國立臺北科技大學

資訊工程系碩士班

碩士學位論文

Department of Computer Science &

Information Engineering

Master Thesis

Use Efficiency-Oriented Genetic Programming to Create Loss Function for Image Classification Tasks

研究生：周子榆

Researcher: Tzu-Yu Chou

指導教授：陳香君博士

Advisor: Shiang-Jiun Chen, Ph.D.

July 2025

國立臺北科技大學
研究所碩士學位論文口試委員會審定書

本校　資訊工程　研究所　周子榆　君

所提論文，經本委員會審定通過，合於碩士資格，特此證明。

學位考試委員會

委　　員：吳和庭

馬奕葳

陳香君

指導教授：陳香君

所　　長：劉建宏

中 華 民 國　一百一十四　年　七　月　九　日

Abstract

Keyword: Image Classification, Genetic Programming, Loss Function, Deep Learning

In the recent, with the rapid rise of deep learning, image classification models have quickly become one of the most popular and widely known models. When training such models, the steps are broadly divided into the following: preparing image datasets, dividing them into training, validation and testing sets as needed, training the model, evaluating the model, and repeating these steps until the ideal result is met or the computational resources are exhausted.

A crucial function during the process of training a model is called the loss function, which calculate the difference between the predicted values and the ground-truth values. The results of the loss function can significantly influence the effectiveness of the model's training because it simply decides the direction of the adjustment to the model. However, designing a loss function often requires the assistance of experts in the related field, leading to a resource-intensive design process. Recent research has proposed using Genetic Programming (GP) to generate loss functions to avoid the necessity of hiring numerous domain experts for assistance. Nevertheless, using this kind of algorithms typically result in decreased computational efficiency. This study aims to improve the efficiency of this class of methods by incorporating the concept of a sample tournament into the existing selection mechanism of GP. Additionally, we refine the crossover method to enhance population diversity. By implementing these modifications, we ensure the necessary randomness in the selection process, effectively reducing the likelihood of repeatedly selecting the same top-performing individuals for genetic operations.

The experimental findings demonstrate that the implementation of the proposed algorithm substantially decreases execution time while preserving comparable performance outcomes. In particular, the algorithm yielded execution time reductions of 64% and 62.9% in small-scale and large-scale experiments, respectively. Furthermore, in large-scale settings, model accuracy was enhanced by 9% on the CIFAR-10 dataset and by 7% on the CIFAR-100 dataset.

Acknowledgements

在本論文完成之際，謹向在此過程中給予我指導與支持的所有人致以誠摯的感謝。

首先，我衷心感謝我的指導教授陳香君博士，在撰寫論文的各個階段提供了悉心的指導與幫助。教授的專業知識、耐心教導與無私協助，使我得以順利完成本篇論文，其恩澤將令我銘記在心。

此外，我要感謝我的父母、妹妹、外婆、舅舅、大哥與大嫂，感謝你們在我攻讀碩士學位期間，以無條件的愛與支持陪伴著我，在我遇到挑戰時給予我力量與鼓勵。謝謝你們在這段求學旅程中始終相信我、支持我，是你們成就了我今日的成果。

我也誠摯感謝在研究所期間結識的夥伴 Felix、Elija、Sven、Fabian、Merjem、Ifi、Welsey、Byrant、Mick、Ricky、Grant、Meteor、Seraphin、Alexis、Sam、Eric、Gary、璟霆、劭睿、正承、建璋、祥語、玟萱、宏傑、德劭、楊越、柏勳、詠暄、泳霈、承諺、伃君。謝謝你們一路相伴，無論是在課堂討論、論文交流，或是日常生活的分享中，都給予我深刻的啟發與正面的影響。你們的陪伴與支持，讓我的研究生生活豐富而精彩，是我珍貴的回憶。

最後，感謝我摯愛的朋友旻玄、昌澔、其康、志傑、暐澈、楷又、昱伶，在我攻讀碩士的兩年間給予我無微不至的關懷與鼓勵，從學業上的建議，到生活中的陪伴與放鬆時刻，都是我完成本論文不可或缺的動力來源。願我們的友誼長存，並肩迎接更加燦爛的未來。

周子榆　謹誌於

臺北科技大學資訊工程系碩士班

中華民國 114 年 7 月 18 日

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

After several decades of development, deep learning [1] technology has achieved significant breakthroughs in the last ten years, largely due to the remarkable improvements in GPU computational power. Consequently, the predictive and analytical capabilities of models have achieved substantial advancements. Numerous models have been launched by major companies and applied in practical scenarios, leading to significant changes in our daily lives.

When it comes to training deep learning model, we usually refer to the following steps: collecting relevant data and organizing it into datasets, partitioning the datasets into training and validation sets as needed, initializing model parameters, training the model, evaluating the performance of the model, adjusting the model parameters, and repeating the training until the target is achieved or computational resources are exhausted. During the evaluation of the model's performance, we use a function called the loss function [2]. Its purpose is to calculate the difference between the model's predicted results and the ground truth, which helps model adjust its parameter to better fit the target. Therefore, different loss functions can influence the direction of model adjustments, significantly impacting the final outcomes of the model.

However, the design of a loss function is often closely related to the propose of the model [3]. In other words, different models may require experts from specific fields to assist in designing the loss function to enhance the speed and effectiveness of model training. Nevertheless, recent research has shown that it's feasible to use genetic programming (GP) [4] to automatically generate loss functions. Due to the domain-independent nature of GP, it allows us to develop an algorithm that can automatically create the required loss function without needing specialized knowledge in that particular field. Although such methods are not a primary focus within the deep learning research domain, a prior study [5] has demonstrated that designing and utilizing novel loss functions, which departs from conventional model tuning practices, can serve as an effective approach for improving deep learning performance. However, we observed that these algorithms typically demand substantial time and computational resources to construct and evaluate the loss functions. Therefore, we aim to refine certain mechanisms introduced in earlier work to reduce execution time while maintaining comparable performance. Before presenting our proposed method, we first provide a brief overview of GP.

The operation mode of GP can be simply divided into the following steps: Representing the solution we hope to find (which may be a value or a function) in an encoded form, then forming these solutions into a population. After evaluating the whole population, we perform genetic operations (e.g., mutation or crossover) on the population's solutions, reevaluate them, and repeat the above steps until the target is achieved or computational resources are exhausted.

Although the computational power of GPUs today is far superior to that of the past, both training models and using GP to find solutions still require substantial computational resources and time to achieve a decent result. In this paper, an efficiency-oriented GP algorithm is proposed to reduce the computational resources and the time required to search for the optimal solution while maintaining the same level of effectiveness. We aim to improve the genetic operation part of the existing GP framework by referencing the concept that offspring in a better living environment can usually receive better care and extending the concept of randomness.

We modify the GP operations so that only a select number of high-scoring individuals are eligible to perform genetic operations. Additionally, we refine the crossover method to enhance population diversity. Our approach consists of the following steps: first, a certain proportion of offspring is randomly selected from the population. From this subset, a fixed ratio of top-performing individuals is chosen to undergo genetic operations. This randomized selection prevents the same individuals from being repeatedly selected, allowing a broader range of offspring the opportunity to improve.

Regarding crossover, unlike previous study [5] that involve selecting either an abandoned individual or an existing individual to perform crossover with the chosen individuals, our approach utilizes either a newly generated individual or an existing individual as the second parent in the crossover process. By implementing this method, we aim to further enhance population diversity.

Experimental results demonstrate that the aforementioned improvements allow the proposed method to achieve comparable performance to peer algorithms while reducing computational time by approximately 57.9% to 64%. Moreover, in large-scale experiments, our method exhibits a 9% increase in accuracy on CIFAR-10 and a 7% increase on CIFAR-100 compared to peer algorithms.

The structure of this paper is as follows: Chapter 2 is related work. This chapter provides a comprehensive overview of the development and history of GP, a detailed explanation of loss functions and their role in deep learning models, an analysis of how loss functions are randomly generated through GP, and a review of image classification along with its most commonly used loss functions. Chapter 3 is proposed algorithm. In this chapter, we will explain the flowchart of the algorithm implemented in this paper. After that, we will then describe the software and hardware requirements, the experimental environment setup and the pseudo code of this method. Finally, we will provide an in-depth explanation of the method used in this paper. Chapter 4 is result and analysis. This section provides a detailed explanation of the experimental methodology and presents our results. Subsequently, we analyze the findings by organizing the experimental data into tables and figures, allowing for a direct comparison with peer algorithm. Finally, we interpret the outcomes and examine the reasons our algorithm achieves comparable results while significantly reducing computational resource consumption. Chapter 5 is conclusion and future work. In this chapter, we will provide the conclusion of this paper by outlining the contribution we have made so far. After that, we would like to discuss potential directions for future research or possible way to apply this algorithm in practical.

Chapter 2 Related Work

In the following sections, we will introduce the concept of metaheuristics [6], [7] . Then, we will discuss the importance of loss functions and their role in deep learning [1] models. Finally, we will explore image classification [8] and its real-life applications.

2.1 Metaheuristic

In the following section, we will discuss metaheuristic and some of the most famous metaheuristic algorithms include Evolving Algorithm (EA) [9], Genetic Algorithm (GA) [10], and Genetic Programming (GP) [4]. After that, we will introduce related research in the GP domain.

2.1.1 Overview of Metaheuristic

Metaheuristic [11] was first proposed by Glover. It can be regarded as a high-level procedure used to find solutions to optimization problems or other issues that conventional algorithms cannot solve. Within metaheuristics, an important concept is that these procedures must find a potentially optimal solution under reasonable computational costs or insufficient information. In this paper, we will primarily focus on nature-inspired metaheuristics [6]. In these methods, a common approach is to use a population-based strategy to find the optimal solution. Within a population, each individual typically represents a potential solution. We perform various operations on the individuals within the population to achieve our goal of approximating the optimal solution. A subset of specialized algorithms falls under population-based methods [12], commonly referred to as evolving algorithms (EAs). Commonly used methods within EAs include genetic algorithms (GA), genetic programming (GP) , evolutionary programming (EP) [13], and differential evolution (DE) [14]. The relationship between Metaheuristic, EA, GA, GP, EP and DE is illustrated in the figure 2.1. We will briefly discuss EAs and GAs, and then we will focus primarily on GPs in the following paragraph.

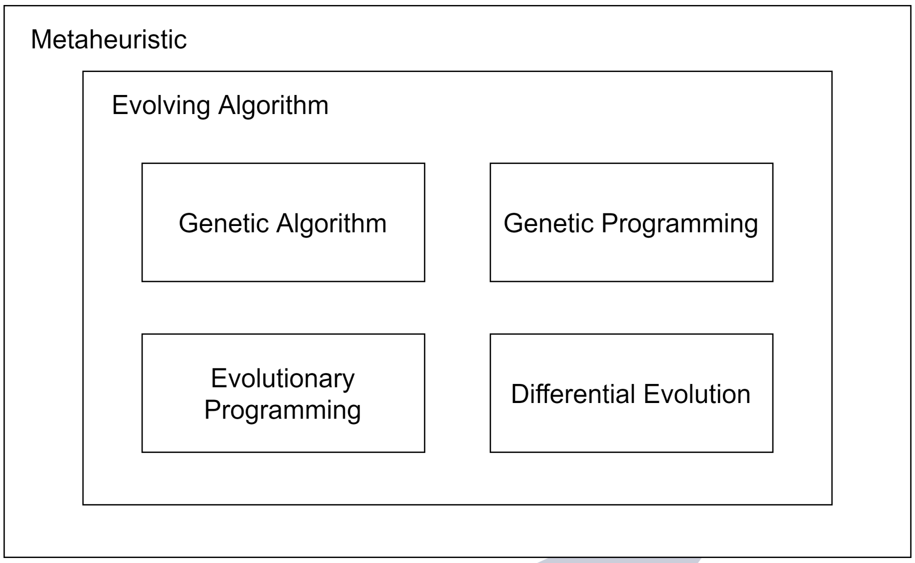


Figure 2.1 Relationship between metaheuristic, EA, GA, GP, EP, and DE

2.1.2 Evolving Algorithms

EAs can be described as algorithms inspired by the concept of ”survival of the fittest [15].” These algorithms often take cues from natural evolutionary mechanisms to create algorithms that mimic animal behavior in the search for optimal solutions. Common examples include Ant Colony Optimization (ACO) [16], GA, and Particle Swarm Optimization (PSO) [17]. The common EA procedure is shown in figure 2.2. In this type of algorithm, the typical approach is to initialize a population, which consists of several individuals. Each individual represents a potential optimal solution. After initializing the population, a special function is used to calculate the fitness value of each individual, determining their level of excellence. Then we set a number to represent how many generations we want this population to evolve. Next, we perform selection, mutation, and crossover on the population. Selection involves choosing individuals based on their fitness to reproduce the next generation. Mutation generally refers to making random changes to an individual, while crossover combines the characteristics of two individuals to create the next generation. Following that, we will conduct an evaluation of the newly created individuals. These steps are repeated until the predefined number of generations is reached, or the individuals fail to achieve the expected results, leading to a forced stop. Ultimately, the best individual in the population is obtained as the solution.

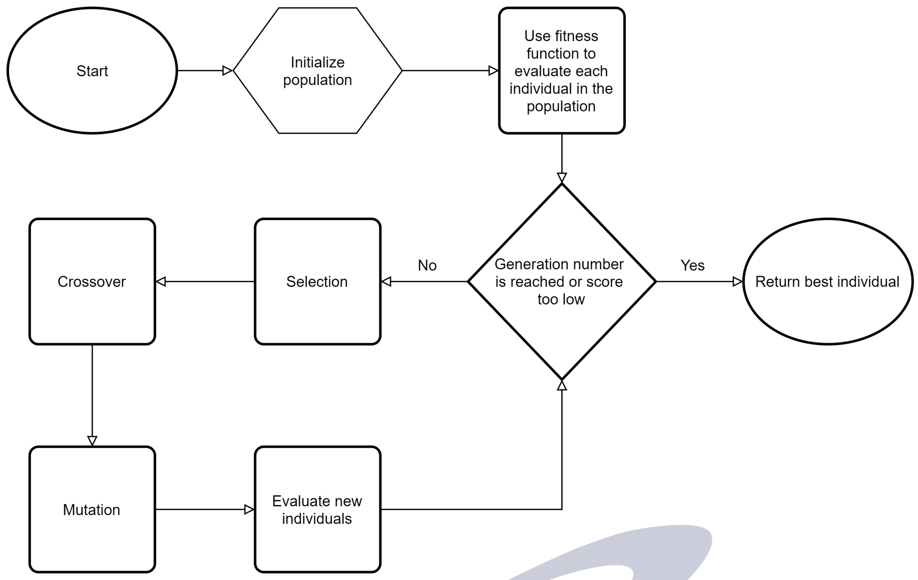

Start
Initialize population
Use fitness function to evaluate each individual in the population
Generation number is reached or score too low
Yes
Return best individual
No
Selection
Crossover
Mutation
Evaluate new individuals

Figure 2.2 Flowchart for EA

2.1.3 Genetic Algorithms & Genetic Programming

GA is considered a type of EA, utilizing the representation of each solution in the form of chromosomes to perform selection, mutation, and crossover. GP is regarded as a special case of GAs, but due to its versatility and practicality, it is seen as a reliable solution. Unlike GAs, GP typically uses a more robust encoding method to represent chromosomes. For instance, GAs often use strings to represent chromosomes, which can present potential issues that need to be addressed, such as the priority of operations for each symbol or the validity of the equation itself. As shown in the figure 2.3, if we want to express the function $(x + y) \div (5 * x)$ in GA, the chromosome representation of this function will be $x + y \div 5 * x$. Without the proper parentheses, it can easily cause confusion. In contrast, GP usually represents chromosomes as trees, a method that can employ predefined traversal techniques to avoid these issues. As illustrated in the figure 2.3, the same function represented by GA can be expressed through a tree structure in GP. By specifying the use of inorder traversal, we can obtain the same function. It is worth noting that in GAs and GP, we can leverage their inherent ability to evolve automatically to find a reasonable optimal solution without having an in-depth understanding of the knowledge domain related to the problem we aim to solve. In the following paragraphs, we will introduce related research.

