Optimiranje neuronske mreže evolucijskim računanjem za detekciju tumora u MR slikama mozga

Đurinec, Jan

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS No. 705

EVOLUTIONARY COMPUTING OPTIMIZATION OF NEURAL NETWORK FOR TUMOR DETECTION IN BRAIN MR IMAGES

Jan Đurinec

Zagreb, February 2025

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS No. 705

EVOLUTIONARY COMPUTING OPTIMIZATION OF NEURAL NETWORK FOR TUMOR DETECTION IN BRAIN MR IMAGES

Jan Đurinec

Zagreb, February 2025

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Zagreb, 30 September 2024

MASTER THESIS ASSIGNMENT No. 705

Student:	**Jan Đurinec (0036526931)**
Study:	Computing
Profile:	Computer Science
Mentor:	asst. prof. Jelena Božek, PhD

Title: **Evolutionary computing optimization of neural network for tumor detection in brain MR images**

Description:

Evolutionary computation is a diverse family of algorithms inspired by the principles of biological evolution, particularly suited for global optimization challenges. These algorithms have demonstrated significant potential in medical image analysis, particularly in detecting tumors within brain MR images. They can efficiently navigate complex, high-dimensional search spaces to identify patterns and abnormalities that may be difficult to capture with traditional methods. This thesis aims to provide a comprehensive introduction to evolutionary computing, followed by a detailed survey of the latest state-of-art application to brain tumor detection. The primary objective is to implement evolutionary computing techniques for brain MR image segmentation, serving as a preprocessing step for neural networks input, and to implement evolutionary computing optimized neural network for brain tumor detection. Explore several optimization algorithms and compare their performance on publicly available dataset comprising annotated brain MR images containing tumors.

Submission date: 14 February 2025

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 30. rujna 2024.

DIPLOMSKI ZADATAK br. 705

Pristupnik:	**Jan Đurinec (0036526931)**
Studij:	Računarstvo
Profil:	Računarska znanost
Mentorica:	doc. dr. sc. Jelena Božek

Zadatak: **Optimiranje neuronske mreže evolucijskim računanjem za detekciju tumora u MR slikama mozga**

Opis zadatka:

Evolucijsko računanje raznolika je obitelj algoritama nadahnutih načelima biološke evolucije, posebno prikladnih za izazove globalne optimizacije. Ovi algoritmi pokazali su značajan potencijal u analizi medicinskih slika, posebice u otkrivanju tumora na MR slikama mozga. Oni se mogu učinkovito kretati složenim, visokodimenzionalnim prostorima pretraživanja kako bi identificirali obrasce i abnormalnosti koje je teško uhvatiti tradicionalnim metodama. Ovaj diplomski rad ima za cilj pružiti sveobuhvatan uvod u evolucijsko računanje, nakon čega slijedi detaljan pregled suvremenih primjena u području detekcije tumora mozga. Primarni cilj je implementacija tehnika evolucijskog računanja za segmentaciju MR slike mozga, koja služi kao korak predprocesiranja podataka za unos u neuronsku mrežu, i implementacija neuronske mreže optimirane evolucijskim računanjem za detekciju tumora mozga. Potrebno je istražiti različite algoritme za optimizaciju i usporediti njihovu izvedbu na javno dostupnom skupu podataka koji sadrži označene MR slike mozga koje sadrže tumore.

Rok za predaju rada: 14. veljače 2025.

Contents

Introduction

In recent years, the global hype surrounding Artificial Intelligence (AI) and machine learning (ML) has spurred across a variety of fields, from healthcare to autonomous vehicles. As these technologies evolve, they continue to demonstrate their potential to transform industries and improve our daily lives. One example is the realm of medical image analysis, where brain tumors can be easily detected in brain magnetic resonance imaging (MRI) scans with the help of AI. However, the complexity and high dimensionality of medical data pose significant challenges, making it difficult for traditional algorithms to achieve optimal results. Brain MRI scans are often large, containing vast amounts of information, and the tumors themselves may vary in size, shape, and location. This variability makes it difficult to identify tumors using conventional image processing techniques, which may struggle to account for the wide range of possible tumor characteristics. This is where evolutionary computing, a branch of AI inspired by biological evolution and nature behaviors, comes into action. Evolutionary computing offers a unique approach to solving optimization problems. By mimicking the way nature evolves organisms to adapt and survive, evolutionary algorithms can explore large, complex solution spaces and find optimal or near-optimal solutions in a more efficient manner. This thesis explores the application of evolutionary computing algorithms was explored to optimize a process of brain tumor detection, with the focus on image segmentation as a step to simplify and extract the most valuable features from the MRI scans and optimization of convolutional neural networks. In the context of brain tumor detection, evolutionary algorithms can be employed to optimize three main aspects: the preprocessing step for the network's input images, the selection of optimal network hyperparameters, and the minimization of the network's loss function. All of these actions are necessary to improve the network's performance and accuracy. The primary objective is to implement an evolutionary computing optimized neural network for brain tumor detection and to implement and explore evolutionary computing techniques for brain MR image segmentation, serving as a preprocessing step for the network's input. Furthermore, the thesis aims to explore several optimization algorithms and compare their performance on a publicly available dataset with brain MR images containing various types of tumors. This work is organized as follows: a brief introduction to brain tumor MRI, ML techniques, an introduction to CNNs and evolutionary computing, a short survey of related

works and finally, an overview of brain tumor detection system without evolutionary computing (third section) and with evolutionary computing (fourth section). In summary, the aim of this thesis is to provide a comprehensive introduction to evolutionary computing, followed by a detailed survey of the latest state-of-the-art applications to brain tumor detection and classification.

1.1. Brain tumor MRI

There are many ways and techniques for creating detailed brain images, their structure and functionality. Some of the most renowned imaging techniques are:

- *Magnetic resonance imaging (MRI)* – a non-invasive method that creates an image using magnetic fields and radio waves
- *Functional magnetic resonance imaging (fMRI)* – a subcategory of MRI, indirectly measures the brain's blood flow and creates an image based on that
- *Computed tomography (CT)* – uses X-rays to create an image, a method is avoided due to ionizing radiation
- *Positron emission tomography (PET)* – uses a small sample of radioactive material that is injected into the bloodstream

Similar to finding the best model in the field of machine learning, there is no best method to create an optimal image of brain in general. It depends on a specific task and the health condition of a subject. Some of the most important features in brain tumor detection and classification problems are the volume of tumor, its texture, and the subject's age and gender. Based on those extracted features, machine learning models are able to learn the characteristics of specific tumor types and make general assumptions on new, never seen samples. Normal brain tissue is composed of gray matter, white matter, and cerebrospinal fluid (CSF) and tumors can form anywhere in that area. According to definition [1], tumors are abnormal growth of cells that can form in any part of the body. They occur when cells begin to divide uncontrollably. Tumors can be benign, and non-cancerous, in which case they usually do not spread, or they can be malignant, meaning they are cancerous and have the potential to invade nearby tissues or spread to other parts of the body [2]. Because of their nature, especially for malignant tumors, early detection is crucial for further diagnosis and successful treatment. There are many types of brain tumors, and the dataset used in this work, which will be further explained in the dataset section, contains only three types: pituitary tumors, glioma tumors and meningioma tumors. Tumor cell characteristics, including irregular shapes, heterogeneous intensity distributions, variability in tumor location, and the presence of imaging artifacts, significantly impact the diagnostic process. Tumor heterogeneity refers to the distinct morphological and phenotypic variations observed among tumor cells, such as differences in cellular structure, gene expression profiles,

metabolic activity, motility, proliferation rates, and metastatic potential [5]. This heterogeneity poses substantial challenges in the design and implementation of effective treatment strategies. Studies conducted by the National Brain Tumor Foundation (NBTF) indicate that brain tumors are a leading cause of mortality worldwide, with their incidence having more than tripled over the past three decades [4]. This dramatic increase highlights the urgent need for advanced and reliable detection or classification techniques, which are essential for early diagnosis, precise treatment planning, and ultimately improving outcomes for many patients impacted by this condition.

In this work, only MRI samples are used, since MRI is one of the most used techniques for brain imaging because of its non-invasive nature and high resolution. Those images can provide critical information about the structure of the brain, which is crucial for detecting abnormalities or classification of tumors. MRI scans can be performed using various sequences, each designed to highlight specific tissue properties. The two most common sequences are T1-weighted images and T2-weighted images.

