, the average accuracy is equal to 73.66% which is greater than the accuracies we got on the first two experiments and, is a satisfactory value to compare it with BP. The second unexpected conclusion is the poor performance of our proposed hybrid GA-BPNN model. In theory, this model works by combining the best features of both algorithms, such as the GA's ability to search for an optimal solution and the BPNN's ability to quickly find a local minimum and creates a more powerful and robust predictive model in comparison to traditional algorithms. Yet the efficiency of BP creates a super-fit individual that when inserted into the population eliminates the genetic diversity in just few generations.

5. Conclusions and future work

In this paper, we investigated the training of a deep learning CNN by an evolutionary algorithm, namely GAs. The proposed approach is evaluated on the MNIST dataset.

1. Considering the ability of GAs to train a CNN with increased generalization, we used a constant subset of 1000 images as the test set, whereas varying the train set from 5000 images to 200 images. Each individual was trained with the same train set. Results showed a deterioration of test accuracy from 74% to 67.4% respectively. If BP is applied afterwards with only $8\ epochs$ the values of accuracy become 96.2 and 86.8% respectively.
2. The novel idea of allowing each chromosome to train with a random subset of the available training set is also explored. In these experiments, we allowed the training set per chromosome to vary from 5000 images (the whole training set) to 200 images. Results showed a deterioration of test accuracy from 74% to 64.2% respectively.
3. The effect of the rank of the $9 \times 9 \times 20$ tensor was also studied. To this end, the chromosomes encoded tensors with rank equal to 5, 7 and 11, using the inverse canonical polyadic decomposition (CPD). The achieved test accuracy was effectively constant, which is a remarkable result.
4. The hybridization of GAs with BP was studied next, with striking results. Fine tuning the best chromosome of the generation with BP using just $8\ epochs$, dominates the population, causing quick loss of genetic diversity and resulting in overfitting, achieving 100% train accuracy and very poor test accuracy: 10% if BP is executed $once$ every generation and just over 20% if BP is invoked $once\ every\ 20$ generations.
5. It is also interesting to note that overfitting does not occur even after $400\ generations$. This is evident, since the test accuracy does not seem to decrease, throughout the evolution.

Considering the arithmetic complexity of the GA-training of CNN, one epoch of the BP is equal to the forward pass of the full training set of $N_{tr} = 5000$ images, thus:

$O_{BP}(N_{tr} \times n_{epoch})$.

In the above expression one should add the computational cost of adjusting the network's weights. On the other hand, the GA-training requires

$O_{GA}(n_C \times N_{iter} \times train_{Chrom})$.

Thus, if $\frac{train_{chrom}}{N_{tr}} << 1 \ (12)$ then O_{GA} can become comparable to the arithmetic complexity of BP, O_{BP}.

Although a few novel ideas have been explored in this work, the results presented are considered initial. First, the CNN used for experimentation is very simple. The dataset also consists of very small images of 10 classes. These limitations were necessary to conclude the thesis within the available time frame and with the available computer hardware. Future work includes applying GAs to train more complicated CNNs, with larger datasets. The use of more difficult classification tasks may reveal the superiority of the GAs versus BP, which is still a remaining issue after this work. The BP may also be modified to be applied to low rank tensors of the convolutional layers.

We believe that our proposed approach is an effective way to improve the accuracy of deep learning classifiers. We also observe that the proposed approach can be used for other deep learning tasks, such as object detection and image segmentation. We are currently exploring ways to further improve our approach by incorporating more advanced evolutionary algorithms and other optimization techniques.

References

[1] Alex Pereira da Silva, Comon, P., & de Almeida, A. L. (2015, April). An iterative deflation algorithm for exact CP tensor decomposition. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (pp. 3961-3965). IEEE.

[2] Amin Dastanpour, Suhaimi Ibrahim, RezaMashinchi, "Effect of Genetic Algorithm on Artificial Neural Network for Intrusion Detection System", Advanced Informatics School, University Technology Malaysia, Jalan Semarak, 54100 Kuala Lumpur, Malaysia | Faculty of Computing, University Technology Malaysia, Johor (2016)

[3] Delibasis, K. K., Asvestas, P. A., Matsopoulos, G. K., Zoulias, E., & Tseleni-Balafouta, S. (2008). Computer-aided diagnosis of thyroid malignancy using an artificial immune system classification algorithm. *IEEE Transactions on Information Technology in Biomedicine*, *13*(5), 680-686.