In [18], Co-Reyes et al. proposed a meta-learning reinforcement learning algorithm. They

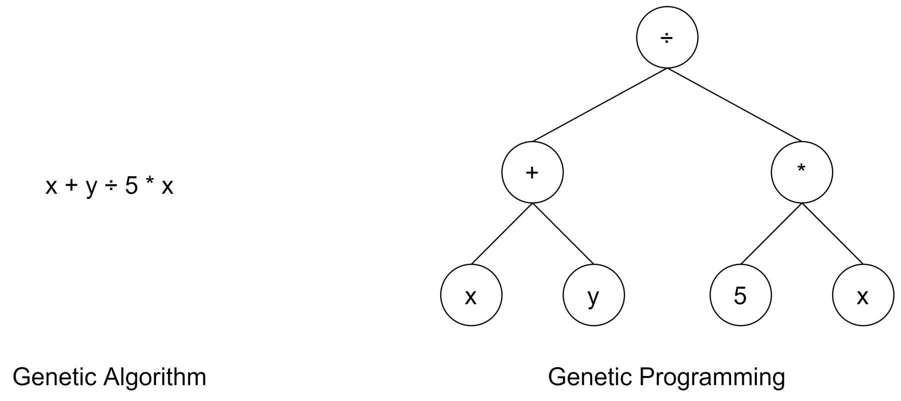


Figure 2.3 Different chromosome representation between GA and GP

used computational graphs to represent algorithms. By doing so, algorithms could be identified, calculated, and optimized through Reinforcement Learning (RL) [19]. Notably, in this study, algorithms were represented as directed acyclic graphs (DAG) [20] of nodes. Within the DAG , all nodes were classified into three categories: input nodes, parameter nodes, and operation nodes. Once the algorithms were represented as DAGs, they could be placed into RL for training and evaluation. In this proposed algorithm, they employed the concept of regularized evolution [21] to evolve a population formed by several randomized and known algorithms. The method was as follows: initialize the population with algorithms, evaluate each algorithm in the population and record their performance, then in a loop, repeatedly use a sample tournament [22] to select algorithms, perform mutations on the algorithms by the mutator they designed, and evaluate them again until the loop ends. During the evaluation process, they trained and assessed the algorithms using RL, continuously testing the performance of each algorithm in different training environments. They also utilized normalized training performance to avoid numerical biases caused by varying environments. Through this approach, the study successfully created two algorithms, named DQNClipped and DQNReg, which outperformed classical control tasks.

In [5], Akhmedova and Körber proposed a genetic programming-based method to find a better function for the image classification training task. They encoded the loss functions into trees, with each tree considered an individual. All individuals were aggregated into a population. In this study, the population was evolved across multiple generations. Before the start of each generation, all individuals were evaluated. Each individual had a probability of undergoing crossover with an individual from a special external archive to create a new individual; otherwise, it would perform the crossover with another individual. Following this, two probabilistic decisions were made. If successful, the new individual could perform subtree mutation or one-point mutation, respectively. At the end of each generation, the fitness value of all individuals was re-evaluated. A certain number of low-scoring individuals were eliminated to the special external archive mentioned earlier, maintaining the stability of the population's size. After these steps, a new generation began, continuing until a predefined number of generations was reached. By employing this method, this study successfully evolved an outstanding individual within the population, creating a function that could train an image classification model more effectively compared to Cross Entropy (CE) [23].

2.2 Loss Function

In this section, we will first introduce deep learning model and explain how does it work. Secondly, we will discuss the importance of loss function and present some related researches.

2.2.1 Deep Learning Model

In recent years, due to the significant increase in the computational speed of GPUs, training deep learning models to solve a wide variety of problems has become widely regarded as a feasible and popular solution. The process of training a deep learning model is often divided into several steps: collecting the data needed to train the model, and adding ground truth values to the data according to the model's requirements. Ground truth can be thought of as the ideal answers we hope to achieve after the model's computation. Once the data collection is complete, these data sets are collectively referred to as the dataset. Typically, the dataset is divided into training sets, validation sets, and testing sets. Data from the training set are used to train the model, the validation set is used to evaluate the model's effectiveness after each training loop, and the testing set is used to calculate the model's final score and determine whether the model successfully achieves its designed purpose. It is worth noting that the testing set data should not be seen during training and validation phases. After dividing the dataset, we decide on the model's architecture. Common architectures include Fully Connected Networks (FCN) [24] and

Convolutional Neural Networks (CNN) [25], etc. Then the training process can begin. During training, we compare the model's output with the ground truth of the training data and use a loss function to calculate the difference between the output and the ground truth. This guides the adjustment of the model's parameters. Therefore, the loss function significantly determines the effectiveness of model training, and this aspect will be explained further later. After adjusting the model, the next round of training begins, continuing until the training is deemed ineffective or a predetermined maximum number of iterations is reached.

2.2.2 Loss Function in Deep Learning Model

In section 2.2.1, we discussed the role of loss functions in deep learning models. Here, we would like to introduce some commonly used loss functions along with their advantages and disadvantages. Mean-Square Error (MSE) [26] is a loss function used for solving regression problems. It calculates the difference between the actual and predicted values, squares them, and then takes the average of these squared differences to obtain the final loss value. However, a notable disadvantage of MSE is that it amplifies the impact of outliers exponentially. Cross-Entropy (CE), unlike MSE, leverages probability concepts to help compute the error in classification tasks. In other words, CE measures the difference between the predicted probability and the true probability to calculate the error. CE can also be adapted to different types of classification tasks to better suit their requirements, with common variants including Binary CE Loss [27] and Multiclass CE Loss [28].

From the above paragraphs, it seems that using a single type of loss function to train all models is considered impractical. Therefore, researchers design different loss functions based on specific needs to calculate more appropriate loss values for the model. As the introduction above indicates, designing a loss function requires an in-depth understanding of the model and a clear grasp of how to define and calculate the loss value. Consequently, designing a loss function, similar to adjusting hyperparameters or model architecture, is considered as a challenging task that needs a deep understanding of the domain for effective design and adjustment.

In [29], Gonzalez and Miikkulainen proposed a meta-learning approach to create a loss function called Genetic Loss-function Optimization (GLO). Through their research, the authors experimentally found a loss function named de novo, created by GLO, outperforms the well-known CE loss in standard image classification tasks. Additionally, because GLO enables training to be completed in fewer steps, this method also enhances the speed and efficiency of the training process.

In [5], Akhmedova and Körber utilize GP to create a loss function that outperforms CE in image classification. The method described in the paper led to the creation of a loss function named Next Generation Loss (NGL). When trained with the Inception model, NGL outperforms CE on datasets such as CIFAR-10 [30], CIFAR-100 [31], and Fashion-MNIST [32]. Additionally, it demonstrates excellent performance on larger datasets like ImageNet [33]. Furthermore, NGL is also effective in improving model performance in segmentation downstream tasks.

From the above paragraphs, we can observe that numerous studies indicate that in image classification tasks, if a well-designed loss function is developed, it has the potential to outperform the most famous and widely regarded as the most effective CE loss function. However, as mentioned earlier, designing a loss function is usually considered a resource-intensive task. Therefore, the two studies discussed above have begun to use genetic programming to enable computers to automatically design a loss function that can effectively collaborate with a model in a specific domain and significantly improve the model's performance.

2.3 Image Classification

Image classification is a technique in computer vision that enables computers to identify what object present in an image. This technology is often used with image localization [34]. Image localization determines the location of objects within an image, using bounding box [35] to enclose the areas where the objects are found. Additionally, object detection [36] is a similar technique to image classification and image localization. Unlike image classification and image localization, which focus on identifying one label per image, object detection can identify multiple labels in a single image. The difference between image classification, image localization and object detection is illustrated in the figure 2.4. We can see that image classification is capable of recognizing which label the entire image belongs to, whereas image localization identifies which specific regions of the image belong to which labels. Object detection is a more complex technique, which can identify multiple labels and also mark the object by using bounding boxes.

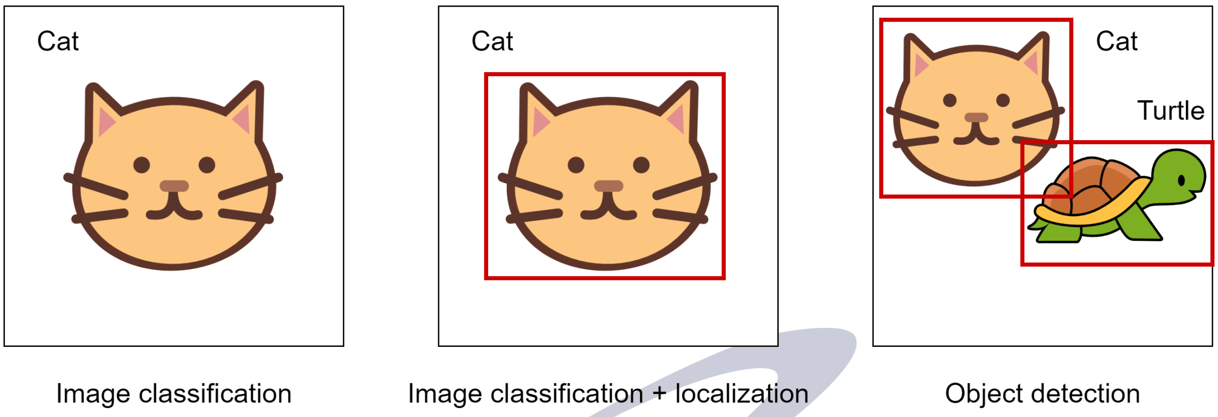


Figure 2.4 Difference between image classification, image localization and object detection

In image classification, the most common types are binary classification [37], multiclass classification [38] and multilabel classification [39]. As the name implies, in binary classification, there are only two possible outcomes when identifying an image. For instance, if the model's labels are cat and dog, the prediction can only be either a cat or a dog. Conversely, multiclass classification means that the image can belong to multiple possible classes instead of two classes. Using the previous example, the prediction in multiclass classification could be a cat, dog, elephant, snake, or other different animals, rather than choosing between just two labels. It is important to note that the final prediction in both binary and multiclass classification will result in only one label. If an image needs to have multiple labels, a different type of image classification model is required, such as a multilabel model.

The process of training an image classification model is generally similar to what is described in section 2.2.1 on Deep Learning. In image classification, commonly used datasets include: ImageNet [40], CIFAR-10, CIFAR-100, and MNIST [41]. It's worth mentioning that we often perform preprocessing on images. Common preprocessing techniques include image cropping, image resizing, and image normalization. Image cropping is a technique that removes the unnecessary parts of the image. Through image cropping, the irrelevant parts of the image will be removed. As shown in the figure 2.5, the pink area in the original picture is removed to prevent model from learning the useless information. Image resizing ensures that all images are of the same size, which helps improve computational speed. However, there is a potential risk of degrading the model's performance by using image resizing [42]. Hence, the size of the resized images is typically determined based on actual requirements. Normalizing images also helps the model learn from each image more effectively. After preprocessing, we usually use pretrained models instead of designing a new model architecture from scratch. This approach enhances computational efficiency and results. Common image classification models used include InceptionV3 [43], ResNet [44], and others.

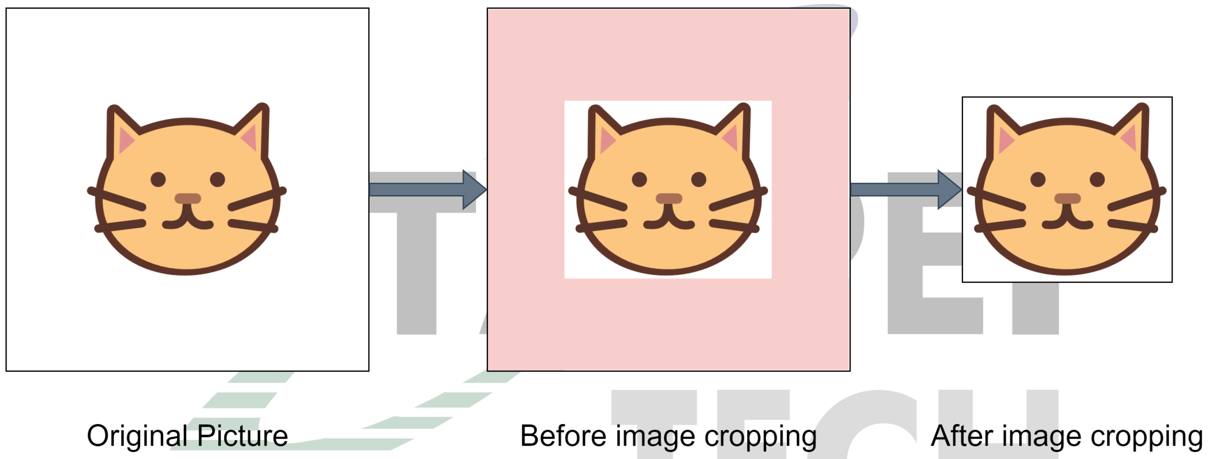


Figure 2.5 The process of image cropping

In the real world, image classification can be applied in many areas. For example, it can be used in medical imaging to identify tumors in X-rays. It can also be employed in autonomous vehicles to quickly recognize objects surrounding the vehicle. Facial recognition is another application of image classification, used for company access control and as a method to unlock personal devices such as phones or computers.

Chapter 3 Proposed Algorithm

In this chapter, we will introduce the architecture and flowchart of our algorithm. Next, we will provide an in-depth explanation of our proposed algorithm, including the underlying concepts and the rationale behind our approach. Subsequently, we will present a detailed description of our software and hardware specifications, as well as the setup procedures. Finally, we will execute the experiment and describe how we implemented our concept and successfully completed the entire experiment.

3.1 Main Method

On our existing hardware, we use Windows 11 as our operating system. In addition, we use Python 3.9.18 as the programming language for this research. Based on Python 3.9.18, the main packages we use are TensorFlow, NumPy, and Scikit-learn.

In previous research [5], we observed that genetic operations are performed on the entire population, requiring the evaluation of the entire new population after each complete set of genetic operations. However, the evaluation process is time-consuming. Therefore, in this study, we will employ the concept of a sample tournament to select certain individuals rather than the entire population for genetic operations. By doing so, we only need to evaluate the newly generated individuals. Figure 3.1 illustrates the conceptual flowchart of this study.

As illustrated in the figure 3.1, we initialize a population consisting of P individuals before commencing the experiment. Once initialized, we evaluate all individuals, store their scores, and identify the best score for subsequent analysis and research. The process then enters a loop, continuing until the generation number reaches the specified value. Initially in this loop, we use a sample tournament to randomly select N individuals from the population, followed by choosing the top X individuals for genetic operations (crossover and mutation). The offspring of these X individuals are then evaluated and added to the population. We update the best-performing individual and remove the poorly performing individuals until the population size returns to P. Upon concluding the loop, we generate the best-performing loss function and conclude the experiment.