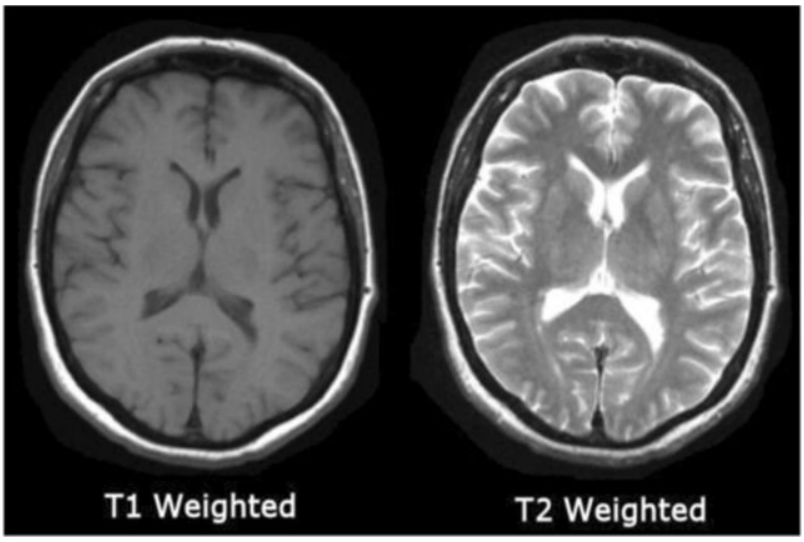

Figure 1.1. T1 and T2 weighted MR image comparison
[5]

T1-weighted images are often used to assess the overall structure of the brain and identify abnormalities such as tumors or lesions. Cerebrospinal fluid has a dark appearance, and white matter has a bright appearance as it can be seen in Figure 1.1. On the other hand, T2-weighted images have cerebrospinal fluid appearing bright and white matter appearing dark. They are often used for detecting areas of edema, inflammation or pathological changes [6]. MRI data can be acquired in either two-dimensional (2D) slices or three-dimensional (3D) volumes. While 2D imaging involves capturing individual slices of the brain, 3D imaging collects data across an entire volume, allowing for more detailed and isotropic analysis. In research and clinical applications, 3D MRIs are often preferred because they enable more precise segmentation, registration, and volumetric measurements. For the processing of MRI data, there are significant computational challenges such as high dimensionality (a single 3D MRI scan can contain hundreds of slices) or preprocessing complexity (noise reduction, normalization…). Due to computational limitations of the available hardware, only 2D MRI scans were utilized in this work. Processing 3D volumetric MRI data requires significant amount of memory and computational power, which exceeded the capacity of the system used in this work. The 2D approach, while less detailed than 3D analysis, still allowed effective processing and analysis within the given constraints.

1.2. Machine learning

In general, machine learning (ML) is divided into supervised, unsupervised and semi-supervised (reinforcement) machine learning (Figure 1.2.). The first category is supervised learning, where models are trained on labeled data. In this case, each input is paired with a corresponding output, allowing the model to learn the relationship between them. The primary goal of supervised learning is to predict the output for new, unseen inputs. It is commonly used for tasks such as classification, where the model predicts discrete categories, and regression, where the model predicts continuous values. Examples include predicting the type of brain tumor from MRI scans or estimating the age of a subject based on brain imaging data. As opposed to supervised learning, unsupervised learning deals with unlabeled data. In this approach, the model identifies patterns, structures, or relationships within the data without explicit guidance. The main goal of unsupervised learning is to discover hidden patterns or groupings in the data. This is particularly useful for clustering, where similar data points are grouped together, or for dimensionality reduction, which reduces the number of features while retaining significant information. For example, clustering techniques could be used to group MRI scans based on texture or intensity, while dimensionality reduction methods, like principal component analysis, can simplify complex datasets. Reinforcement learning represents a different approach, where an agent learns to make decisions by interacting with an environment. The agent receives feedback in the form of rewards or penalties and optimizes its actions over time to maximize cumulative rewards. This type of learning is widely used in robotics, game-playing AI, and autonomous systems, where sequential decision-making is critical [7].

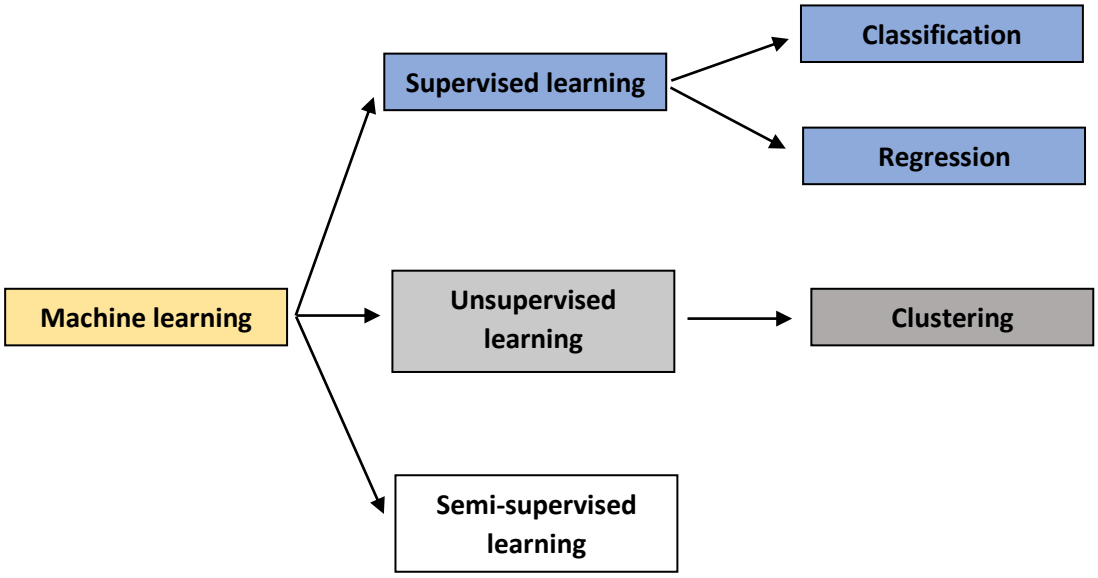

Figure 1.2. Division of ML methods

In this work, the focus is going to be on supervised learning. Hence, other branches of machine learning will not be further explained. Brain tumor detection, in this case, is a binary classification problem. On the output of a model, there are only two classes: tumor is detected and tumor is not detected. As mentioned above, the main difference between a supervised and unsupervised approach is whether labeled data is available for the learning process or not. Some of the most common classification algorithms are:

- **Logistic regression**: Despite its name, it is a classification algorithm. It predicts the probability of a data point belonging to a specific class using a logistic function.
- **Decision trees**: These algorithms split data into subsets based on feature values, creating trees of decision. They are very easy to interpret.
- **Support vector machines (SVM):** The algorithm finds the hyperplane that optimally separates classes in the feature space. It is very effective in high-dimensional space and for data with clear margins of separation.
- **Neural networks:** Inspired by biological neural systems, they are a powerful tool for complex problems but require significant computational resources.

- **Random forest:** An ensemble method that combines decision trees to improve classification accuracy and reduce overfitting.
- **Naïve Bayes:** Based on Bayes' theorem, this algorithm assumes feature independence. Works well for text classification problems.

There are several different machine learning classification algorithms that are not mentioned above because they are either a combination of the algorithms above, their generalization, or it was found that they do not perform well in practice. Furthermore, it is important to mention that there is no single best classification algorithm. Each one of them depends on computational resources and input data, and depending on data, they can perform better or worse than the other ones. In order to find out the best possible machine learning model for a specific input data, it is necessary to try various different models and compare their results. Choosing the best possible ML model is often a matter of experience and domain knowledge, because understanding the characteristics of the data and the problem is crucial for selecting the most appropriate algorithm. In practice, small subsets of the data can be used to train and evaluate different models, allowing for a quick comparison of their performance. This approach, often referred to as model benchmarking, helps in identifying optimal candidates without the need to invest excessive computational resources upfront. Once the best-performing models are identified on the smaller subset, they can be further fine-tuned and validated on the full dataset. Fine-tuning is a process of finding the optimal hyperparameters for the chosen ML model. For example, finding the optimal number of decision trees in the random forest algorithm.