[4] Delibasis, K. K., Asvestas, P. A., & Matsopoulos, G. K. (2011). Automatic point correspondence using an artificial immune system optimization technique for medical image registration. *computerized medical imaging and graphics*, *35*(1), 31-41.

[5] Delibasis, K., Undrill, P. E., & Cameron, G. G. (1996, April). Genetic algorithms applied to Fourier-descriptor-based geometric models for anatomical object recognition in medical images. In *Medical Imaging 1996: Image Processing* (Vol. 2710, pp. 635-645). SPIE.

[6] Delibasis, K., Undrill, P. E., & Cameron, G. G. (1997). Designing Fourier descriptor-based geometric models for object interpretation in medical images using genetic algorithms. *Computer Vision and Image Understanding*, *66*(3), 286-300.

[7] Delibasis, K., & Undrill, P. E. (1994). Anatomical object recognition using deformable geometric models. *Image and vision computing*, *12*(7), 423-433.

[8] Esfahanian, P., & Akhavan, M. (2019). Gacnn: Training deep convolutional neural networks with genetic algorithm. *arXiv preprint arXiv:1909.13354*.

[9] Gill, E. J., Singh, E. B., & Singh, E. S. (2010, September). Training back propagation neural networks with genetic algorithm for weather forecasting. In *IEEE 8th international symposium on Intelligent systems and informatics* (pp. 465-469). IEEE.

[10] Goldberg, D. E. (1989). A Simple Genetic Algorithm in Genetic Algorithms in Search Optimization and Machine Learning.

[11] Huang, H. X., Li, J. C., & Xiao, C. L. (2015). A proposed iteration optimization approach integrating backpropagation neural network with genetic algorithm. *Expert Systems with Applications*, *42*(1), 146-155.

[12] Ijjina, E. P., & Chalavadi, K. M. (2016). Human action recognition using genetic algorithms and convolutional neural networks. *Pattern recognition*, *59*, 199-212.

[13] Jebari, K., Madiafi, M., & Elmoujahid, A. (2013). Parent selection operators for genetic algorithms. *International Journal of Engineering Research & Technology*, *2*(11), 1141-1145.

[14] Kim, P. (2017). *Matlab deep learning with machine learning, neural networks and artificial intelligence*. by Phil Kim.

[15] Kolda, T. G., & Bader, B. W. (2009). Tensor decompositions and applications. *SIAM review*, *51*(3), 455-500.

[16] Liu, H., Simonyan, K., Vinyals, O., Fernando, C., & Kavukcuoglu, K. (2017). Hierarchical representations for efficient architecture search. *arXiv preprint arXiv:1711.00436*.

[17] Liu, P., El Basha, M. D., Li, Y., Xiao, Y., Sanelli, P. C., & Fang, R. (2019). Deep evolutionary networks with expedited genetic algorithms for medical image denoising. *Medical image analysis*, *54*, 306-315.

[18] Luo, X. J., Oyedele, L. O., Ajayi, A. O., Akinade, O. O., Owolabi, H. A., & Ahmed, A. (2020). Feature extraction and genetic algorithm enhanced adaptive deep neural network for energy consumption prediction in buildings. *Renewable and Sustainable Energy Reviews*, *131*, 109980.

[19] Montana, D. J., & Davis, L. (1989, August). Training feedforward neural networks using genetic algorithms. In *IJCAI* (Vol. 89, pp. 762-767).

[20] Such, F. P., Madhavan, V., Conti, E., Lehman, J., Stanley, K. O., & Clune, J. (2017). Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv preprint arXiv:1712.06567*.

[21] Sun, Y., Xue, B., Zhang, M., Yen, G. G., & Lv, J. (2020). Automatically designing CNN architectures using the genetic algorithm for image classification. *IEEE transactions on cybernetics*, *50*(9), 3840-3854.

[22] Tang, S., Gong, R., Wang, Y., Liu, A., Wang, J., Chen, X., ... & Tao, D. (2021). Robustart: Benchmarking robustness on architecture design and training techniques. *arXiv preprint arXiv:2109.05211*.

[23] Tarigan, J., Diedan, R., & Suryana, Y. (2017). Plate recognition using backpropagation neural network and genetic algorithm. *Procedia Computer Science*, *116*, 365-372.

[24] Undrill, P. E., & Delibassis, K. (1997, October). Stack filter design for image restoration using genetic algorithms. In *Proceedings of International Conference on Image Processing* (Vol. 2, pp. 486-489). IEEE.