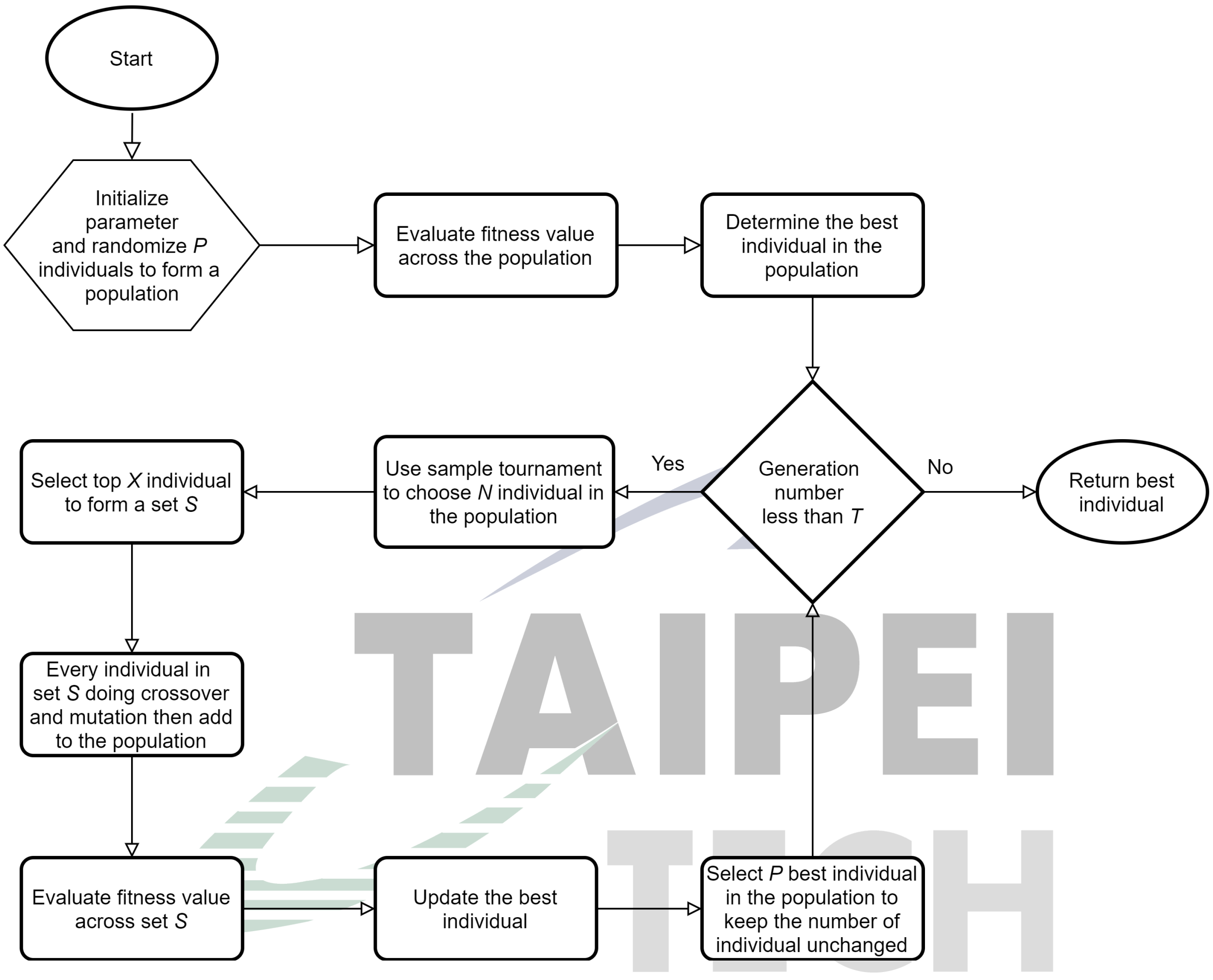

Start
Initialize parameter and randomize *P* individuals to form a population
Evaluate fitness value across the population
Determine the best individual in the population
Generation number less than *T*
Yes
No
Return best individual
Use sample tournament to choose *N* individual in the population
Select top *X* individual to form a set *S*
Every individual in set *S* doing crossover and mutation then add to the population
Evaluate fitness value across set *S*
Update the best individual
Select *P* best individual in the population to keep the number of individual unchanged

Figure 3.1 FlowChart for efficiency-oriented GP

3.2 Implementation Details

To implement our experiment, the hardware and software requirements are first defined in Section 3.2.1. Next, the environment setup steps are described in Section 3.2.2. Finally, the implementation steps are explained in Section 3.2.3.

3.2.1 Requirements

Table 3.1 outlines our software requirements. Our code execution environment is Windows 11. We utilize Python version 3.9.18 and TensorFlow version 2.6.0. The specific packages used within Python and TensorFlow are detailed in Table 3.2.

Table 3.1 Software requirements

Software Type	Software Name	Version
Operating System	Windows	11
Programming Language	Python	3.9.18

Table 3.2 Python package requirements

Package Name/Version	Imported from	License
tensorflow/2.6.0	N/A	Apache License 2.0
numpy/1.20.3	N/A	Modified BSD License
cudnn/8.2.1	N/A	[45]
os	N/A	[46]
copy	N/A	[46]
time	N/A	[46]
math	N/A	[46]
random	N/A	[46]
datetime	datetime	[46]
cmp_to_key	functools	[46]
Sequential	tensorflow.keras.models	Apache License 2.0
Dense	tensorflow.keras.layers	Apache License 2.0
Flatten	tensorflow.keras.layers	Apache License 2.0
Dropout	tensorflow.keras.layers	Apache License 2.0
UpSampling2D	tensorflow.keras.layers	Apache License 2.0
BatchNormalization	tensorflow.keras.layers	Apache License 2.0
InceptionV3	tensorflow.keras.applications.inception_v3	Apache License 2.0
preprocess_input	tensorflow.keras.applications.inception_v3	Apache License 2.0
ReduceLROnPlateau	tensorflow.keras.callbacks	Apache License 2.0
to_categorical	tensorflow.keras.utils	Apache License 2.0
train_test_split/1.5.2	sklearn.model_selection	BSD 3-Clause
ImageDataGenerator	tensorflow.keras.preprocessing.image	Apache License 2.0
np_config	tensorflow.python.ops.numpy_ops	Apache License 2.0

Table 3.3 presents our hardware requirements. The computer we used to execute our code is equipped with a 13th Gen Intel(R) Core(TM) i7-13700 CPU, 32GB of memory operating at a frequency of DDR5-5600MHz, and an NVIDIA RTX A2000 12GB GPU to meet our computational needs.

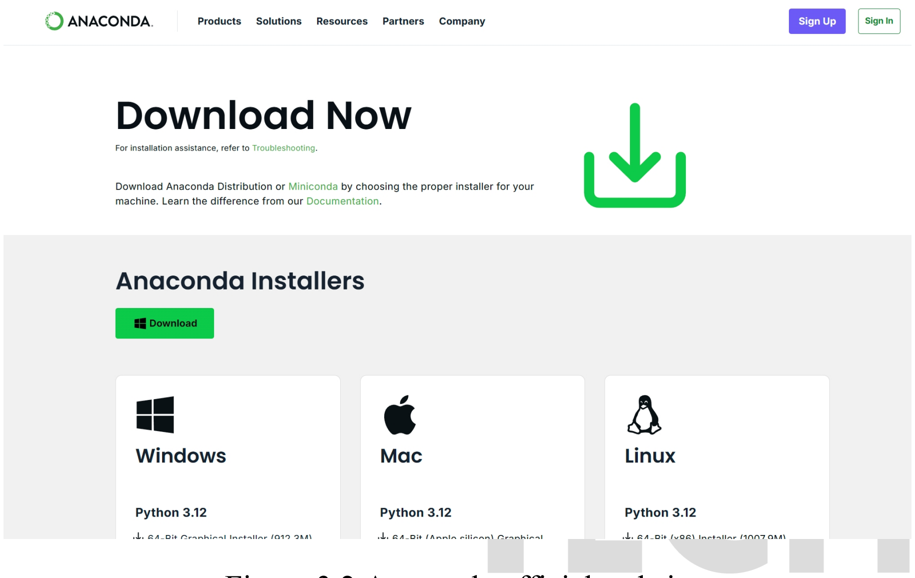
3.2.2 Environment Setup

The following section will describe how to properly set up the environment we used to conduct our experiment. First, we used Anaconda as our package management tool for the im-

Table 3.3 Hardware requirements

Hardware Type	Name
CPU	13th Gen Intel(R) Core(TM) i7-13700
Memory	DDR5-5600MHz 32 GB
Graphic Card	NVIDIA RTX A2000 12GB

plementation. Therefore, you must first visit the Anaconda official website (`https://www.anaconda.com/download`) to download the software. As shown in figure 3.2, click the Download button to complete the download.



Download Now

For installation assistance, refer to Troubleshooting.

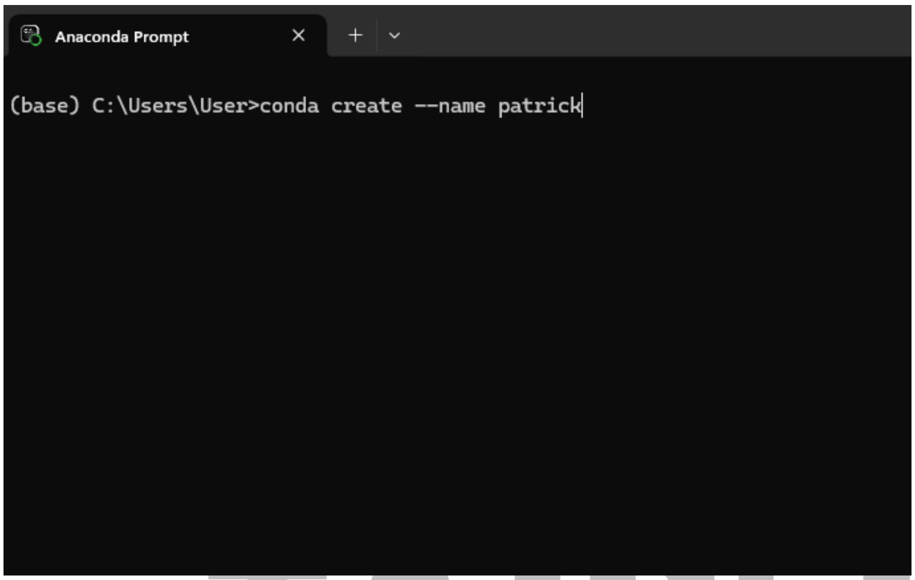
Download Anaconda Distribution or Miniconda by choosing the proper installer for your machine. Learn the difference from our Documentation.

Anaconda Installers

Figure 3.2 Anaconda official website

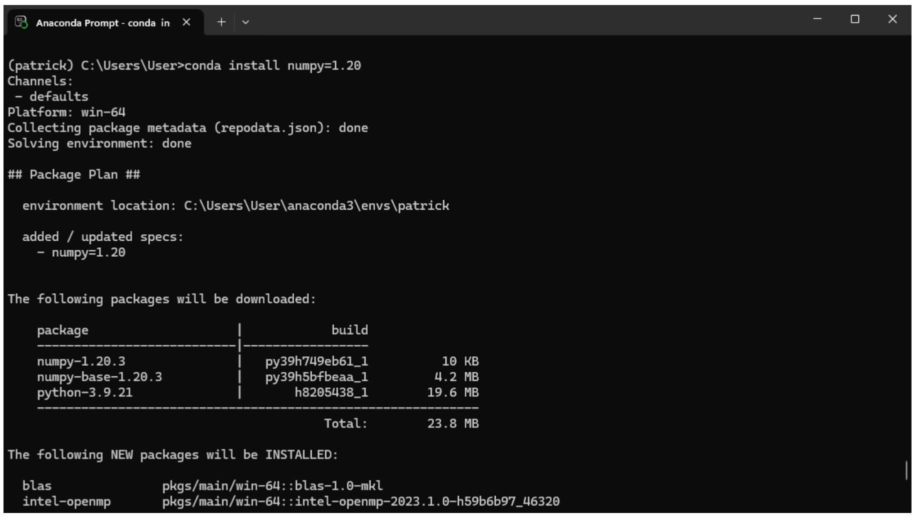
After installation, we need to create a new environment to ensure the independence of all the packages. In Anaconda prompt, enter `conda create --name environment-name` to create a new environment, as shown in figure 3.3. Next, enter `conda activate environment-name` to activate the environment just like the first prompt entered in figure 3.5.

Since directly installing TensorFlow may result in the installation of incompatible versions of NumPy, we first use the command shown in figure 3.4 to download version 1.20.3 of NumPy: `conda install numpy=1.20`. Afterward, we can install the GPU version of TensorFlow by entering the following command in the Anaconda prompt: `conda install tensorflow-gpu`, as shown in figure 3.5. Finally, we need to download Scikit-learn to split the dataset during the model training process. As shown in figure 3.6, enter the following command to complete the final installation step: `conda install scikit-learn`.



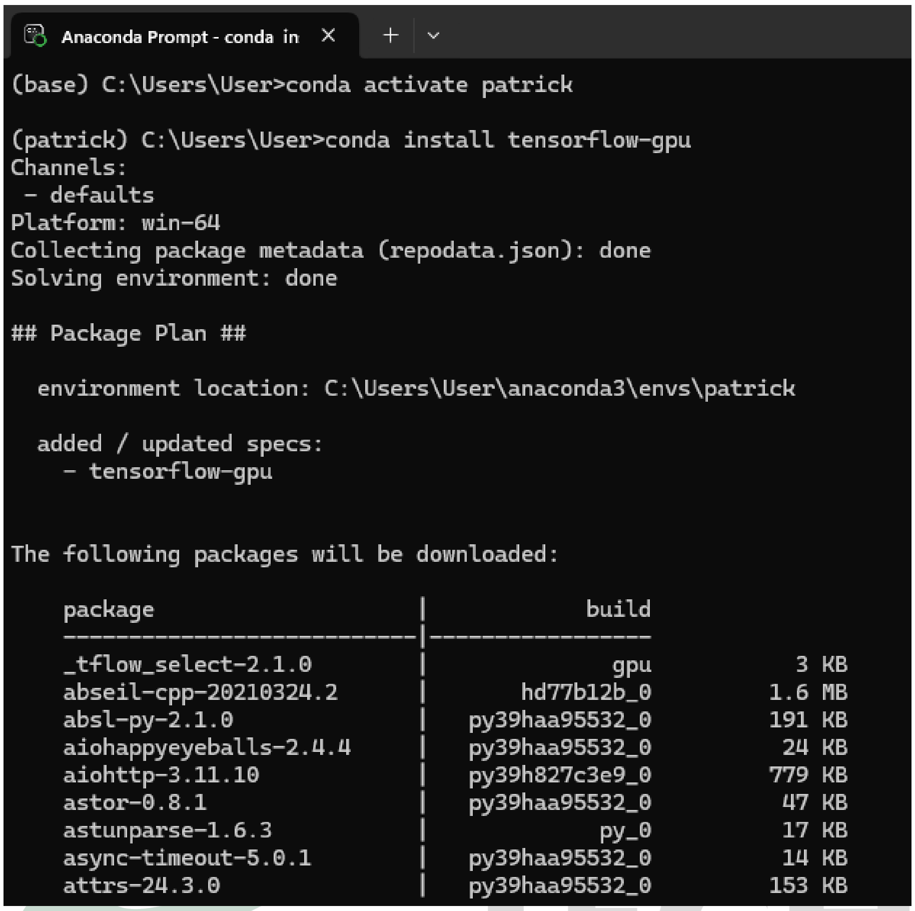
```
(base) C:\Users\User>conda create --name patrick
```