1.3. Convolutional neural networks

One of the biggest aspects of this work are convolutional neural networks (CNNs), which have revolutionized the field of machine learning, particularly in the realm of image processing and computer vision. In order to fully grasp the significance and work behind the CNNs, understanding the fundamentals of neural networks and the process of convolution is essential. Some of the first learning algorithms we know today were designed as computational representations of biological learning, or in other words, models of how learning occurs or could occur in the brain. As a result, deep learning has often been referred to as artificial neural networks (ANNs) [8]. These models consist of layers of interconnected "neurons" that process input data, mimicking how biological neurons transmit and process signals. They are connected by weighted links. Each connection between neurons represents a synapse, and the weight associated with each link determines the strength of the connection. As a result, the network is able to adapt to the data it processes because the network's weights are adjusted during the learning process (Figure 1.3.)

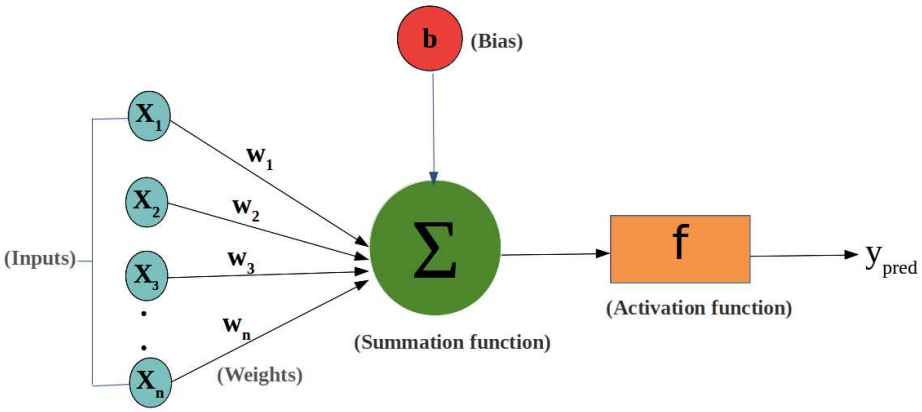

Figure 1.3. Components of the basic artificial neuron [10]

A neural network (NN) is organized in three types of layers: an input layer, hidden layers, and an output layer (Figure 1.4.). The input layer receives raw data, which is then processed through one or more hidden layers. Each neuron in these hidden layers applies a mathematical function, known as an activation function, to the incoming signals. The most common activation functions include the sigmoid, hyperbolic tangent (tanh), and Rectified

Linear Unit (ReLU). The processed data then passes to the output layer, which produces the network's final predictions or decisions.

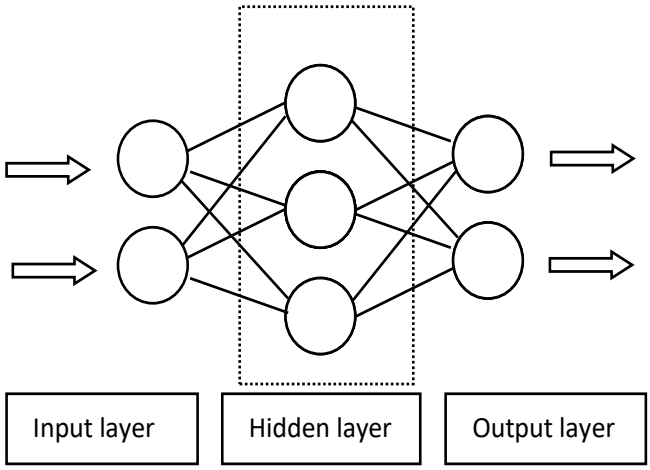

Figure 1.4. Structure of NN

In machine learning, every algorithm is composed of three essential components: the model, the loss function, and the optimization process. These components work together to enable the algorithm to learn from the data – to train and generate predictions. Training involves adjusting the weights of the network to minimize the difference between the predicted output and the actual target. This is typically done using a supervised learning algorithm, where the network is provided with input-output pairs, and the goal is to reduce the error between predicted and true outputs. The error is usually measured using a loss function, such as mean squared error (MSE), and the network updates its weights through a process called backpropagation. Backpropagation computes the gradient of the loss function with respect to each weight and adjusts the weights in the direction that reduces the error, using an optimization algorithm like gradient descent. Müller [9] emphasizes that one of the challenges in neural network training is avoiding overfitting, which occurs when a model becomes too complex and performs well on training data, but poorly on unseen data. Regularization techniques, such as dropout layers or L2 regularization, are often employed to mitigate the issue of overfitting and help the network generalize better to new data. On the other hand, underfitting occurs when a model is too simple, or too shallow. For example,

a linear model cannot successfully solve a problem, regression or classification, of input data where the relationship of the features is non-linear. A field of machine learning that observes deep neural networks – ones with many hidden layers, is called deep learning. These networks have been instrumental in achieving breakthroughs in complex tasks such as object detection, language translation, and medical diagnosis.

Convolutional neural networks are a type of classical multi-layer feedforward neural network consisting of an input layer, a hidden layer, and an output layer. The output of each neuron is a feature map, while kernels are used instead of weights. (Figure 1.5.)

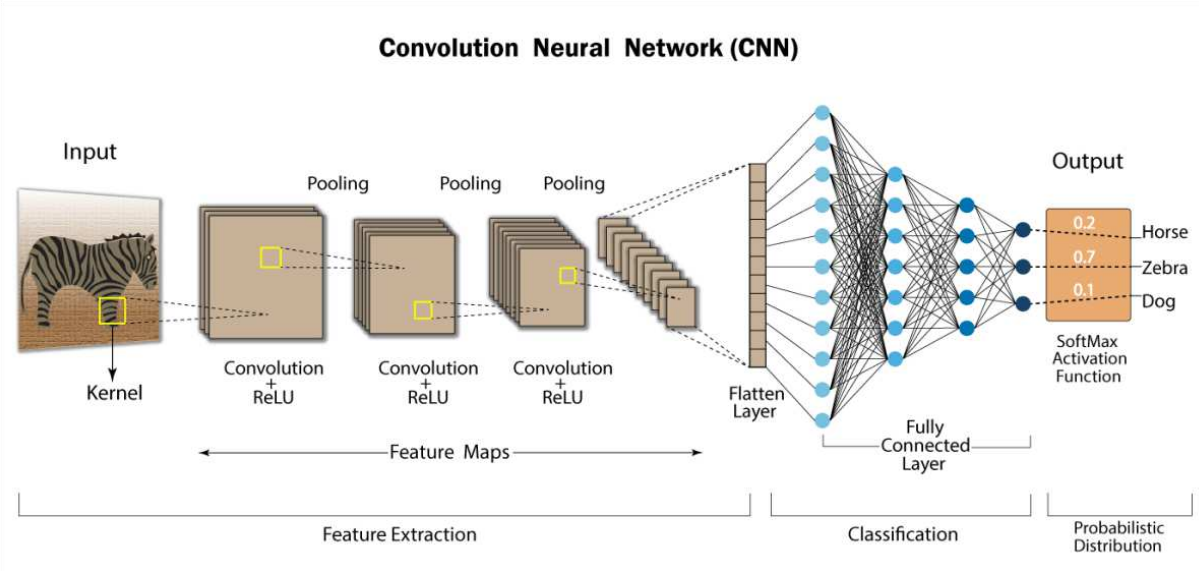

Figure 1.5. Typical CNN architecture [11]

CNNs are particularly suitable for classification, segmentation, and regression of image-based inputs. The most common architecture consists of convolution layers, pooling layers, and at the end, a few fully connected layers. The term “convolutional” comes from the concept of convolution, whose mathematical definition describes it as an operation on two functions that produces a third function which expresses how the shape of one alters the other. In the context of machine learning, convolution occurs between the input feature map and the kernels. In CNNs, convolution is a crucial step in feature extraction. Early convolution layers can detect simple features such as edges and textures, while deeper layers can detect more complex patterns such as shapes, objects, or even faces. Other types of

layers, besides convolutional or fully connected layers, are pooling layers. Pooling layers reduce the resolution of the feature maps and increase the spatial invariance of the neural network. There are several types of pooling, with max pooling being the most common. A flattened layer, the one between the last pooling layer and the first fully connected layer (Figure 1.5.), has a purpose to transform multi-dimensional input into a one-dimensional vector. There are also layers such as the dropout layer and the batch normalization layer which are used to prevent overfitting and stabilize the training process. The main advantage of convolutional neural networks over fully connected networks is a smaller number of input nodes, equivalence to small shifts in the image, fewer connections, and that they cannot easily learn noise from the input data.