[25] Vervliet, N., Debals, O., Sorber, L., Van Barel, M., & De Lathauwer, L. (2016). Tensorlab user guide. *Available on: http://www. tensorlab. net*.

[26] Yin, F., Mao, H., & Hua, L. (2011). A hybrid of back propagation neural network and genetic algorithm for optimization of injection molding process parameters. *Materials & Design*, *32*(6), 3457-3464.

[28] Zhang, J., & Qu, S. (2021). Optimization of backpropagation neural network under the adaptive genetic algorithm. *Complexity*, *2021*, 1-9.

Appendix

Implementation details

Genetic algorithm uses the fitness function to evaluate the individuals in a population. Initially, for a constant number of populations, our model will randomly generate the weights of all layers -that need training- for each population. Next, the training set will be fed into the neural network and the predicting process begins. It is important to mention that the training set is deliberately shuffled so that the neural network will have a better generalization. After the fitness calculation, which compares the true output and the predicted output, the program will update the maximum fitness value for the final training since its weights are the optimal ones and could likely yield higher accuracy in the final training stage. This process will continue going on until the maximum generation (iteration) is met.

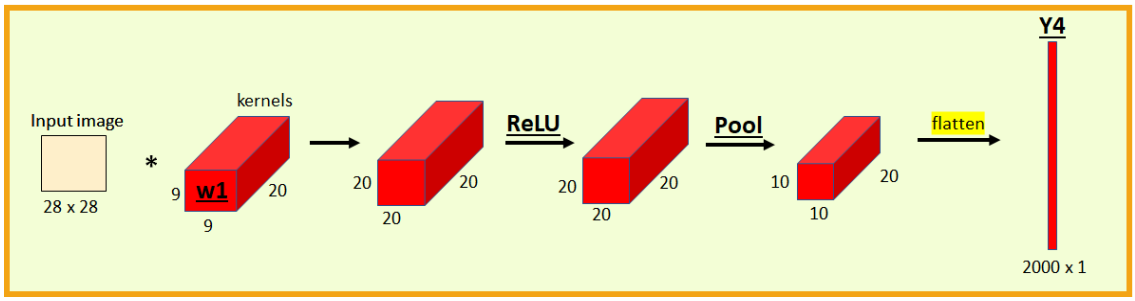


Figure 20: Convolution, ReLU and Pooling of one MNIST image

For the training of Convolutional Neural Networks (CNNs), we need convolutional filters (masks) and the training and testing subsets of the available dataset. There are three ways to train a CNN that refer to the convolutional layer:

1. apply *conv* function to all your training images and convolutional filters.
2. apply *conv* function to each training image and the convolutional filters, using a for-loop from 1 to size of your training subset.
3. apply *conv* function to the training subset and each convolutional filter using a for-loop from 1 to the size of the convolutional filters.

In this project, we implemented the third option (Plot No3), where we have 5000 training images and 20, 9×9 convolutional filters.

Creating Fitness Function

Full-Rank

Accuracy is a metric that generally describes how the model performs across all classes. According to 3.5.2.1, the chromosome is the flattened combination of weight matrices W1,

W5, and Wo. The script for calculating chromosomes' fitness has three input variables: the chromosome, the images and labels of the training set. We need to get the weights that are going to optimize the CNN. Hence, the chromosome is decoded into three parts that have the same dimensions as W1, W5, and Wo. In section 3.2.1, it is explained that W1 represents the weight values of the convolutional layer (size = 1690 or 9×9×20) and this variable is convoluted with the training images. The output of this function is then passed into the ReLU activation function (new size = 20×20×20) and later, to the Mean pooling layer (size = 10×10×20). According to Plot No3, this process is repeated 20 times and the output of each iteration is stored in a variable called Y4 (Fig. 21a). Y4 is a 2000×5000 matrix that passes into the Classification section of our CNN. There, Y4 is multiplied by W5 and their product (size: 100×5000) passes through another ReLU function (new size: 100×1000) (Fig. 21b). Eventually, this product is multiplied by Wo and the product passes into the Softmax function. The output of the Softmax function is a matrix sized 10×5000 and shows our training images' probabilities for classes 0-9.