Figure 3.3 Conda prompt for creating new environment



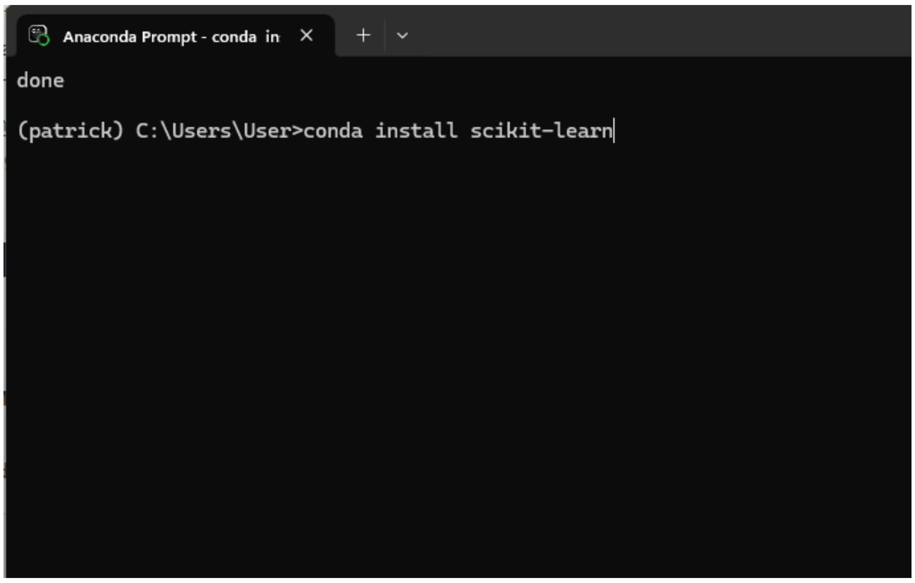
```
(patrick) C:\Users\User>conda install numpy=1.20
Channels:
 - defaults
Platform: win-64
Collecting package metadata (repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: C:\Users\User\anaconda3\envs\patrick

  added / updated specs:
    - numpy=1.20


The following packages will be downloaded:

    package                    |            build
    ---------------------------|-----------------
    numpy-1.20.3               |   py39h749eb61_1          10 KB
    numpy-base-1.20.3          |   py39h5bfbeaa_1         4.2 MB
    python-3.9.21              |       h8205438_1        19.6 MB
    ------------------------------------------------------------
                                           Total:        23.8 MB

The following NEW packages will be INSTALLED:

  blas               pkgs/main/win-64::blas-1.0-mkl
  intel-openmp       pkgs/main/win-64::intel-openmp-2023.1.0-h59b6b97_46320
```

Figure 3.4 Install numpy in conda environment



```
(base) C:\Users\User>conda activate patrick

(patrick) C:\Users\User>conda install tensorflow-gpu
Channels:
 - defaults
Platform: win-64
Collecting package metadata (repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: C:\Users\User\anaconda3\envs\patrick

  added / updated specs:
    - tensorflow-gpu


The following packages will be downloaded:

    package                    |            build
    ---------------------------|-----------------
    _tflow_select-2.1.0        |              gpu           3 KB
    abseil-cpp-20210324.2      |       hd77b12b_0         1.6 MB
    absl-py-2.1.0              |   py39haa95532_0         191 KB
    aiohappyeyeballs-2.4.4     |   py39haa95532_0          24 KB
    aiohttp-3.11.10            |   py39h827c3e9_0         779 KB
    astor-0.8.1                |   py39haa95532_0          47 KB
    astunparse-1.6.3           |             py_0          17 KB
    async-timeout-5.0.1        |   py39haa95532_0          14 KB
    attrs-24.3.0               |   py39haa95532_0         153 KB
```

Figure 3.5 Install tensorflow in conda environment



```
done

(patrick) C:\Users\User>conda install scikit-learn
```

Figure 3.6 Install scikit-learn in conda environment

3.2.3 Implementation Steps

Based on the architecture we introduced on section 3.1, the pseudo code of this algorithm is presented as below.

Algorithm 1 Efficiency-based GP to generate loss function

```
1: Initialize: $P, T, G = P * 3/4$, $N = G/2$, $C_r = 0.5$, $M_{ST} = 0.3$, $M_N = 0.1$
2: Randomly initialize population which include P trees
3: Evaluate the fitnesses for each individual in the population
4: Determine the best individual
5: while generation number < $T$ do
6:     $S$ = Sample tournament G $\sim$ Uniform (P)
7:     Select top $N$ trees from $S$ to form a set $X$
8:     for Individual in $X$ do
9:         if $rand_1 < C_r$ then
10:            random_individual = Randomly select a individual in $X$ except Individual
11:        else
12:            random_individual = Randomly generated tree
13:        end if
14:        generated_child = Crossover(Individual, random_individual)
15:        if $rand_2 < M_{ST}$ then
16:            Apply subtree mutation to the generated_child
17:        end if
18:        if $rand_3 < M_N$ then
19:            Apply one-point mutation to the generated_child
20:        end if
21:        Evaluate the fitness for generated_child
22:        Append generated_child to the population
23:    end for
24:    Update the best individual
25:    Select $P$ best trees from population to form a new population containing $P$ trees
26: end while
27: return Best individual in the population
```

Initially, we will initialize all the necessary parameters (line 1). P represents the number of individuals that can exist within a population. T denotes the number of generations our experiment will go through. G indicates the number of individuals selected from the population for comparison in a single sample tournament, from which the top-performing N individuals will be chosen for genetic operations. C_r is the probability that an individual will select different types of crossover methods (lines 9-13). M_{ST} represents the probability that an individual will apply subtree mutation (lines 15-17). M_N denotes the probability that an individual will apply one-point mutation (lines 18-20).

Next, before officially starting, we will generate P trees (line 2). Each generated tree will have a height between 2 and 100. The leaf node will be randomly selected from real numbers,

y_{pred} or y_{true}. The root and other nodes will be randomly chosen from the set of operations: $+$, $-$, $*$, $/$, $*(-1)$, $\sqrt{}$, $\log$, $\exp$, $\sin$, or $\cos$. These trees, as depicted in the right-side diagram of figure 2.3, each represent a unique loss function. It is important to note that the tree must include at least one instance of y_{pred} and y_{true}, and the loss value produced by the generated loss function should fall within a reasonable numerical range, specifically between 10^{-5} and 10^{5}. If the resulting loss value lies outside this predefined interval, either being excessively small or excessively large, we consider the loss function to be numerically unstable or unsuitable for training. In such cases, the loss function is discarded and replaced with a newly generated one in order to preserve the validity and reliability of the optimization process. Subsequently, we will use the evaluation method from *Next Generation Loss Function for Image Classification* [5] to evaluate each tree in the population (line 3) and determine the best individual (line 4).

The purpose of this paper is to accelerate the execution speed of the algorithm which can generate a decent loss function. Hence, we introduce the concept of a sample tournament (line 6). By employing random sampling, we reduce the number of newly generated individuals. Consequently, this reduction decreases the number of evaluations required, thereby shortening the overall execution time of the algorithm. From the set S generated by the sample tournament, we select the top-performing N individuals (line 7) for genetic operations and evaluation.

Before performing crossover, we will determine how to choose the second parent for the crossover: we first randomly generate a decimal number between 0 and 1. If this number is less than C_r, we will randomly select another individual from set X to act as the parent. Otherwise, a new individual will be randomly generated to act as the parent for this crossover (lines 9-13). The logic of the crossover (line 14) is as follows: we will select a node from both the individual and the $random\_individual$, and treat the node from the $random\_individual$ as the root to produce a subtree. Then, we will also pick a random node from the individual as another root, delete this root and its subtree, and replace it with the subtree from $random\_individual$, thereby completing the crossover.

We utilize the $generated\_child$ produced during the previous crossover process to perform mutation. Mutation is divided into two parts: subtree mutation (line 16) and one-point mutation (line 19). Before starting the mutation process, we use random probabilities, similar to the crossover process, to determine whether to perform subtree mutation (line 15) or one-point mutation (line 18). The logic of both methods is fundamentally similar. For one-point mutation, a node other than the root is randomly selected from the generated_child. This selected node is then randomly replaced with one of the following operators: $+$, $-$, $*$, $/$, $*(-1)$, $\sqrt{}$, $\log$, $\exp$, $\sin$, or $\cos$. This process completes the one-point mutation. The logic of subtree mutation is similar to that of one-point mutation. However, instead of merely modifying the selected node, the node along with its subtree is deleted. It is then replaced with a newly generated random subtree, thereby completing the subtree mutation.

Following the sample tournament, crossover, and mutation processes, we evaluate (line 21) the newly generated individual and append it to the population (line 22). At the end of each generation, we select the top-performing P individuals and eliminate the others to maintain a constant population size (line 25). After all generations have concluded, we obtain and return the best-performing individual (line 27).

Chapter 4 Result & Analysis

4.1 Experiment

4.1.1 Experiment Method

In our experimental design, two distinct configurations were employed: a small-scale setting and a large-scale setting. The primary differences between these settings are the number of training epochs and the generation number which denotes the number of generations executed within the genetic programming process.

As shown in table 4.1, in the small-scale experiment, we set the generation parameter to 4 and the population parameter to 8. Each dataset was trained for 6 epochs. In contrast, in the large-scale experiment, we configured the generation parameter to 12 while keeping the population parameter at 8, and each dataset was trained for 10 epochs.

Table 4.1 Algorithm parameter settings

	Population	Training epochs	Generation number
Small scale	8	6	4
Large scale	8	10	12

The experimental procedure is illustrated in algorithm 2. Initially, we trained three new model for three different datasets by using cross-entropy as the loss function and stored loss values. Subsequently, we initialized the required population for our experiment and stored it accordingly. Once the population was initialized, we created an exact copy of it. This duplicated population was then used to execute the NGL algorithm, during which we recorded execution time along with other relevant data. Following this, we used the same initial population to execute our own algorithm, ensuring that execution time and other pertinent information were logged as well.

It is worth emphasizing that all code was executed in a single instance to minimize the influence of randomness on our results. Since random values vary between different executions, conducting all computations in a single run helps mitigate the impact of such randomness on the experiment. Additionally, by using the same population for both algorithms, we aimed to further reduce the variability introduced by random number generation.

Lastly, we would like to clarify that the starting point for time measurement was the exact moment immediately after the population was duplicated, rather than the moment when the population started initializing. This approach ensures a fair comparison of the computational time required for both algorithms to complete their respective executions.

Algorithm 2 Experiment procedure

```
1: Train three new model for three datasets by using Cross Entropy Loss and store the loss
   values.
2: Initialize population $P$
3: Copy population $P$ to $P_1$
4: Start record time $T_1$
5: Execute NGL using population $P_1$
6: Stop record time $T_1$ and output result for NGL
7: Start record time $T_2$
8: Execute our code using population $P$
9: Stop record time $T_2$ and output result for our code
```

4.1.2 Experiment Result

Before delving into the analysis of the experimental results, we first provide a detailed explanation of the evaluation criteria employed in this study. Among the various metrics considered, Average Improvement is designated as the primary evaluation standard. This metric, which has also been utilized in previous studies [5], serves as a key indicator for assessing the quality of the generated loss functions. Specifically, Average Improvement is computed by comparing the loss values obtained from the generated loss functions against those derived from the CE loss function. The rationale behind this comparison lies in the assumption that a superior loss function should yield lower loss values during training, thereby indicating better performance. A higher Average Improvement score thus signifies a more effective and higher-quality loss function, suggesting that the generated function leads to more favorable learning outcomes when used in training neural networks.

The procedure for computing the Average Improvement is briefly described in Algorithm 3. In addition, a small x-axis bias was introduced to each data point in figures 4.1 through 4.4 to improve visual clarity, as the ”win over generation” segments in these figures often exhibit overlapping elements.

Algorithm 3 Average Improvement Calculation Algorithm

```
 1: Input: CIFAR-10, FashionMNIST, CIFAR-100 datasets, CE loss values, newly generated
    loss function $L$.
 2: Train InceptionV3 model from scratch on these three datasets by loss function $L$ and store
    the corresponding loss values.
 3: Compare the loss values from the new model with the CE errors for each of the three
    datasets.
 4: if all three loss values of the new model are higher than their respective CE errors then
 5:     Calculate the average of negative improvements across all three datasets.
 6:     Return: Average of the negative improvements.
 7: else
 8:     Calculate the average of the improvement(s) for the dataset(s) where the loss value is
    lower than the CE error.
 9:     Return: Average improvement.
10: end if
```

4.1.2.1 Small Scale

Figure 4.1 illustrate the performance of our proposed algorithm in the small-scale setting. The figure depicts the evolution of the Average Improvement over the course of 4 generations, with each generation consisting of 8 individuals.

In the course of our experiment, it can be observed that not all individuals within the population undergo genetic operations during each generation. Instead, a selection mechanism based on sample tournament is employed, whereby only a subset of individuals, specifically the top 50% selected through the tournament process, are eligible to participate in genetic operations. As a result of this selective evolutionary strategy, improvements in fitness or performance are typically confined to a limited portion of the population in each generation. The remaining individuals either retain their existing performance levels or exhibit no significant change.

It is worth noting that individuals are not ranked at generation 0; therefore, evaluations of the best individual should be based on results starting from generation 1. Additionally, it can be observed that certain individuals occasionally exhibit a decrease in Average Improvement. This occurs because the final evaluation prioritizes the number of dataset wins as the primary criterion, with Average Improvement considered only subsequently. As a result, as shown in the lower part of figure 4.1, when the Average Improvement of an individual declines, the number of wins for that individual inevitably increases.

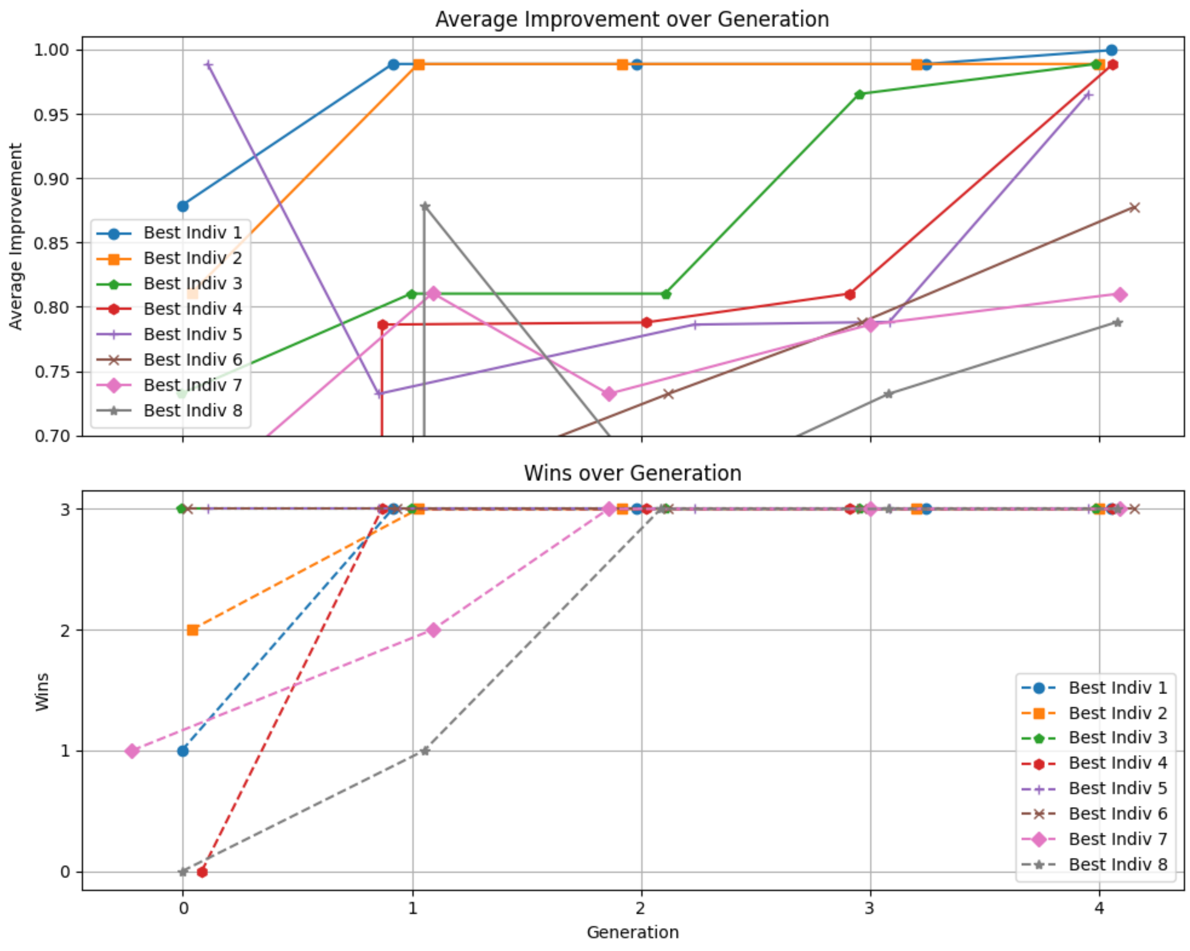


Figure 4.1 Small-scale result

4.1.2.2 Large Scale

Figure 4.2 presents the results of our algorithm in the large-scale setting. In this scenario, we increased the number of generations to 12 and the training epochs to 10, while maintaining a population size of 8 individuals.