1.4. Evolutionary computing

Metaheuristics are advanced problem-solving techniques used to find solutions for complex optimization problems that are hard to solve using exact methods. These algorithms are designed to explore large solution spaces efficiently and to provide robust, approximate solutions to a wide range of optimization challenges. They often draw inspiration from natural phenomena or human problem-solving strategies, and their adaptability and ability to escape local optima make them effective for real-world applications. Optimization methods can be divided into two categories: exact methods and heuristic methods (Figure 1.6.). Exact methods, such as dynamic programming, A star and linear programming, guarantee the optimal solution with a lack of scalability for large or complex problems. These methods are computationally expensive. On the other hand, heuristic or approximate methods, including metaheuristics, provide approximate solutions without guaranteeing optimality. They are mostly used when a problem is too complex or a good-enough solution is acceptable taking into account computational costs and computational time. Heuristic methods are designed to explore the solution space more efficiently by applying simple rules.

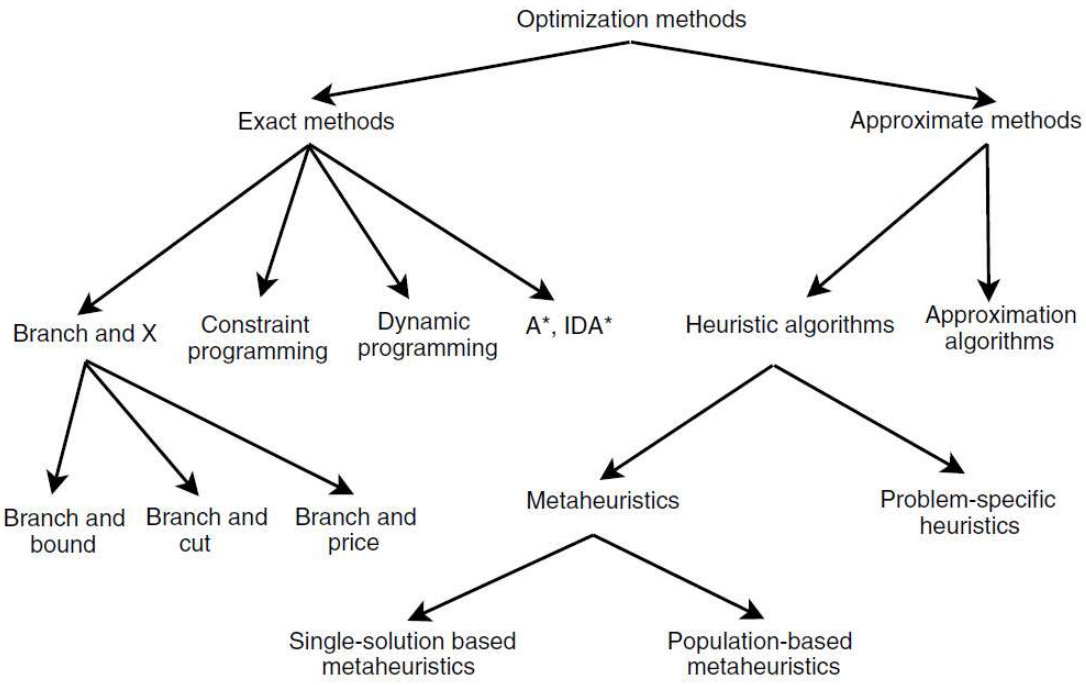

Figure 1.6. Classic optimization methods [11]

Evolutionary computing (EC), a subset of metaheuristics – population-based metaheuristics, is particularly influential, built on top of the principles of biological evolution. It uses mechanisms such as selection, mutation, and recombination to evolve a population of candidate solutions over successive generations. This approach includes popular algorithms like *Genetic Algorithms* (GA), *Genetic Programming* (GP), *Simulated Annealing* (SA), *Particle Swarm Optimization* (PSO) and *Ant Colony Optimization* (ACO), which simulate the process of natural selection and survival of the fittest to progressively improve solution quality. The flexibility of metaheuristics, and how they can be implemented across diverse fields from engineering and logistics to artificial intelligence has grown in the past years [11]. Overall, metaheuristic and evolutionary computing provide powerful tools for tackling large-scale, complex optimization tasks where traditional exact methods are not viable. In Figure 1.7. there is an obvious peek in deep learning and evolutionary computing publications in the period from 2008 to 2018.

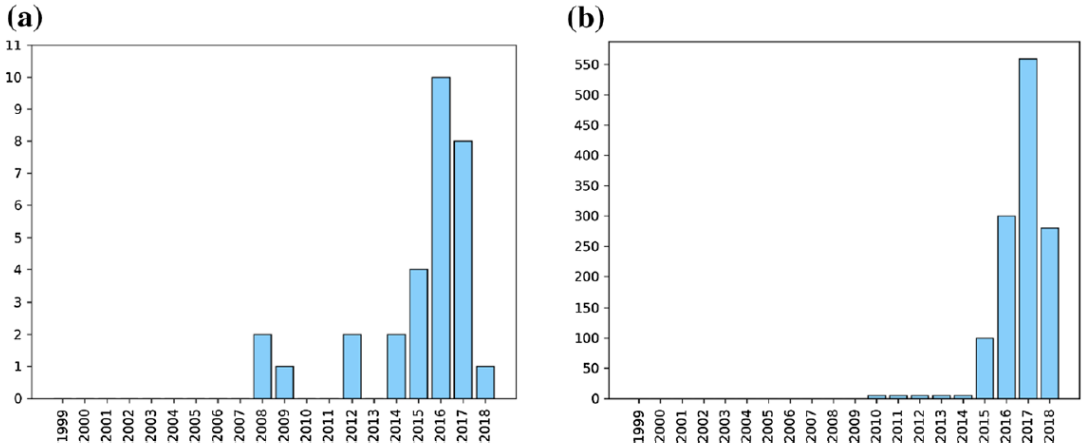

Figure 1.7. Histograms for deep learning and evolutionary computation publications according to the Web of Science based on related keywords: **a)** total publications by year and **b)** number of citations by year [12]

In the past decade, several evolutionary approaches have been proposed to solve learning problems, and these approaches have displayed remarkably good performance compared to the traditional learning methods that usually require human intervention and expertise. Traditional gradient-based methods for training multilayer neural networks are prone to problems like getting stuck in local optima and can be computationally expensive, particularly when dealing with large training datasets. As a result, training neural networks with multiple hidden layers has not been widely explored for various applications until recently. This shift in interest is due to the development of adaptive stochastic variants of the gradient descent method – a method to find the minimum in model's complex loss function. Additionally, metaheuristics can be used to optimize neural network parameters (such as determining the number of hidden layers for a specific task) and improve prediction accuracy. Notably, in recent years, there has been growing interest among core machine learning researchers in using evolutionary computing algorithms for DL. As mentioned above, the main objective of EC algorithms is to improve the efficiency of computations that can assist in solving challenging computational problems. The initial population explores the search space randomly, then the algorithm evaluates the obtained solutions to select some of them for the further step (best ones, random ones…). The new population is built as a better version of the first population (performing operations such as mutation or selection). The previous population is often referred to as the parent of the current one. EC algorithm performs a combination of exploration – investigating new areas of the search space and exploitation – refining and improving the current best solution. After some iterations, the algorithm converges to a good-enough, near-optimal solution. The balance between exploration and exploitation is critical. If exploitation dominates, the algorithm might converge too quickly to a local optimum. If exploration is too strong, the algorithm might not focus enough on promising areas and fail to refine solutions effectively. The right balance can reduce the risk of getting stuck in a local optimum. A local optimum refers to a solution in an optimization problem that is the best within a specific neighborhood, but not necessarily the best overall (global optimum), or relatively to a given neighboring function N, a solution s ϵ S is a local optimum if it has a better quality than all its neighbors; that is, $f(s) \leq f(s')$[2] for all $s' \epsilon N(s)$ (Figure 1.8.). Global optimum - a solution $s^* \epsilon S$ is a global optimum if it has a better objective function than all solutions of the search space, that is, $\forall s \epsilon S, f(s^*) \leq f(s)$ [11].