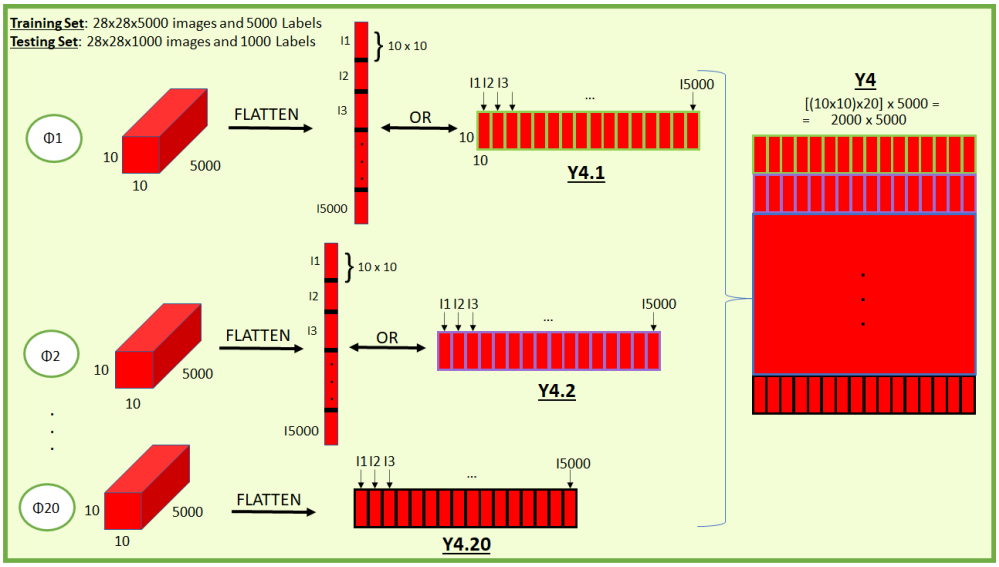


Figure 21a: here there are 5000, 28×28 training images (I1-I5000) that are convoluted with each filter of the convolutional layer (Φ1-Φ20); each result is passed through ReLU and mean-pool functions and then, it is flattened into a short and fat matrix (Y4.1-Y4.20). These matrices are put together to form Y4.

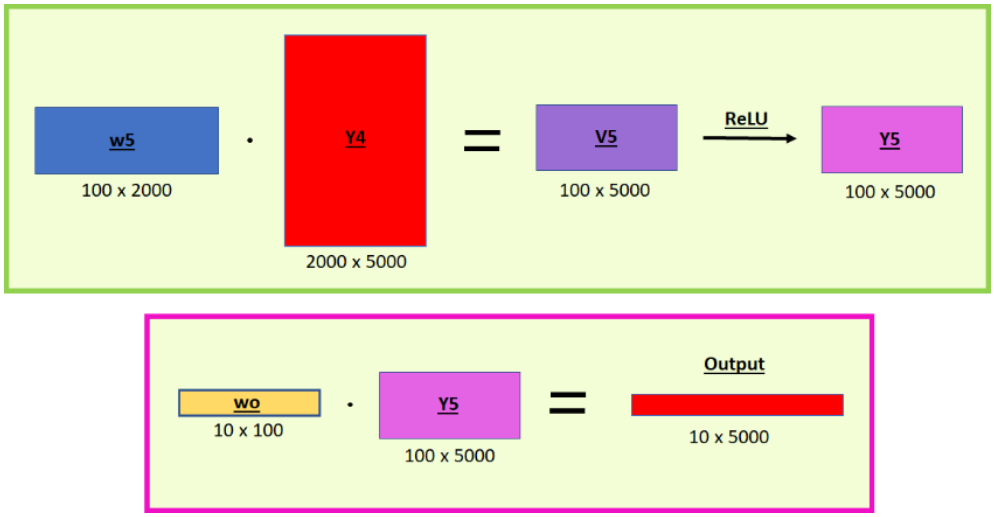


Figure 21b: Y4 is multiplied with W5 and passed through the ReLU function, whose result is then multiplied with Wo; their product is a matrix that contains each image's probability of recognizing classes 0-9.

Low-Rank

The process of calculating the fitness of low-rank chromosomes is the same as described above. There is only one difference between the two scripts which is the decoding of the chromosomes. In details, this script has four input variables: the chromosome, the images and labels of the training set and the Rank of the tensors. In the proposed work, we choose to turn W1 into a low-rank tensor. Hence, we need to fix its sub-matrices accordingly.

W1 is composed of the Core tensor Σ and the factor matrices: A, B, C, as mentioned in section 2.4. Here, da Silva et. al.'s functions were used in order to build the fitness function for the low-rank chromosomes. In details, W1 is decomposed to the corresponding three factor matrices and, we create a variable, called alpha, that works as a measure value for the decomposition of our input chromosome. Then, our CNN is trained and its accuracy is calculated by comparing the true output and the predicted output of our model.