In the large-scale setting, we observe a similar trend to that of the small-scale experiment, wherein the performance at the initial stages tends to exhibit a certain degree of instability. This early fluctuation is likely attributable to the randomness inherent in the initialization of individuals and the limited influence of genetic operations in the early generations. However, as the number of generations increases, a gradual and consistent improvement in the performance of individuals becomes evident across the population. This overall upward trajectory in individual fitness indicates that, despite the fact that not every individual is subjected to genetic operations in each generation, as was similarly observed in earlier, the evolutionary process remains effective over a longer timescale.

More specifically, the algorithm's design, which restricts genetic operations to a selected subset of individuals in each generation, does not impede the overall progress of the population when evaluated over a sufficiently large number of generations. This is due to the modified crossover mechanism, which enhances population diversity. This finding highlights the robustness of our approach in more realistic scenarios, where computational constraints or design considerations may prevent full-population genetic updates in every generation. The observed long-term improvements demonstrate that the evolutionary pressure introduced by selective operations and enhanced crossover diversity is sufficient to drive the population toward higher-quality solutions, even in large and diverse sampling conditions.

Furthermore, based on the above findings, we observe that in small-scale experiment, the result is highly dependent on the initial population values, as these initial values significantly influence the final experimental outcomes due to the limited evolutionary time. In contrast, in large-scale experiment, by increasing the number of generations, we can leverage a longer evolutionary process to minimize the impact of the initial population on the final results.

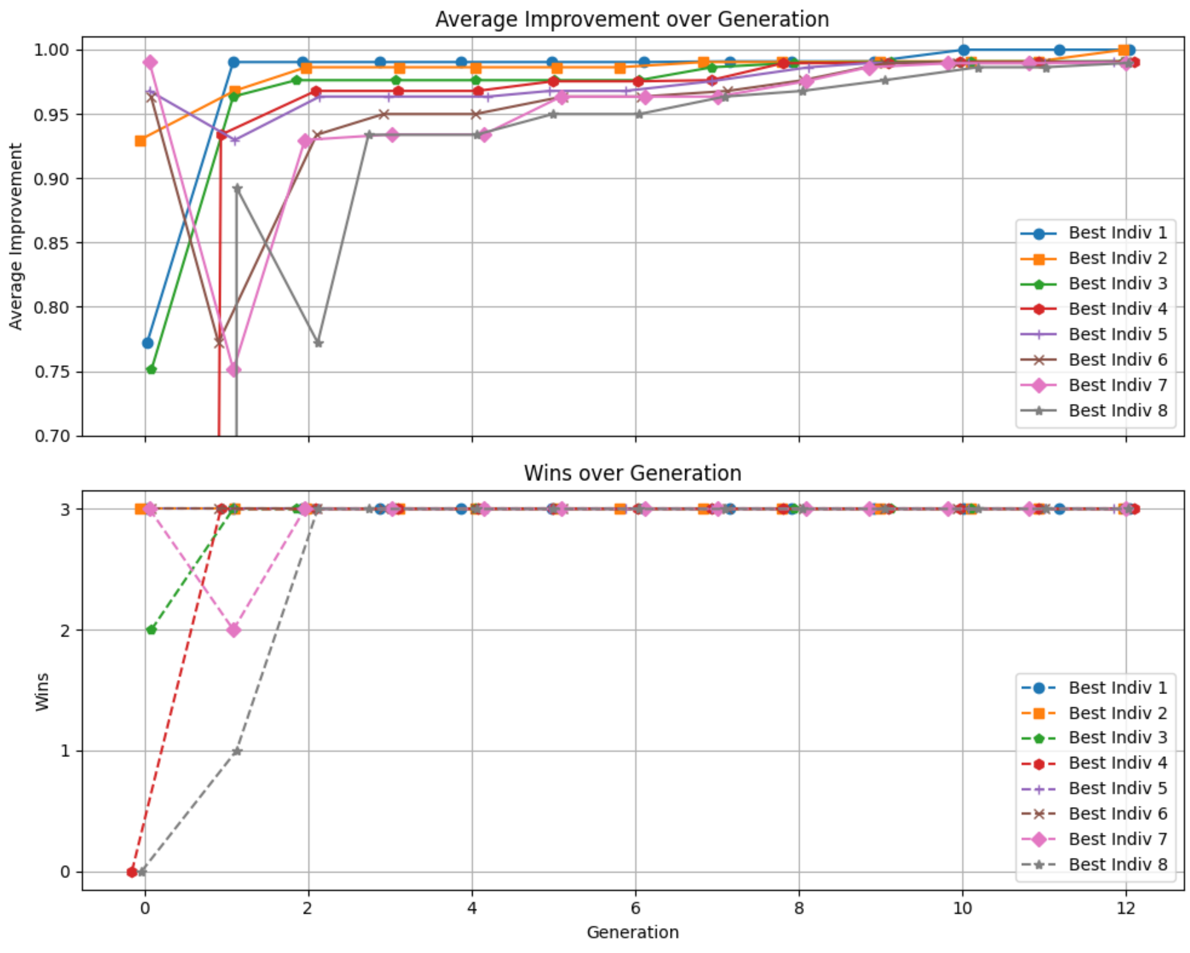


Figure 4.2 Large-scale result

4.2 Comparison

4.2.1 Next Generation Loss (NGL) Result

Figure 4.3 and figure 4.4 present the results of the NGL algorithm across two different scale sizes: small and large.

In the implementation of the NGL algorithm, a notable characteristic is that genetic operations are applied uniformly across the entire population in each generation. This comprehensive application of genetic operators to all individuals contributes to a more rapid convergence of the algorithm, regardless of whether it is evaluated in a small-scale or large-scale experimental setting. When compared with our proposed method, which selectively applies genetic operations to only a subset of individuals in each generation, NGL demonstrates a faster trajectory toward optimal or near-optimal solutions.

However, this accelerated convergence comes at a cost. Performing genetic operations on the entire population significantly increases the computational overhead per generation. In practice, this results in longer processing times and greater resource consumption, particularly as the population size or generation number scales. Consequently, while NGL benefits from quicker convergence in the early phases of training, it may be less efficient in terms of overall time-to-solution and computational scalability. In contrast, our method, by selectively applying genetic operations and thereby reducing per-generation complexity, offers a more computationally efficient alternative, especially in scenarios where long-term performance and scalability are of greater concern.

4.2.2 Compare with NGL

4.2.2.1 Small Scale

Figure 4.5 illustrate the comparative performance between our proposed algorithm and the NGL approach in terms of best individual in each generation within the small-scale experimental setting. Given the limited number of generations in this scenario, the differences between

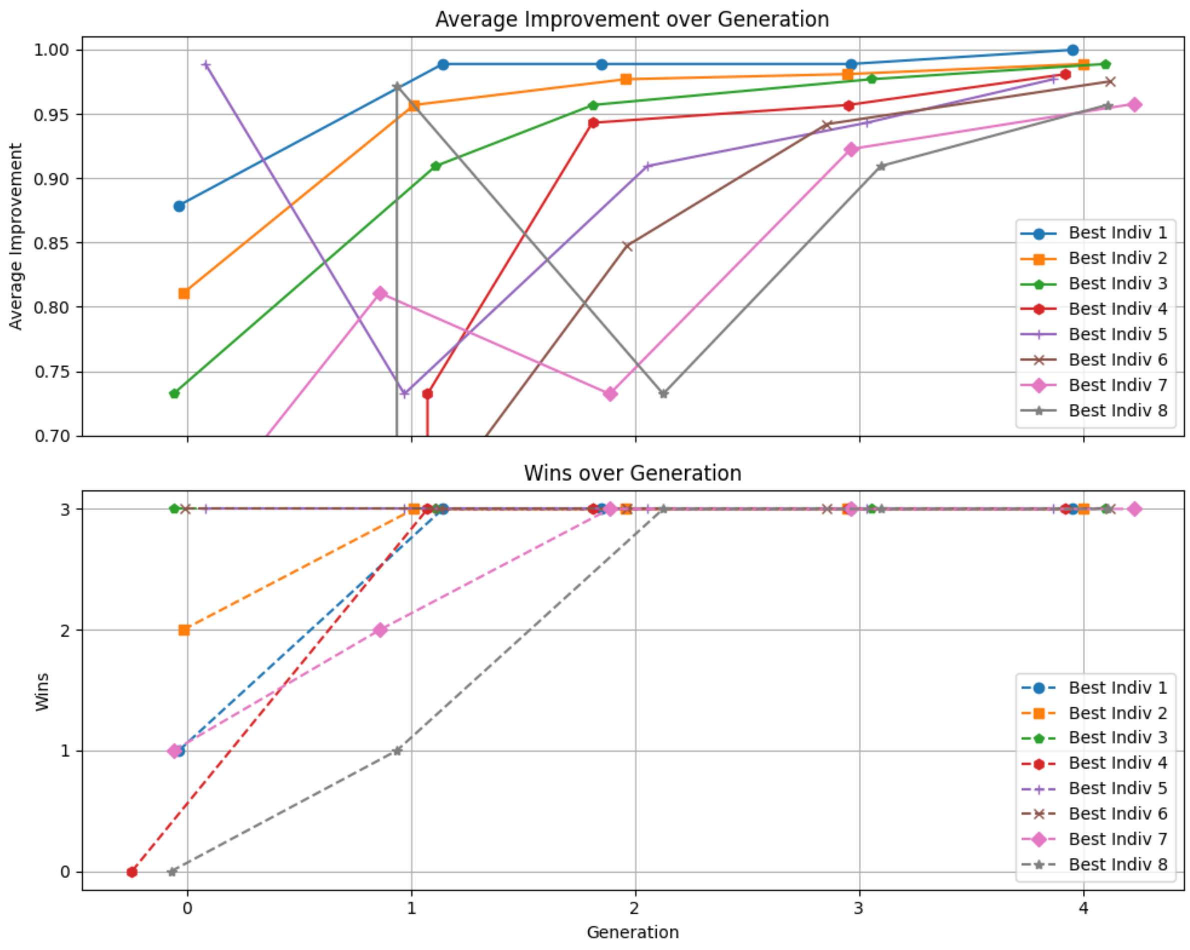


Figure 4.3 Small-scale result for NGL

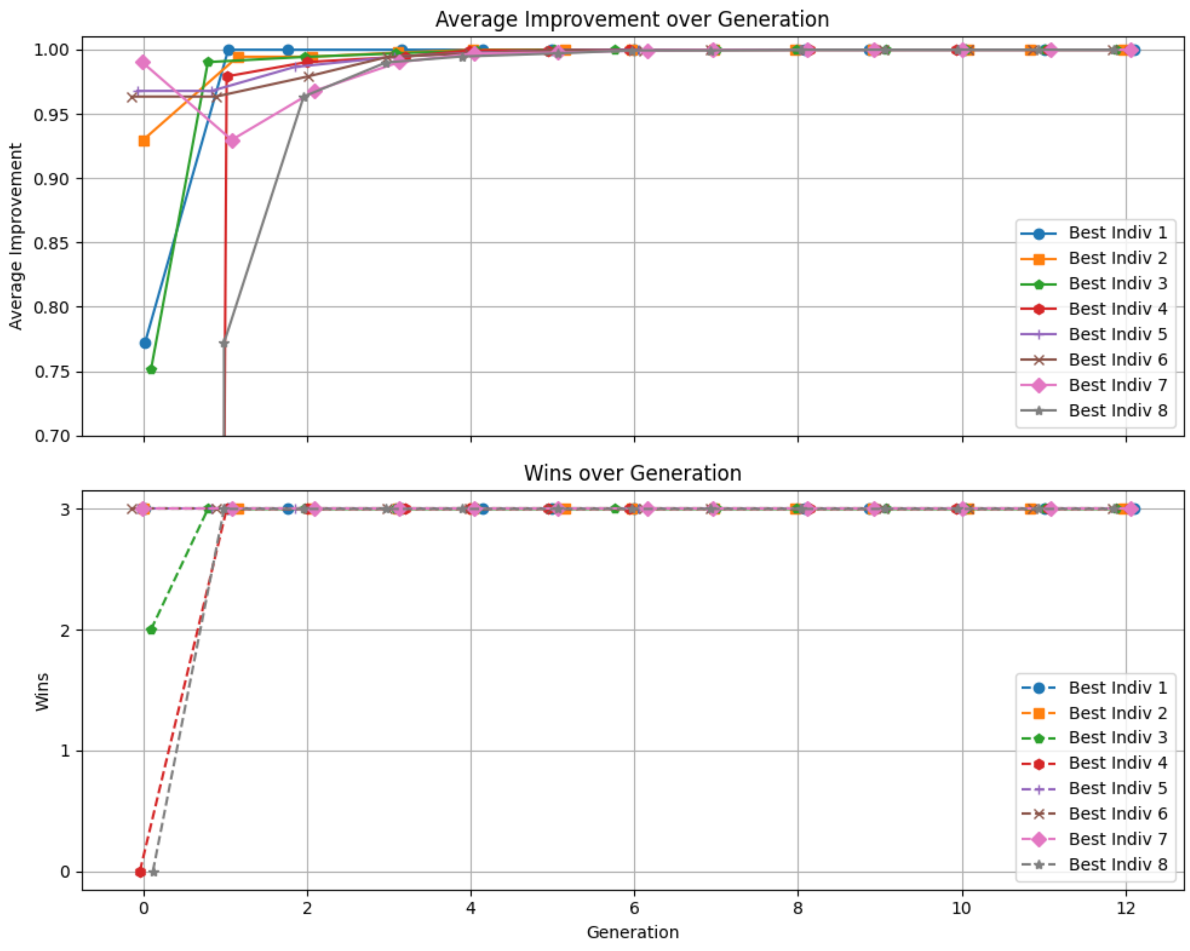


Figure 4.4 Large-scale result for NGL

the two methods become particularly apparent. Our algorithm, which employs a sample tournament selection mechanism to determine which individuals are eligible to undergo genetic operations, inherently applies these evolutionary processes to only a subset of the population in each generation. As a result, the potential for significant performance improvement within a short generational span is constrained. In contrast, NGL, by applying genetic operations to the entire population, demonstrates faster initial convergence under these same conditions. As shown in table 4.2, our algorithm outperforms NGL in the CIFAR 100 dataset, while in the other two datasets, our algorithm performs worse than NGL. This is likely due to the limited number of generations available for optimization in the small-scale setting.