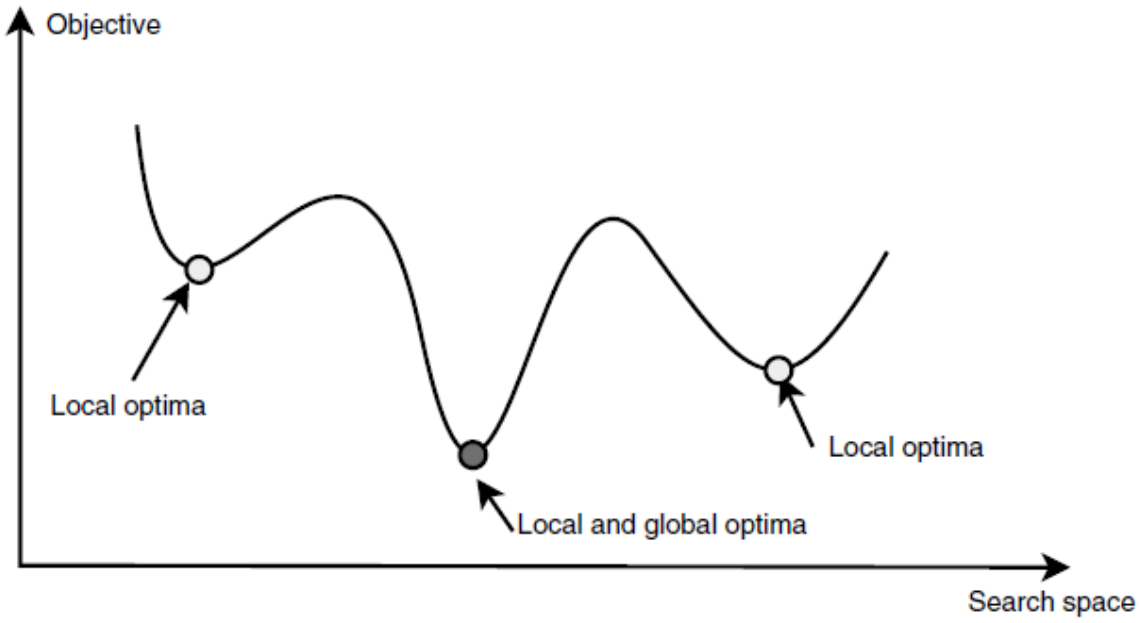

Figure 1.8. Local optimum and global optimum in a simple search space [11]

2. Related work: evolutionary computing in DL

As one of the main objectives of this work was to provide a comprehensive introduction to evolutionary computing and a detailed survey of the latest state-of-the-art applications to brain tumor detection, the following section will analyze and discuss the various approaches, methodologies, and results from recent papers, theses and similar works. It is essential to categorize these approaches, as EC can be applied in various ways, such as tuning the hyperparameters of CNNs, serving as a loss function within the network, or being utilized in the segmentation process of input images. Some of these approaches will be analyzed and described in detail in the following sections.

As highlighted by Yao [13], learning and adaptation are two essential aspects of the ability to adapt and learn, which have evolved over time in various species. The selection of parameters, weights, and the appropriate architecture of an ANN are considered crucial challenges. Evolutionary Computing (EC) can be applied at various stages of an ANN's development, such as during training, weight optimization, and the design of learning rules. Numerous approaches in literature have been proposed to tackle the optimization problem of fine-tuning the architecture and parameters of ANNs. Leung et al. [14] proposed GA to optimize the network structure and learning for a specific application by an ANN. Later, EC was used to adjust the number of nodes, layers or even the polynomial type. The application of EC in CNNs was introduced by Cheung and Sable in 2011 [15] where they used stochastic diagonal Levenberg-Marquardt method to accelerate the convergence of training while reducing the cost of fitness evaluation. Fujino et al. [16] used a GA to optimize the hyper-parameters of a CNN. Typically, those parameters can be the number of iterations, learning rate, momentum rate, number of filters, the shape of a filter and max pooling sizes. Aside from finding the best parameters, Khalifa et al. [17] used PSO to optimize the connection weights of the last classification layer of a seven-layer CNN architecture for handwritten digit classification and reported better results over a normal CNN that uses stochastic gradient descent (SGD) for all layers. Another example of using PSO algorithm can be found in the study by Zhang et al. [18], where they explored a novel computer-aided diagnosis system for detecting pathological brains in MRI scans. The authors combined wavelet entropy for feature extraction and hybridization of biogeography-based optimization and

PSO for training an NN. The proposed system achieved 99.49% accuracy on the large dataset which included 11 types of pathological conditions, such as glioma, Alzheimer's disease, and Huntington's disease. The proposed system outperformed 14 state-of-the-art CAD systems in accuracy. The following study by Dehkordi A. et al. [19] introduced enhanced CNN architecture optimized using *Nonlinear Levy Chaotic Moth-Flame Optimization* (NLCMFO) for brain tumor detection and classification. The performance of the proposed model was validated using the BRATS 2015 dataset, achieving superior accuracy (97.4%) and F1 score (96.6%) compared to the state-of-the-art methods. The model was also benchmarked against PSO, MFO, *Salp Swarm Algorithm* (SSA), *Whale Optimization Algorithm* (WOA) and *Gray Wolf Optimizer* (GWO) algorithms. As shown in the figure below (Figure 2.1.), the NLCMFO algorithm achieved superior performance compared to other methods in nearly every objective space explored in the paper even before the 200th iteration. Overall, the authors concluded that metaheuristic optimization overcomes premature convergence and improves convergence speed through the combination of chaotic maps and levy flight theorem. The study above will be mentioned once again in the following sections. The study by Mahesh K. and Renjit J. [20] provides a critical survey of evolutionary intelligence and other segmentation techniques for recognizing brain tumors from MRI images. It reviews various evolutionary computing and optimization algorithms used in brain tumor recognition, alongside traditional approaches like thresholding, region-based techniques, clustering, and model-based techniques. In the study, authors used GA for tumor segmentation after preprocessing via Discrete Walvet Transform (DWT) and PSO to enhance tumor classification on the BRATS dataset.

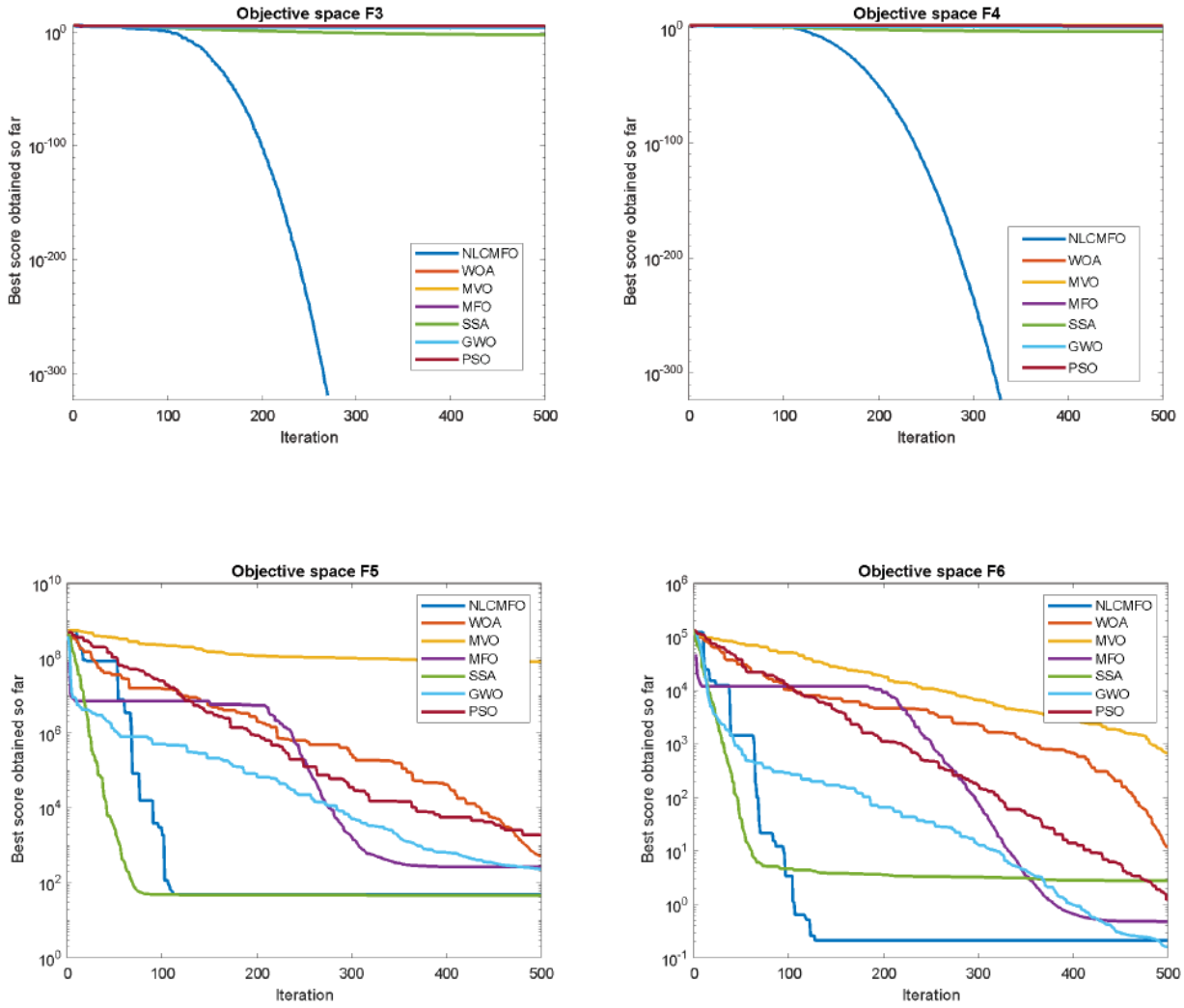

Figure 2.1. Convergence analysis of NLCMFO versus other approaches [19]

3. Brain tumor detection

As mentioned before, the significance of early brain tumor detection is crucial for improving patient outcomes and the effectiveness of treatment options. Nowadays, there are several solutions and algorithms suited for brain tumor detection, classification or segmentation. Because the main focus of this work was CNN optimization, only the problem of brain tumor detection has been further discussed with the appropriate implementation. In general, the term object detection involves identifying and localizing specific objects within and image. It provides the class labels and their location. On the other hand, classification assigns a label to an entire image answering the question "What is in this image?". Image segmentation divides an image into multiple segments or regions. The process involves a detailed understanding of object boundaries and shapes within the image. The following implementation and work in general, refers to brain tumor detection as a binary classification – if there is a tumor in the specific image or not. However, the work described in this thesis can be adjusted and used in problems such as brain tumor classification, segmentation or detection with the specified location of a tumor. Another important note regarding this and the following sections, is that only the 2D brain MR images were used. Details will be discussed in the section about dataset and data manipulation. Due to resource constraints associated with the computational capabilities of the PC used in this work, it was not possible to utilize 3D MRI scans. All the methods can be scaled to 3D MRI scans if appropriate resources become available, but this is something for future work. The following section will describe the process of developing a CNN for the binary classification of brain tumors using MR images. Subsequently, evolutionary computing and metaheuristics will be employed to enhance the metrics obtained in this section.

3.1. Brain tumor detection – dataset

The most important thing in any ML solution is a dataset. If there is no knowledge that can be extracted from the given dataset, the ML algorithm cannot learn patterns and will struggle to make accurate predictions. Poorly selected, incomplete, noisy or wrong-labeled data can lead to inaccurate results. According to the survey by Anaconda [21] in 2022, data scientists spend more than 50% of their time on data preparation tasks such as loading, cleansing and visualizing the data. In this work, two datasets were merged together which resulted in 3509 images of both T1-weighted and T2-weighted MRI scans acquired from various perspectives containing 2915 images with tumor and 594 healthy brain images. Figure below shows some samples without tumors (Figure 3.1.), and some samples with tumors (Figure 3.2.)

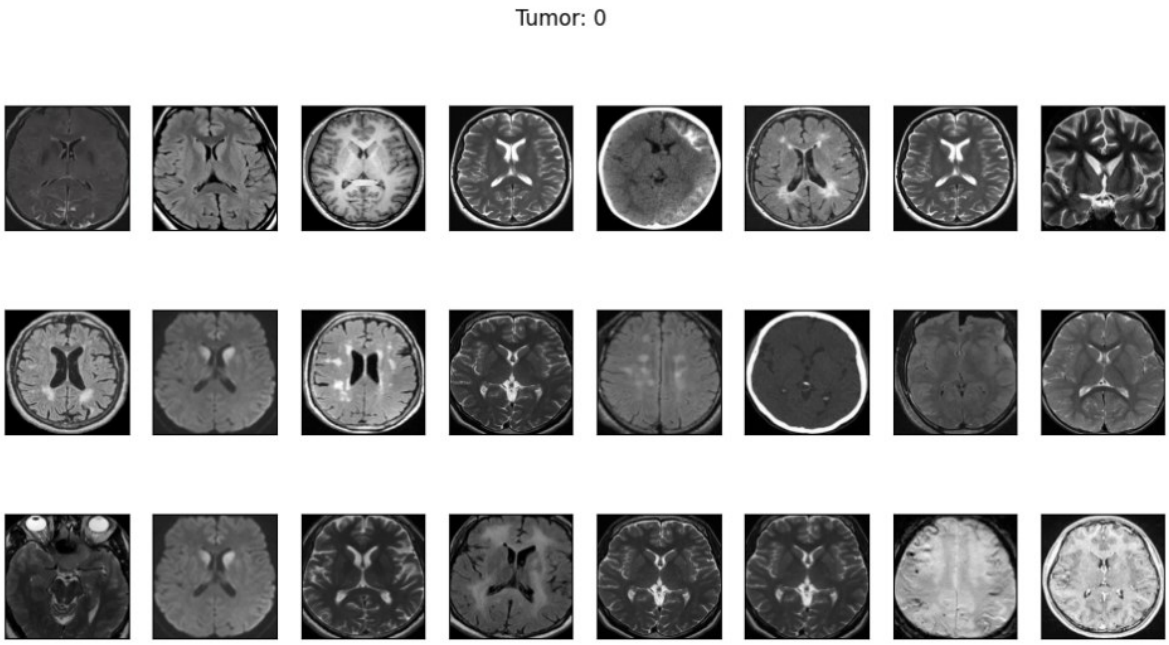

Figure 3.1. Samples from the dataset without tumor

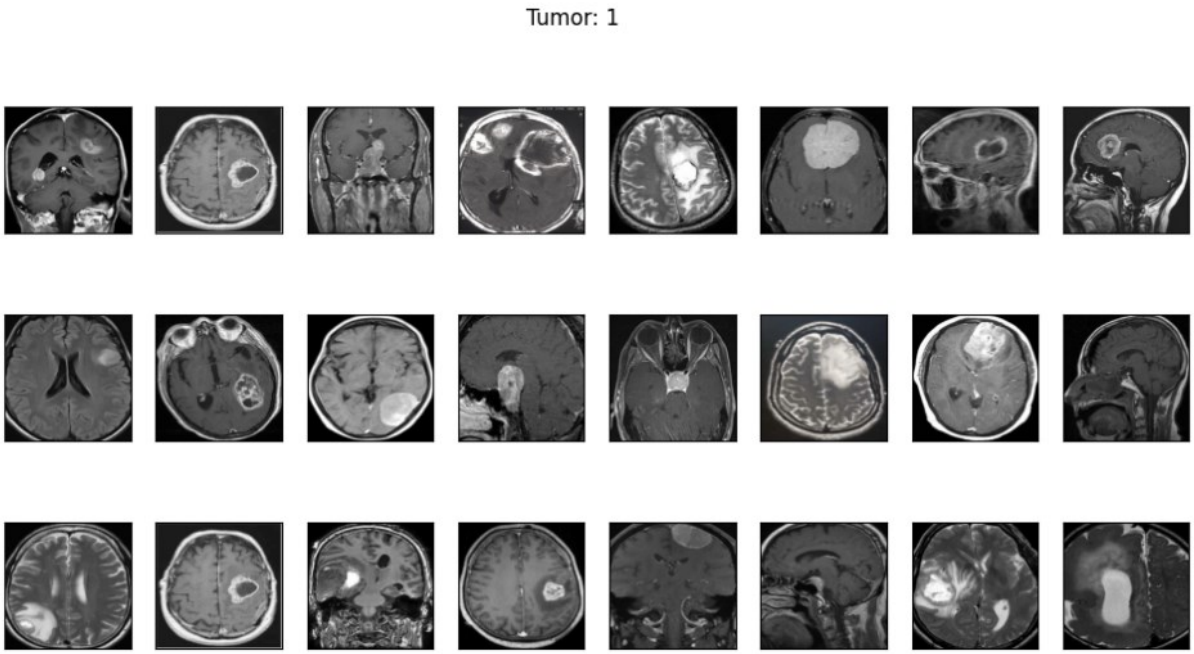

Figure 3.2. Samples from the dataset with tumor

The first dataset, publicly available on the Kaggle platform [22], contains 255 images in total, including a few images that are not images of a brain, some with wrong labels, and some duplicates. There are 98 images without tumors and 155 images with tumors. In the end, a total of 247 images were used. A second dataset, also publicly available on the Kaggle platform [23], contains 3264 images in total. This dataset was built for brain tumor classification in four classes: no tumor, glioma tumor, meningioma tumor, and pituitary tumor. For the purposes of this work, i.e. binary classification, all images with tumors were put into the same folder and given the same label. After the collection of all images, they were resized to the specified size (240 - width, 240 - height, 3 - color channels), normalized and shuffled. Further processing will be explained in details in the next sections. It is important to note that the dataset is slightly imbalanced, with only 15% of images representing non-tumor class on the whole dataset. While data balancing was not addressed in this work, it is acknowledged that implementing such techniques could enhance the performance metrics and overall results. In the process of model training, 80% of the dataset was used for training, whereas 10% of that 80% was used for validation. Another 20% of the dataset was used for testing and calculating accuracy and F1 score metrics. The full dataset was mainly used for training and testing the final optimized CNN model, whereas for the research part (hyperparameter tuning, input image preprocessing…) only 70% of the whole data was used as a subset to perform studies.