However, it is important to emphasize a key advantage of our method: computational efficiency. As shown in Table 4.4, while the extent of performance improvement may be less pronounced in the early generations due to the selective nature of genetic operations, our approach achieves a significant reduction in execution time by 64%. This gain in efficiency becomes particularly valuable when deploying such algorithms in real-world scenarios characterized by limited computational resources or strict time constraints. The experimental results, therefore, indicate a trade-off between rapid convergence and computational cost, with our algorithm prioritizing long-term efficiency and scalability over immediate performance gains in small-scale, short-horizon settings.

Table 4.2 Model accuracy on small-scale setting

Dataset	NGL	Ours
CIFAR 10	0.0239	0.0149
FashionMNIST	0.001	9.999e-05
CIFAR 100	0.0013	0.0062

4.2.2.2 Large Scale

As illustrated in figure 4.6, the performance of our proposed algorithm under the large-scale setting demonstrates several notable strengths. Due to the increased number of generations available for performing genetic operations and the enhanced diversity in crossover execution, the algorithm benefits from an extended evolutionary horizon, allowing for more meaningful

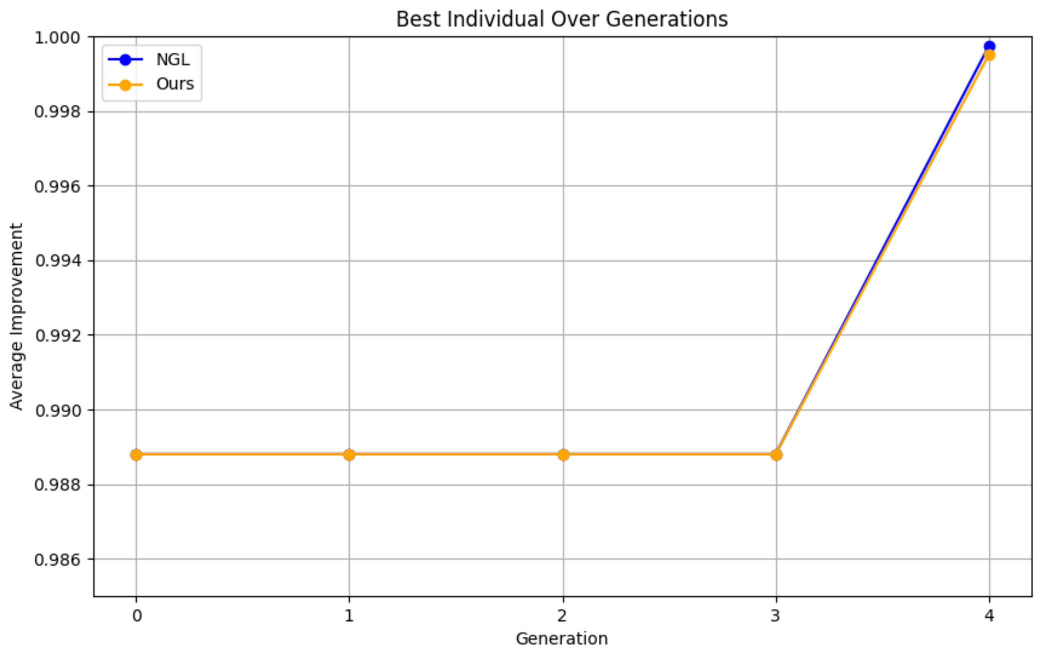


Figure 4.5 Small-scale result comparison

optimization over time. This prolonged generational window enables the algorithm to systematically refine the quality of generated loss functions through iterative selection and variation. As shown in table 4.3, we observe that our method is able to outperform NGL on certain datasets. For the remaining datasets, although our method does not surpass the original NGL results, it is nevertheless able to achieve performance levels that are closely comparable to those reported in the original study.

In terms of computational efficiency, our approach offers a significant advantage. By incorporating a sample tournament selection strategy, we effectively reduce the number of individuals that undergo genetic operations in each generation, which in turn minimizes the overall computational burden when evaluating them. This design choice proves especially beneficial in the large-scale context, where the cost of evaluating the entire population can become prohibitive. Empirical results show that, even while maintaining a comparable level of loss function quality to that of NGL, our algorithm is able to reduce execution time by nearly 63%, as shown in table 4.4. This substantial improvement in runtime, coupled with competitive performance outcomes, highlights the practical utility of our approach, especially in scenarios where computational efficiency is a critical consideration.

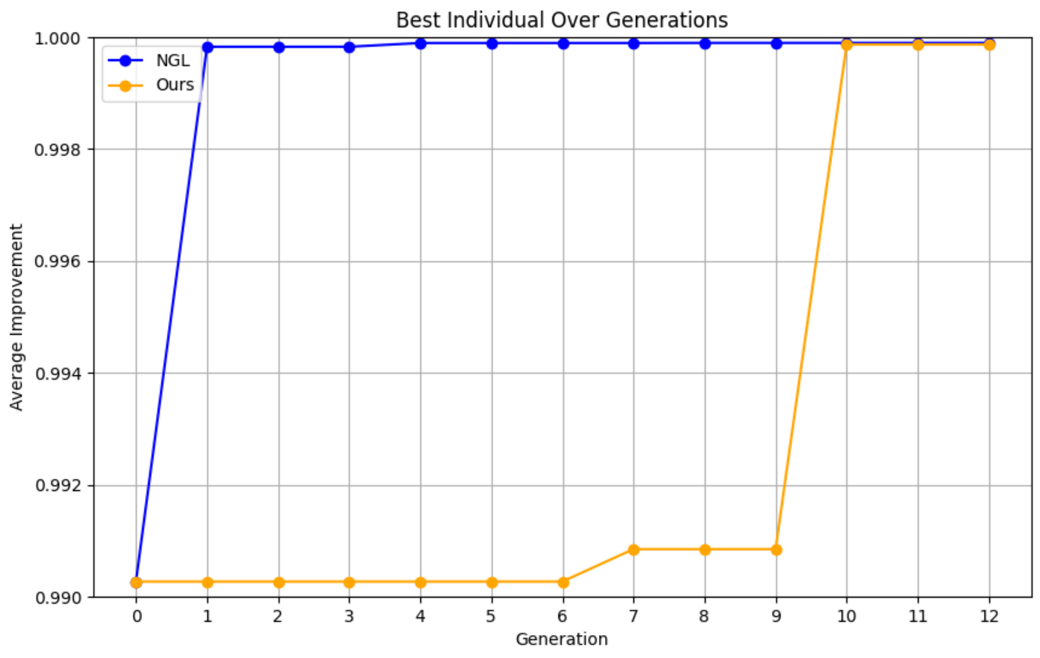


Figure 4.6 Large-scale result comparison

Table 4.3 Model accuracy on large-scale setting

Dataset	NGL	Ours
CIFAR 10	0.7046	0.7953
FashionMNIST	0.9074	0.8988
CIFAR 100	0.0619	0.1376

Table 4.4 Algorithm execution time

Experiment Scale	NGL	Ours
Small scale	173286.98 seconds	62363.94 seconds
Large scale	1013738.83 seconds	426915.20 seconds

4.3 Discussion

During the course of our experimental investigation, we identified several noteworthy observations that merit further discussion within this section.

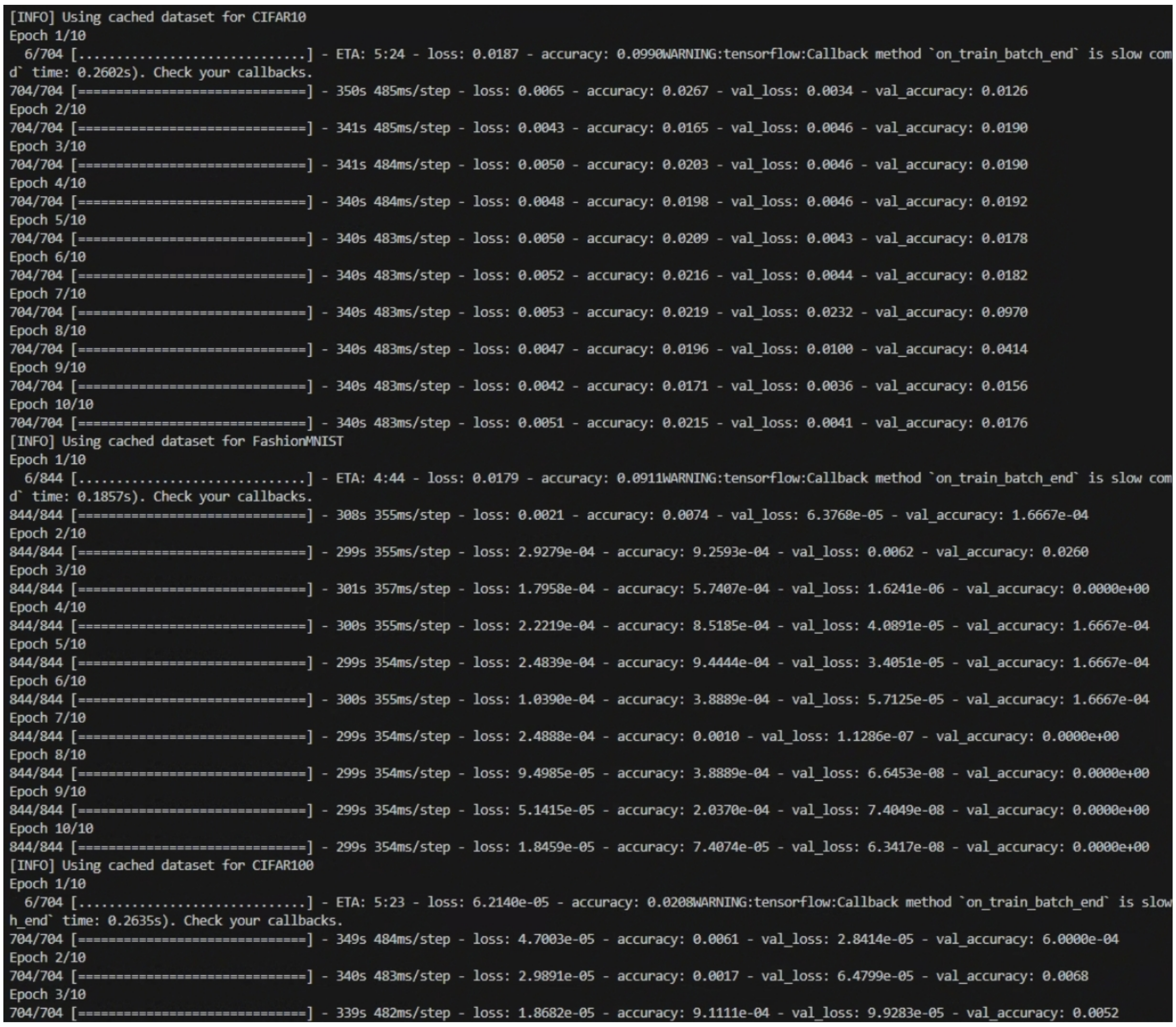
As mentioned at the beginning of Section 4.1.2, the principal criterion for evaluating the effectiveness of a loss function in our study is the Average Improvement relative to CE loss. This metric quantifies the degree to which a given loss function reduces the loss value in comparison to CE, and serves as a proxy for its potential utility in training neural networks. However, a significant challenge arises from the nature of the generated loss functions themselves: since these functions are generated randomly by the system, they often defy intuitive interpretation based on established principles of neural network training. In other words, the mathematical forms of many of the generated loss functions do not lend themselves to human-understandable explanations or conventional analytical reasoning.

One of the phenomena we encountered is that certain randomly generated loss functions can produce very low loss values during training, yet the resulting models perform poorly in terms of accuracy—sometimes significantly worse than expected (as shown in the figure 4.7). This is clearly illustrated in the experimental results, where models trained with such loss functions, despite exhibiting similar levels of Average Improvement to those reported in the original NGL paper, nonetheless show substantially lower accuracy across various datasets in the small-scale setting. This discrepancy highlights a critical limitation of using loss value alone as the sole measure of loss function quality.

As shown in figure 4.8, in one of the small-scale experimental trials, a high-quality loss function, capable of yielding strong model performance, was unfortunately excluded from further selection. This occurred because another loss function exhibited a numerically lower loss value, which, under our current evaluation criterion based on Average Improvement, was interpreted as superior. As discussed previously, Average Improvement serves as the primary metric for determining the effectiveness of a loss function by comparing its loss value against that of the standard CE loss.

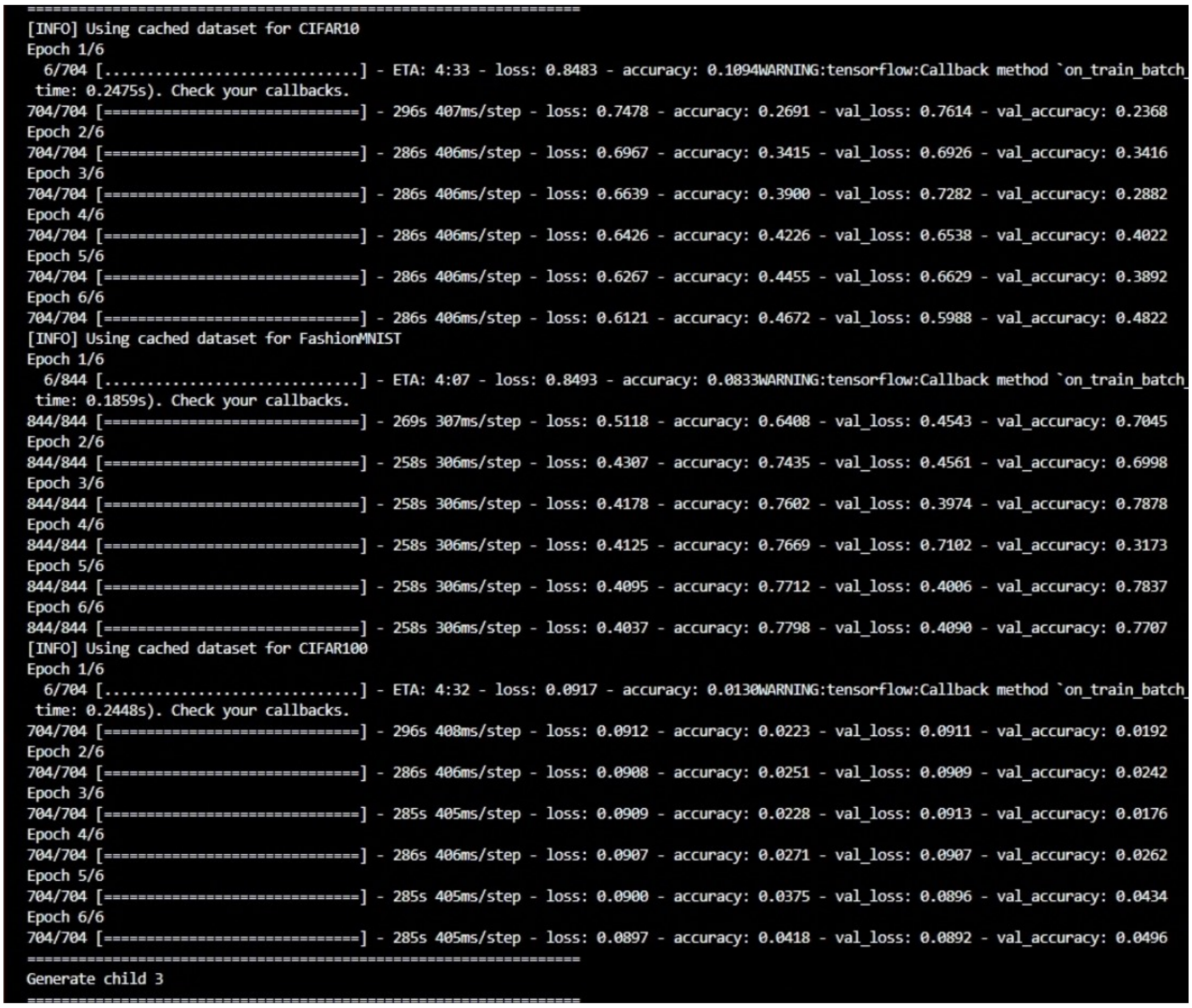
Unfortunately, in this particular case, the loss function that was ultimately selected, despite having a lower loss value, resulted in a model with significantly poorer accuracy. Consequently, the overall experimental outcome was suboptimal, as the evaluation mechanism favored a function that performed worse in terms of actual predictive capability.