3.2. Brain tumor detection - CNN architecture

The second step in every data science or machine learning project is defining the appropriate machine learning model. Based on previous experience and related work, CNNs are used in this work. Determining the optimal architecture for CNNs - including aspects such as depth, the number of convolutional layers, appropriate regularization techniques, and kernel sizes - presents a significant challenge. A network that is too shallow may struggle with generalization to unseen data. On the other hand, a network that is excessively deep often demands a significant amount of time and computational resources for training, while similar performance could potentially be achieved with a less complex architecture. Furhtermore, deep networks often have problems such as exploding gradients (networks have trouble with converging) and overfitting (performing well on training data and poor on test data). One of the first approaches in optimizing the CNN was finding the optimal convolutional kernel window size, convolutional input dimension and the pooling window size. Those terms are explained in the first section. All the code is written in Python programming language on a Linux machine. Specifications are the following:

- Version: #52-Ubuntu SMP PREEMPT_DYNAMIC
- Machine: x86_64
- Processor: x86_64
- CPU cores: 16
- RAM: 31 GB
- GPU: NVIDIA GeForce RTX 4070 8 GB

The first model used on the mentioned dataset was a CNN with two convolutional layers, ReLu activation function, batch normalization layer, max pooling layer, dropout layer and fully connected layer with sigmoid activation. The model was compiled with binary cross-entropy loss and ADAM optimizer [24]. The whole architecture is shown below (Figure 3.3.)

Model: "sequential_2"

Layer (type)	Output Shape	Param #
conv2d_4 (Conv2D)	(None, 240, 240, 32)	896
batch_normalization_2 (BatchNormalization)	(None, 240, 240, 32)	128
conv2d_5 (Conv2D)	(None, 240, 240, 32)	9,248
max_pooling2d_2 (MaxPooling2D)	(None, 120, 120, 32)	0
dropout_2 (Dropout)	(None, 120, 120, 32)	0
flatten_2 (Flatten)	(None, 460800)	0
dense_2 (Dense)	(None, 1)	460,801

Total params: 471,073 (1.80 MB)
Trainable params: 471,009 (1.80 MB)
Non-trainable params: 64 (256.00 B)

Figure 3.3. First CNN arhitecture

The performance of the model was very poor. It achieved an accuracy of 63% and an F1 score of 71% on the first dataset – not the merged one used in the next sections. The model's confusion matrix is shown below (Figure 3.4). A confusion matrix is one of the tools in machine learning used to evaluate the performance of classification models. It provides a comprehensive summary of the model's predictions compared to the actual outcome. It shows the true positive (TP) rate – number of positive instances correctly predicted as positive, the true negative (TN) rate – number of negative instances correctly predicted as negative, the false positive (FP) rate – number of negative instances incorrectly predicted as positive (Type I error) and the false negative (FN) rate – number of positive instances incorrectly predicted as negative (Type II error).

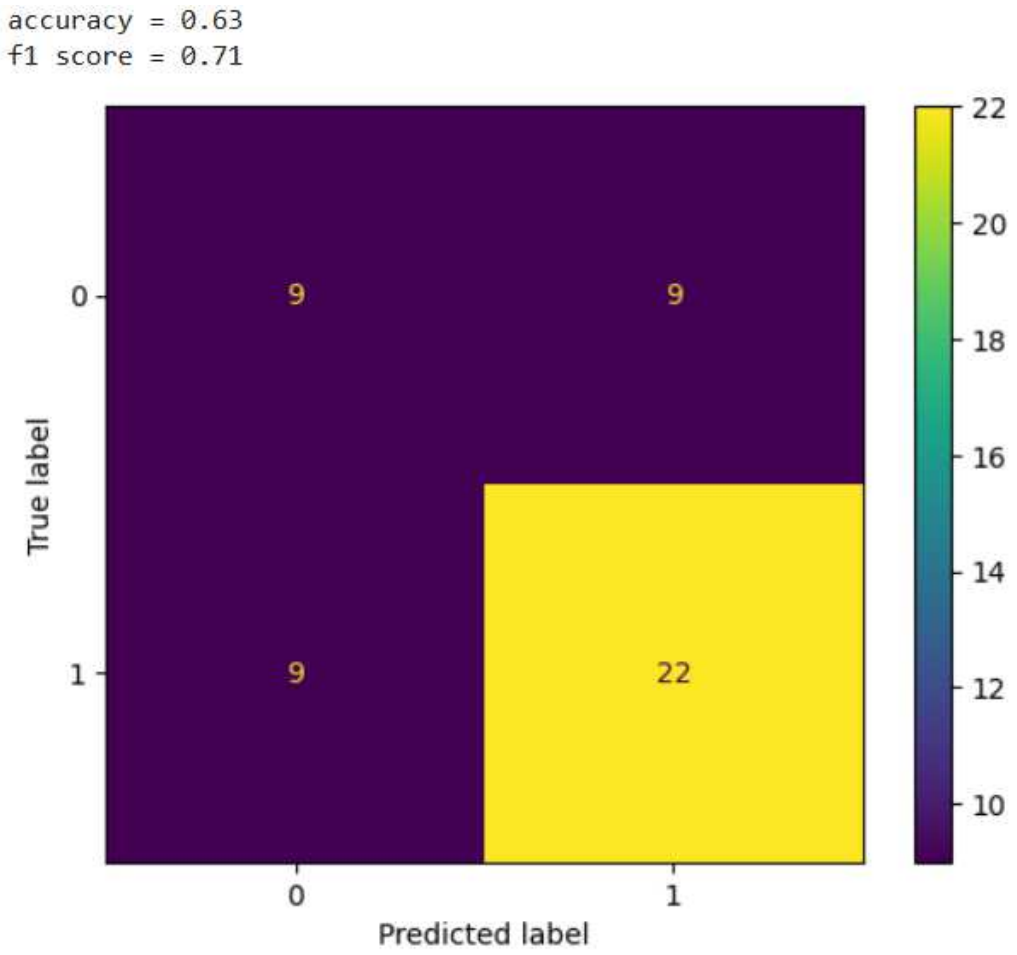

Figure 3.4. First CNN model's confusion matrix

In medicine, type II errors or false negative samples are considered more significant due to their potential to delay critical treatment. For example, if the model incorrectly identifies an MRI scan as having a tumor when it does not, a specialist can give another look and suggest getting a second opinion on the topic. But, if the model fails to identify an existing tumor in an MRI scan from the start, second opinion might never be suggested. That is the reason why all the metrics in the work contain the accuracy and the F1 binary score. When class distributions are imbalanced, a model can reach an accuracy of over 90% by predicting the majority class. For example, in a binary classification, where 95% of the samples are from the majority class – with tumors, a model that always predicts 'tumor' would still have high accuracy. That is the reason why the F1 score, or the F-measure, is used. The mentioned metric takes into account harmonic mean between the model's precision and recall. It ranges from 0 to 1, where 1 indicates perfect performance.

The following chunk of code showcases the implementation of the grid search method with the aim of finding optimal parameters (number of filters in the convolution layer, convolutional kernel window size and pooling layer window size) in a shallow CNN (Figure 3.5.)

```
output_sizes = [16, 32, 64, 128]
kernel_size = [3, 4, 5]
pooling_size = [2, 3, 4]
epochs = [5, 8, 12, 18]

for out in output_sizes:
  for kernel in kernel_size:
    for pool in pooling_size:
      for epoch in epochs:
        print(f'Params: output_size:{out} | kernel_size:{kernel} |
pooling_size:{pool} | epochs:{epoch}')
        m = get_model_v1_custom(out_dimension=out, kernel_window_size=kernel,
pooling_window_size=pool)
        acc, f1 = perform_training_on_model(m, X, y, epochs=epoch, verbose=False)
        best_acc = list(best.keys())[0]
        if best_acc < acc:
          best = {
                  acc: (m, f1, (out, kernel, pool, epoch))
                }
```

Figure 3.5. Grid search method to find optimal CNN's parameters

The grid search method tried every combination of parameters mentioned above, and the best model achieved an F1 score of 87%. Which is 16% more than the first, non-optimized CNN model. The whole Jupyter Notebook is available on the link [25] by the name of '*Brain_tumor_detection_v0_simple_grid_search.ipynb'*. For further observation, the model performance was still poor given the fact the problem is a simple binary classification. To solve the problem, a deeper CNN architecture was introduced, as explained in the following section.