There are several possible explanations for why these anomalously low loss values occur.



```
[INFO] Using cached dataset for CIFAR10
Epoch 1/10
  6/704 [..............................] - ETA: 5:24 - loss: 0.0187 - accuracy: 0.0990WARNING:tensorflow:Callback method `on_train_batch_end` is slow com
d` time: 0.2602s). Check your callbacks.
704/704 [==============================] - 350s 485ms/step - loss: 0.0065 - accuracy: 0.0267 - val_loss: 0.0034 - val_accuracy: 0.0126
Epoch 2/10
704/704 [==============================] - 341s 485ms/step - loss: 0.0043 - accuracy: 0.0165 - val_loss: 0.0046 - val_accuracy: 0.0190
Epoch 3/10
704/704 [==============================] - 341s 484ms/step - loss: 0.0050 - accuracy: 0.0203 - val_loss: 0.0046 - val_accuracy: 0.0190
Epoch 4/10
704/704 [==============================] - 340s 484ms/step - loss: 0.0048 - accuracy: 0.0198 - val_loss: 0.0046 - val_accuracy: 0.0192
Epoch 5/10
704/704 [==============================] - 340s 483ms/step - loss: 0.0050 - accuracy: 0.0209 - val_loss: 0.0043 - val_accuracy: 0.0178
Epoch 6/10
704/704 [==============================] - 340s 483ms/step - loss: 0.0052 - accuracy: 0.0216 - val_loss: 0.0044 - val_accuracy: 0.0182
Epoch 7/10
704/704 [==============================] - 340s 483ms/step - loss: 0.0053 - accuracy: 0.0219 - val_loss: 0.0232 - val_accuracy: 0.0970
Epoch 8/10
704/704 [==============================] - 340s 483ms/step - loss: 0.0047 - accuracy: 0.0196 - val_loss: 0.0100 - val_accuracy: 0.0414
Epoch 9/10
704/704 [==============================] - 340s 483ms/step - loss: 0.0042 - accuracy: 0.0171 - val_loss: 0.0036 - val_accuracy: 0.0156
Epoch 10/10
704/704 [==============================] - 340s 483ms/step - loss: 0.0051 - accuracy: 0.0215 - val_loss: 0.0041 - val_accuracy: 0.0176
[INFO] Using cached dataset for FashionMNIST
Epoch 1/10
  6/844 [..............................] - ETA: 4:44 - loss: 0.0179 - accuracy: 0.0911WARNING:tensorflow:Callback method `on_train_batch_end` is slow com
d` time: 0.1857s). Check your callbacks.
844/844 [==============================] - 308s 355ms/step - loss: 0.0021 - accuracy: 0.0074 - val_loss: 6.3768e-05 - val_accuracy: 1.6667e-04
Epoch 2/10
844/844 [==============================] - 299s 355ms/step - loss: 2.9279e-04 - accuracy: 9.2593e-04 - val_loss: 0.0062 - val_accuracy: 0.0260
Epoch 3/10
844/844 [==============================] - 301s 357ms/step - loss: 1.7958e-04 - accuracy: 5.7407e-04 - val_loss: 1.6241e-06 - val_accuracy: 0.0000e+00
Epoch 4/10
844/844 [==============================] - 300s 355ms/step - loss: 2.2219e-04 - accuracy: 8.5185e-04 - val_loss: 4.0891e-05 - val_accuracy: 1.6667e-04
Epoch 5/10
844/844 [==============================] - 299s 354ms/step - loss: 2.4839e-04 - accuracy: 9.4444e-04 - val_loss: 3.4051e-05 - val_accuracy: 1.6667e-04
Epoch 6/10
844/844 [==============================] - 300s 355ms/step - loss: 1.0390e-04 - accuracy: 3.8889e-04 - val_loss: 5.7125e-05 - val_accuracy: 1.6667e-04
Epoch 7/10
844/844 [==============================] - 299s 354ms/step - loss: 2.4888e-04 - accuracy: 0.0010 - val_loss: 1.1286e-07 - val_accuracy: 0.0000e+00
Epoch 8/10
844/844 [==============================] - 299s 354ms/step - loss: 9.4985e-05 - accuracy: 3.8889e-04 - val_loss: 6.6453e-08 - val_accuracy: 0.0000e+00
Epoch 9/10
844/844 [==============================] - 299s 354ms/step - loss: 5.1415e-05 - accuracy: 2.0370e-04 - val_loss: 7.4049e-08 - val_accuracy: 0.0000e+00
Epoch 10/10
844/844 [==============================] - 299s 354ms/step - loss: 1.8459e-05 - accuracy: 7.4074e-05 - val_loss: 6.3417e-08 - val_accuracy: 0.0000e+00
[INFO] Using cached dataset for CIFAR100
Epoch 1/10
  6/704 [..............................] - ETA: 5:23 - loss: 6.2140e-05 - accuracy: 0.0208WARNING:tensorflow:Callback method `on_train_batch_end` is slow
h_end` time: 0.2635s). Check your callbacks.
704/704 [==============================] - 349s 484ms/step - loss: 4.7003e-05 - accuracy: 0.0061 - val_loss: 2.8414e-05 - val_accuracy: 6.0000e-04
Epoch 2/10
704/704 [==============================] - 340s 483ms/step - loss: 2.9891e-05 - accuracy: 0.0017 - val_loss: 6.4799e-05 - val_accuracy: 0.0068
Epoch 3/10
704/704 [==============================] - 339s 482ms/step - loss: 1.8682e-05 - accuracy: 9.1111e-04 - val_loss: 9.9283e-05 - val_accuracy: 0.0052
```

Figure 4.7 Low loss value but low model accuracy



```
=================================================================
[INFO] Using cached dataset for CIFAR10
Epoch 1/6
  6/704 [..............................] - ETA: 4:33 - loss: 0.8483 - accuracy: 0.1094WARNING:tensorflow:Callback method `on_train_batch_
 time: 0.2475s). Check your callbacks.
704/704 [==============================] - 296s 407ms/step - loss: 0.7478 - accuracy: 0.2691 - val_loss: 0.7614 - val_accuracy: 0.2368
Epoch 2/6
704/704 [==============================] - 286s 406ms/step - loss: 0.6967 - accuracy: 0.3415 - val_loss: 0.6926 - val_accuracy: 0.3416
Epoch 3/6
704/704 [==============================] - 286s 406ms/step - loss: 0.6639 - accuracy: 0.3900 - val_loss: 0.7282 - val_accuracy: 0.2882
Epoch 4/6
704/704 [==============================] - 286s 406ms/step - loss: 0.6426 - accuracy: 0.4226 - val_loss: 0.6538 - val_accuracy: 0.4022
Epoch 5/6
704/704 [==============================] - 286s 406ms/step - loss: 0.6267 - accuracy: 0.4455 - val_loss: 0.6629 - val_accuracy: 0.3892
Epoch 6/6
704/704 [==============================] - 286s 406ms/step - loss: 0.6121 - accuracy: 0.4672 - val_loss: 0.5988 - val_accuracy: 0.4822
[INFO] Using cached dataset for FashionMNIST
Epoch 1/6
  6/844 [..............................] - ETA: 4:07 - loss: 0.8493 - accuracy: 0.0833WARNING:tensorflow:Callback method `on_train_batch_
 time: 0.1859s). Check your callbacks.
844/844 [==============================] - 269s 307ms/step - loss: 0.5118 - accuracy: 0.6408 - val_loss: 0.4543 - val_accuracy: 0.7045
Epoch 2/6
844/844 [==============================] - 258s 306ms/step - loss: 0.4307 - accuracy: 0.7435 - val_loss: 0.4561 - val_accuracy: 0.6998
Epoch 3/6
844/844 [==============================] - 258s 306ms/step - loss: 0.4178 - accuracy: 0.7602 - val_loss: 0.3974 - val_accuracy: 0.7878
Epoch 4/6
844/844 [==============================] - 258s 306ms/step - loss: 0.4125 - accuracy: 0.7669 - val_loss: 0.7102 - val_accuracy: 0.3173
Epoch 5/6
844/844 [==============================] - 258s 306ms/step - loss: 0.4095 - accuracy: 0.7712 - val_loss: 0.4006 - val_accuracy: 0.7837
Epoch 6/6
844/844 [==============================] - 258s 306ms/step - loss: 0.4037 - accuracy: 0.7798 - val_loss: 0.4090 - val_accuracy: 0.7707
[INFO] Using cached dataset for CIFAR100
Epoch 1/6
  6/704 [..............................] - ETA: 4:32 - loss: 0.0917 - accuracy: 0.0130WARNING:tensorflow:Callback method `on_train_batch_
 time: 0.2448s). Check your callbacks.
704/704 [==============================] - 296s 408ms/step - loss: 0.0912 - accuracy: 0.0223 - val_loss: 0.0911 - val_accuracy: 0.0192
Epoch 2/6
704/704 [==============================] - 286s 406ms/step - loss: 0.0908 - accuracy: 0.0251 - val_loss: 0.0909 - val_accuracy: 0.0242
Epoch 3/6
704/704 [==============================] - 285s 405ms/step - loss: 0.0909 - accuracy: 0.0228 - val_loss: 0.0913 - val_accuracy: 0.0176
Epoch 4/6
704/704 [==============================] - 286s 406ms/step - loss: 0.0907 - accuracy: 0.0271 - val_loss: 0.0907 - val_accuracy: 0.0262
Epoch 5/6
704/704 [==============================] - 285s 405ms/step - loss: 0.0900 - accuracy: 0.0375 - val_loss: 0.0896 - val_accuracy: 0.0434
Epoch 6/6
704/704 [==============================] - 285s 405ms/step - loss: 0.0897 - accuracy: 0.0418 - val_loss: 0.0892 - val_accuracy: 0.0496
=================================================================
Generate child 3
=================================================================
```

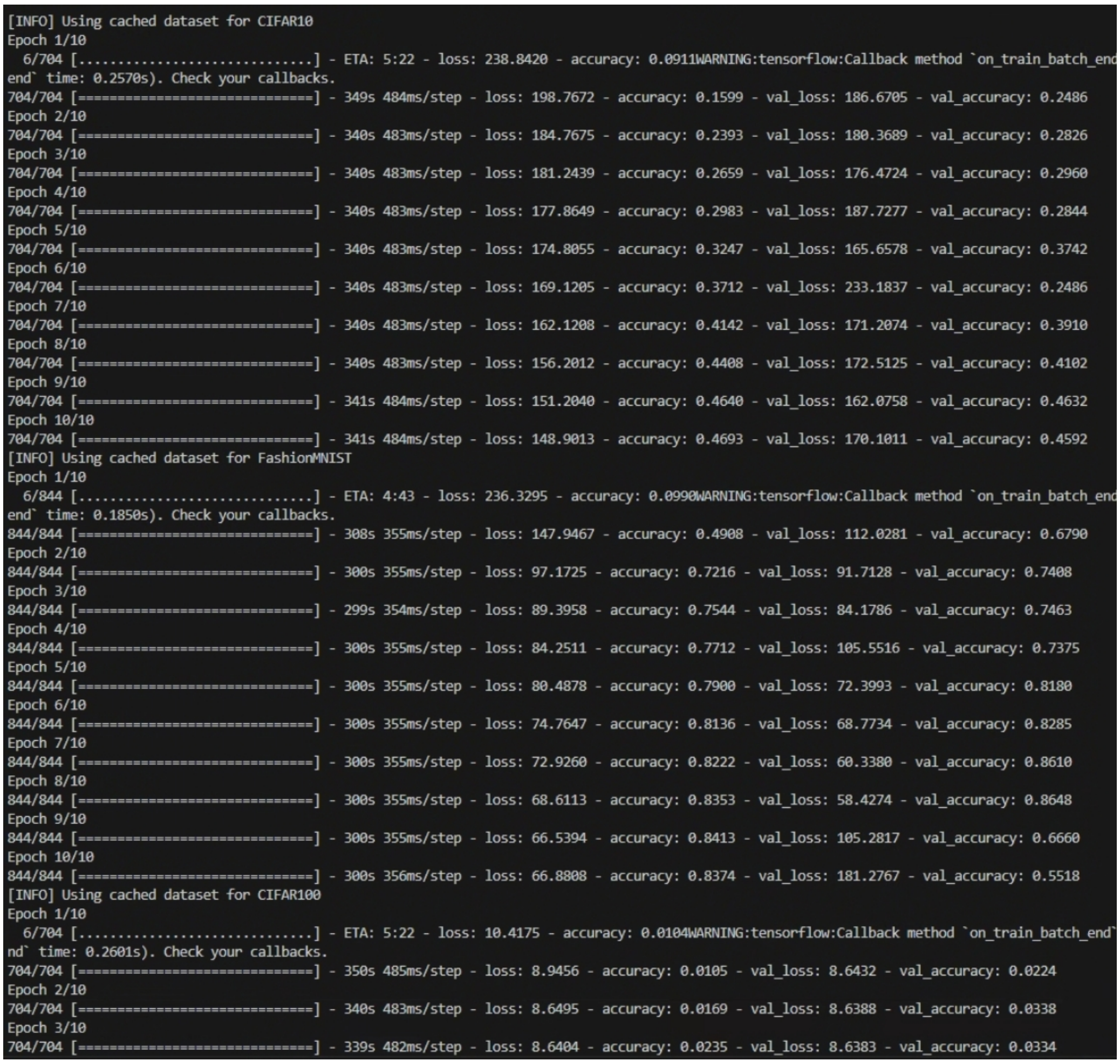
Figure 4.8 Good loss function but failed to be selected

In some cases, the loss function may be dominated by a very small multiplicative constant, effectively scaling down the overall value of the loss without meaningfully guiding the optimization process. In other cases, large constant terms may be subtracted from the loss expression, artificially lowering the numerical loss value while simultaneously impairing the model's ability to learn discriminative features. These types of manipulations can result from the random generation process and may lead to misleadingly low loss values that do not reflect genuine training efficacy.

Another intriguing phenomenon observed during our experiments pertains to the occasional emergence of loss functions that produce unusually large loss values during training, yet surprisingly result in models that achieve higher-than-expected accuracy levels, as illustrated in the figure 4.9. This counterintuitive outcome stands in stark contrast to the issue previously discussed, where low loss values did not necessarily correspond to improved model performance.

We hypothesize that this discrepancy may stem from a similar structural artifact introduced during the random generation process of loss functions—specifically, the unintended inclusion of large multiplicative constants. In this scenario, a loss function might be scaled by a disproportionately large constant, thereby inflating its numerical loss value without adversely affecting the result.

Given these findings, we acknowledge the limitations of the current evaluation metric and propose to explore alternative or complementary strategies for assessing loss function quality in our future work.



```
[INFO] Using cached dataset for CIFAR10
Epoch 1/10
  6/704 [..............................] - ETA: 5:22 - loss: 238.8420 - accuracy: 0.0911WARNING:tensorflow:Callback method `on_train_batch_end
end` time: 0.2570s). Check your callbacks.
704/704 [==============================] - 349s 484ms/step - loss: 198.7672 - accuracy: 0.1599 - val_loss: 186.6705 - val_accuracy: 0.2486
Epoch 2/10
704/704 [==============================] - 340s 483ms/step - loss: 184.7675 - accuracy: 0.2393 - val_loss: 180.3689 - val_accuracy: 0.2826
Epoch 3/10
704/704 [==============================] - 340s 483ms/step - loss: 181.2439 - accuracy: 0.2659 - val_loss: 176.4724 - val_accuracy: 0.2960
Epoch 4/10
704/704 [==============================] - 340s 483ms/step - loss: 177.8649 - accuracy: 0.2983 - val_loss: 187.7277 - val_accuracy: 0.2844
Epoch 5/10
704/704 [==============================] - 340s 483ms/step - loss: 174.8055 - accuracy: 0.3247 - val_loss: 165.6578 - val_accuracy: 0.3742
Epoch 6/10
704/704 [==============================] - 340s 483ms/step - loss: 169.1205 - accuracy: 0.3712 - val_loss: 233.1837 - val_accuracy: 0.2486
Epoch 7/10
704/704 [==============================] - 340s 483ms/step - loss: 162.1208 - accuracy: 0.4142 - val_loss: 171.2074 - val_accuracy: 0.3910
Epoch 8/10
704/704 [==============================] - 340s 483ms/step - loss: 156.2012 - accuracy: 0.4408 - val_loss: 172.5125 - val_accuracy: 0.4102
Epoch 9/10
704/704 [==============================] - 341s 484ms/step - loss: 151.2040 - accuracy: 0.4640 - val_loss: 162.0758 - val_accuracy: 0.4632
Epoch 10/10
704/704 [==============================] - 341s 484ms/step - loss: 148.9013 - accuracy: 0.4693 - val_loss: 170.1011 - val_accuracy: 0.4592
[INFO] Using cached dataset for FashionMNIST
Epoch 1/10
  6/844 [..............................] - ETA: 4:43 - loss: 236.3295 - accuracy: 0.0990WARNING:tensorflow:Callback method `on_train_batch_end
end` time: 0.1850s). Check your callbacks.
844/844 [==============================] - 308s 355ms/step - loss: 147.9467 - accuracy: 0.4908 - val_loss: 112.0281 - val_accuracy: 0.6790
Epoch 2/10
844/844 [==============================] - 300s 355ms/step - loss: 97.1725 - accuracy: 0.7216 - val_loss: 91.7128 - val_accuracy: 0.7408
Epoch 3/10
844/844 [==============================] - 299s 354ms/step - loss: 89.3958 - accuracy: 0.7544 - val_loss: 84.1786 - val_accuracy: 0.7463
Epoch 4/10
844/844 [==============================] - 300s 355ms/step - loss: 84.2511 - accuracy: 0.7712 - val_loss: 105.5516 - val_accuracy: 0.7375
Epoch 5/10
844/844 [==============================] - 300s 355ms/step - loss: 80.4878 - accuracy: 0.7900 - val_loss: 72.3993 - val_accuracy: 0.8180
Epoch 6/10
844/844 [==============================] - 300s 355ms/step - loss: 74.7647 - accuracy: 0.8136 - val_loss: 68.7734 - val_accuracy: 0.8285
Epoch 7/10
844/844 [==============================] - 300s 355ms/step - loss: 72.9260 - accuracy: 0.8222 - val_loss: 60.3380 - val_accuracy: 0.8610
Epoch 8/10
844/844 [==============================] - 300s 355ms/step - loss: 68.6113 - accuracy: 0.8353 - val_loss: 58.4274 - val_accuracy: 0.8648
Epoch 9/10
844/844 [==============================] - 300s 355ms/step - loss: 66.5394 - accuracy: 0.8413 - val_loss: 105.2817 - val_accuracy: 0.6660
Epoch 10/10
844/844 [==============================] - 300s 356ms/step - loss: 66.8808 - accuracy: 0.8374 - val_loss: 181.2767 - val_accuracy: 0.5518
[INFO] Using cached dataset for CIFAR100
Epoch 1/10
  6/704 [..............................] - ETA: 5:22 - loss: 10.4175 - accuracy: 0.0104WARNING:tensorflow:Callback method `on_train_batch_end`
nd` time: 0.2601s). Check your callbacks.
704/704 [==============================] - 350s 485ms/step - loss: 8.9456 - accuracy: 0.0105 - val_loss: 8.6432 - val_accuracy: 0.0224
Epoch 2/10
704/704 [==============================] - 340s 483ms/step - loss: 8.6495 - accuracy: 0.0169 - val_loss: 8.6388 - val_accuracy: 0.0338
Epoch 3/10
704/704 [==============================] - 339s 482ms/step - loss: 8.6404 - accuracy: 0.0235 - val_loss: 8.6383 - val_accuracy: 0.0334
```

Figure 4.9 High loss value but high model accuracy

Chapter 5 Conclusion & Future Work

5.1 Conclusion

Initially, we emphasized that existing methods often suffer from significant computational overhead due to various inherent limitations. To address this issue, we proposed a potential solution: instead of performing genetic operations on the entire population, we selectively sample a subset of individuals and further identify the top-performing individuals from this subset to perform genetic operations. Additionally, during the crossover phase, we refine the existing crossover mechanism to enhance the randomness of the proposed method. This approach strengthens the stochastic nature required for GP while minimizing deviations from the results of previous studies. By implementing this strategy, we effectively reduce execution time without significantly compromising the overall outcome.

Subsequently, in the experimental phase, to verify that our proposed method can achieve time savings under various scales while maintaining comparable performance to previous research, we conducted experiments across two scales: small and large. The experimental results demonstrate that as the scale of the experiment increases, the performance of our method increasingly aligns with that of previous approaches. In the large-scale setting, our method even achieves a 9% increase in model accuracy on CIFAR-10 and a 7% increase on CIFAR-100. Moreover, despite achieving comparable or even superior scores, our method significantly reduces execution time. Specifically, the time reductions achieved are 64% and 62.9% for small and large scales, respectively.

5.2 Future Work

In the discussion, we noted that loss functions generated randomly can sometimes lack intuitive interpretability. As a result, there are instances where a loss value is extremely low, yet the model accuracy is also significantly low. The occurrence of this issue poses a challenge when selecting loss functions based on average improvement, as it may lead to prioritizing loss functions of suboptimal quality. To address this concern in future research, a potential solution could involve incorporating a weighted evaluation approach, where both loss value and model accuracy contribute proportionally to the selection criteria. This adjustment may help mitigate the problem and ensure that the chosen loss functions effectively contribute to model performance.

Furthermore, an alternative method that may be considered for calculating the average improvement involves computing the mean across all observed improvement values, rather than selectively averaging only a subset of them. Through this averaging approach, it may become possible to fully leverage the informational content embedded within the loss values. By incorporating every available improvement data point into the calculation, this method has the potential to yield a more holistic and representative estimate of overall performance gains, ultimately contributing to a more nuanced and data-rich analytical outcome.

References

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[2] K. Janocha and W. M. Czarnecki, *On loss functions for deep neural networks in classification*, 2017. arXiv: 1702.05659 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1702.05659.

[3] L. Rosasco, E. D. Vito, A. Caponnetto, M. Piana, and A. Verri, "Are loss functions all the same?" *Neural Computation*, vol. 16, no. 5, pp. 1063–1076, 2004. DOI: 10.1162/089976604773135104.

[4] M. O' Neill, "Riccardo poli, william b. langdon, nicholas f. mcphee: A field guide to genetic programming," *Genetic Programming and Evolvable Machines*, vol. 10, no. 2, pp. 229–230, 2009.

[5] S. Akhmedova and N. Körber, *Next generation loss function for image classification*, 2024. arXiv: 2404.12948 [cs.CV]. [Online]. Available: https://arxiv.org/abs/2404.12948.

[6] X.-S. Yang, *Nature-inspired metaheuristic algorithms*. Luniver press, 2010.

[7] X.-S. Yang, "Metaheuristic optimization," *Scholarpedia*, vol. 6, no. 8, p. 11 472, 2011.

[8] W. Rawat and Z. Wang, "Deep convolutional neural networks for image classification: A comprehensive review," *Neural computation*, vol. 29, no. 9, pp. 2352–2449, 2017.

[9] H. Mühlenbein, M. Gorges-Schleuter, and O. Krämer, "Evolution algorithms in combinatorial optimization," *Parallel computing*, vol. 7, no. 1, pp. 65–85, 1988.

[10] M. Kumar, D. M. Husain, N. Upreti, and D. Gupta, "Genetic algorithm: Review and application," *Available at SSRN 3529843*, 2010.

[11] F. Glover, "Future paths for integer programming and links to artificial intelligence," *Computers & operations research*, vol. 13, no. 5, pp. 533–549, 1986.

[12] Wikipedia contributors, *Metaheuristic — Wikipedia, the free encyclopedia*, [Online; accessed 15-December-2024], 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Metaheuristic&oldid=1255593755.

[13] X. Yao, Y. Liu, and G. Lin, "Evolutionary programming made faster," *IEEE Transactions on Evolutionary computation*, vol. 3, no. 2, pp. 82–102, 1999.

[14] S. Das and P. N. Suganthan, "Differential evolution: A survey of the state-of-the-art," *IEEE transactions on evolutionary computation*, vol. 15, no. 1, pp. 4–31, 2010.

[15] D. B. Paul, "The selection of the "survival of the fittest"," *Journal of the History of Biology*, vol. 21, pp. 411–424, 1988.

[16] M. Dorigo, M. Birattari, and T. Stutzle, “Ant colony optimization,” *IEEE computational intelligence magazine*, vol. 1, no. 4, pp. 28–39, 2006.

[17] J. Kennedy and R. Eberhart, “Particle swarm optimization,” in *Proceedings of ICNN’95-international conference on neural networks*, ieee, vol. 4, 1995, pp. 1942–1948.

[18] J. D. Co-Reyes et al., *Evolving reinforcement learning algorithms*, 2022. arXiv: `2101.03958 [cs.LG]`. [Online]. Available: `https://arxiv.org/abs/2101.03958`.

[19] L. P. Kaelbling, M. L. Littman, and A. W. Moore, “Reinforcement learning: A survey,” *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[20] V. Thost and J. Chen, *Directed acyclic graph neural networks*, 2021. arXiv: `2101.07965 [cs.LG]`. [Online]. Available: `https://arxiv.org/abs/2101.07965`.

[21] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, “Regularized evolution for image classifier architecture search,” in *Proceedings of the aaai conference on artificial intelligence*, vol. 33, 2019, pp. 4780–4789.

[22] D. E. Goldberg and K. Deb, “A comparative analysis of selection schemes used in genetic algorithms,” in *Foundations of genetic algorithms*, vol. 1, Elsevier, 1991, pp. 69–93.

[23] Z. Zhang and M. Sabuncu, “Generalized cross entropy loss for training deep neural networks with noisy labels,” *Advances in neural information processing systems*, vol. 31, 2018.

[24] M. Iliadis, L. Spinoulas, and A. K. Katsaggelos, “Deep fully-connected networks for video compressive sensing,” *Digital Signal Processing*, vol. 72, pp. 9–18, 2018.

[25] J. Wu, “Introduction to convolutional neural networks,” *National Key Lab for Novel Software Technology. Nanjing University. China*, vol. 5, no. 23, p. 495, 2017.

[26] Z. Wang and A. C. Bovik, “Mean squared error: Love it or leave it? a new look at signal fidelity measures,” *IEEE signal processing magazine*, vol. 26, no. 1, pp. 98–117, 2009.

[27] U. Ruby and V. Yendapalli, “Binary cross entropy with deep learning technique for image classification,” *Int. J. Adv. Trends Comput. Sci. Eng*, vol. 9, no. 10, 2020.

[28] A. Plaquet and H. Bredin, “Powerset multi-class cross entropy loss for neural speaker diarization,” *arXiv preprint arXiv:2310.13025*, 2023.

[29] S. Gonzalez and R. Miikkulainen, *Improved training speed, accuracy, and data utilization through loss function optimization*, 2020. arXiv: `1905.11528 [cs.LG]`. [Online]. Available: `https://arxiv.org/abs/1905.11528`.

[30] A. Krizhevsky, V. Nair, and G. Hinton, “Cifar-10 (canadian institute for advanced research),” [Online]. Available: `http://www.cs.toronto.edu/~kriz/cifar.html`.

[31] A. Krizhevsky, V. Nair, and G. Hinton, “Cifar-100 (canadian institute for advanced research),” [Online]. Available: `http://www.cs.toronto.edu/~kriz/cifar.html`.

[32] H. Xiao, K. Rasul, and R. Vollgraf, *Fashion-mnist: A novel image dataset for benchmarking machine learning algorithms*, 2017. arXiv: `1708.07747 [cs.LG]`. [Online]. Available: `https://arxiv.org/abs/1708.07747`.

[33] O. Russakovsky et al., "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015. DOI: `10.1007/s11263-015-0816-y`.

[34] T. Sattler, B. Leibe, and L. Kobbelt, "Fast image-based localization using direct 2d-to-3d matching," in *2011 International Conference on Computer Vision*, IEEE, 2011, pp. 667–674.

[35] Y. He, C. Zhu, J. Wang, M. Savvides, and X. Zhang, *Bounding box regression with uncertainty for accurate object detection*, 2019. arXiv: `1809.08545 [cs.CV]`. [Online]. Available: `https://arxiv.org/abs/1809.08545`.

[36] Z.-Q. Zhao, P. Zheng, S.-t. Xu, and X. Wu, "Object detection with deep learning: A review," *IEEE transactions on neural networks and learning systems*, vol. 30, no. 11, pp. 3212–3232, 2019.

[37] B. Zhuang, C. Shen, M. Tan, L. Liu, and I. Reid, "Structured binary neural networks for accurate image classification and semantic segmentation," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 413–422.

[38] V. N. Murthy, V. Singh, T. Chen, R Manmatha, and D. Comaniciu, "Deep decision network for multi-class image classification," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2240–2248.

[39] G. Tsoumakas and I. Katakis, "Multi-label classification: An overview," *Data Warehousing and Mining: Concepts, Methodologies, Tools, and Applications*, pp. 64–74, 2008.

[40] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255. DOI: `10.1109/CVPR.2009.5206848`.

[41] L. Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[42] S. Saponara and A. Elhanashi, "Impact of image resizing on deep learning detectors for training time and model performance," in Jan. 2022, pp. 10–17, ISBN: 978-3-030-95497-0. DOI: `10.1007/978-3-030-95498-7_2`.

[43] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, *Rethinking the inception architecture for computer vision*, 2015. arXiv: `1512.00567 [cs.CV]`. [Online]. Available: `https://arxiv.org/abs/1512.00567`.

[44] K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition*, 2015. arXiv: 1512.03385 [cs.CV]. [Online]. Available: https://arxiv.org/abs/1512.03385.

[45] *Cudnn license page*, https://docs.nvidia.com/deeplearning/cudnn/backend/latest/reference/eula.html, Accessed: 2025-03-30.

[46] *Python software foundation license version 2*, https://docs.python.org/3/license.html.