3.3. Proposed model for brain tumor detection

In order to follow the principles of clean code and to make it easier for the reader to understand the logic behind algorithms and methods described in this work, all the important functions have been placed within the same file called *helper_functions.py* [25]. In the mentioned file, functions such as *plot_samples, perform_preprocesing* (will be described in the following sections), *load_data, plot_metrics, test_model* and *train* are developed. In another Python file, called *CNN.py* [25], the model used for the rest of this work is defined as shown in the listing below (3.6.)

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 234, 234, 128)	18,944
max_pooling2d (MaxPooling2D)	(None, 117, 117, 128)	0
batch_normalization (BatchNormalization)	(None, 117, 117, 128)	512
conv2d_1 (Conv2D)	(None, 111, 111, 64)	401,472
max_pooling2d_1 (MaxPooling2D)	(None, 55, 55, 64)	0
batch_normalization_1 (BatchNormalization)	(None, 55, 55, 64)	256
conv2d_2 (Conv2D)	(None, 49, 49, 64)	200,768
max_pooling2d_2 (MaxPooling2D)	(None, 24, 24, 64)	0
batch_normalization_2 (BatchNormalization)	(None, 24, 24, 64)	256
conv2d_3 (Conv2D)	(None, 18, 18, 32)	100,384
max_pooling2d_3 (MaxPooling2D)	(None, 9, 9, 32)	0
batch_normalization_3 (BatchNormalization)	(None, 9, 9, 32)	128
flatten (Flatten)	(None, 2592)	0
dense (Dense)	(None, 1)	2,593

Total params: 725,313 (2.77 MB)
Trainable params: 724,737 (2.76 MB)
Non-trainable params: 576 (2.25 KB)

Listing 3.5. Final CNN architecture

In the next sections, the same model will be used but with different hyperparameters and optimizers. Hence, the *get_CNN_model* function takes as an input the defined optimizer and the L2 regularization factor (figure 3.6.)

```
def get_CNN_model(optimizer, l2_reg = 0.01):
  model = Sequential([
    Input((IMG_SIZE[0],IMG_SIZE[1] ,3)),

    # First convolutional block
    Conv2D(128, 7, activation='relu', kernel_regularizer=l2(l2_reg)),
    MaxPooling2D(pool_size=(2, 2)),
    BatchNormalization(),

    # Second convolutional block
    Conv2D(64, 7, activation='relu', kernel_regularizer=l2(l2_reg)),
    MaxPooling2D(pool_size=(2, 2)),
    BatchNormalization(),

    # Third convolutional block
    Conv2D(64, 7, activation='relu', kernel_regularizer=l2(l2_reg)),
    MaxPooling2D(pool_size=(2, 2)),
    BatchNormalization(),

    # Forth convolutional block
    Conv2D(32, 7, activation='relu', kernel_regularizer=l2(l2_reg)),
    MaxPooling2D(pool_size=(2, 2)),
    BatchNormalization(),

    # Flatten
    Flatten(),
    Dense(1, activation='sigmoid')
  ])
  model.compile(loss='binary_crossentropy', optimizer=optimizer,
metrics=['accuracy'])
  return model
```

Figure 3.6. Python method to get a CNN model

The whole process of data loading, getting the CNN model, training the model and getting the test metrics is described in the diagram below (Figure 3.7). In the end, the proposed CNN model achieved the accuracy score of 63% and the F1 score of 70% without any

optimization. The mentioned metrics will be presented in the next section, where the model will be compared with the same model with a slightly different optimizer and the famous VGG-16 pre-trained model.

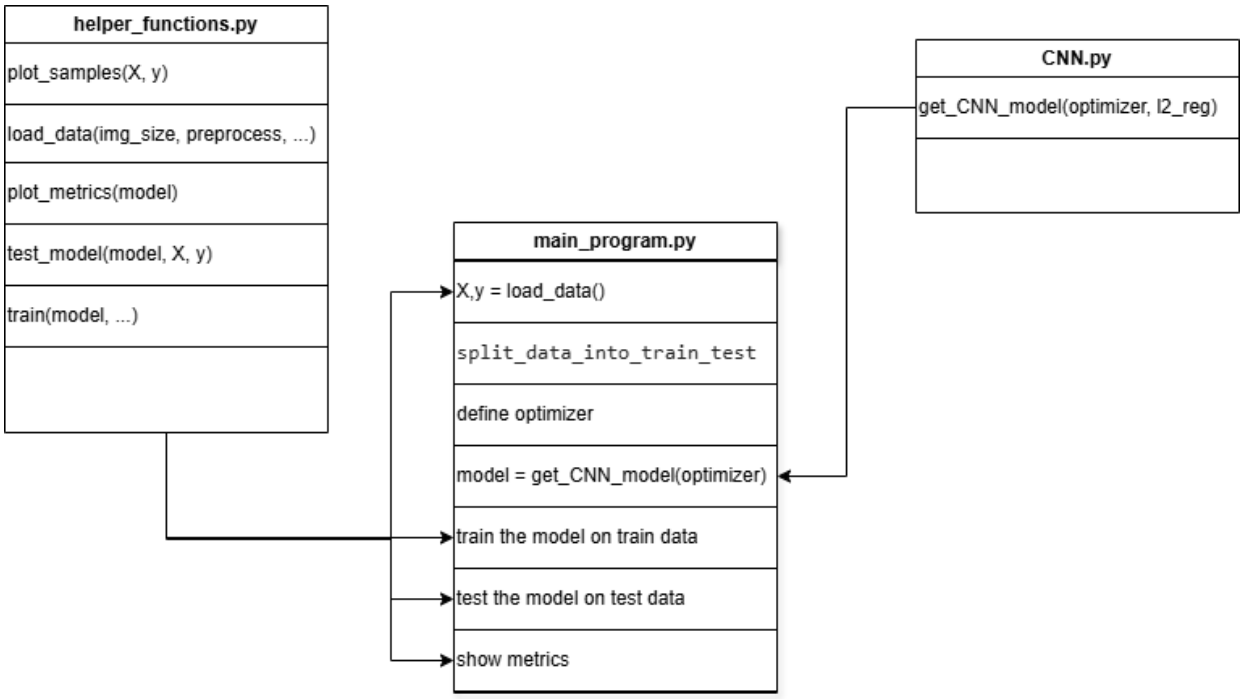

Figure 3.7. Point of view of a main program in terms of the code structure

3.4. Model comparison

The performance of the model described in the previous section was evaluated and their performance was compared against other models in this section, specifically a version of the same model with customized hyperparameters (random selection) and the VGG-16 pre-trained model. All materials are available at the link [25]. People often use pre-trained models to reduce the time and resources required for developing ML applications. Training a model from scratch is an exhausting and time-consuming process. Pre-trained models often show superior performance since they have been exposed to vast amounts of data from which they can easily recognize patterns and features. Open-source pre-trained models, such as VGG, can be customized, e.g. by adding a new classification layer or fine-tuning the existing

layers. Hence, they are simple and efficient to use. The mentioned model comparison is performed over 639 colored images in 240 x 240 resolution format. The data is split in an 80-20 ratio, where every model trains on exactly the same data. All models performed 18 learning epochs with a validation split of 25%. The figure below displays the confusion matrixes and metric for each model (Figure 3.8.), where a) is an initial CNN model with the following hyperparameters:

- Learning rate: 0.05
- Momentum: 0.8
- Optimizer: SGD
- L2 regularization factor: 0.01

Another model, b), is the same CNN with different hyperparameters:

- Learning rate: *RedcudeLROnPlateau* adaptive learning rate starting with 0.01
- Momentum: 0.7
- Optimizer: SGD
- L2 regularization factor: 0

And the last model, c), is a VGG-16 pre-trained model with the following hyperparameters:

- Learning rate: *RedcudeLROnPlateau* adaptive learning rate starting with 0.01
- Optimizer: ADAM
- L2 regularization factor: 0

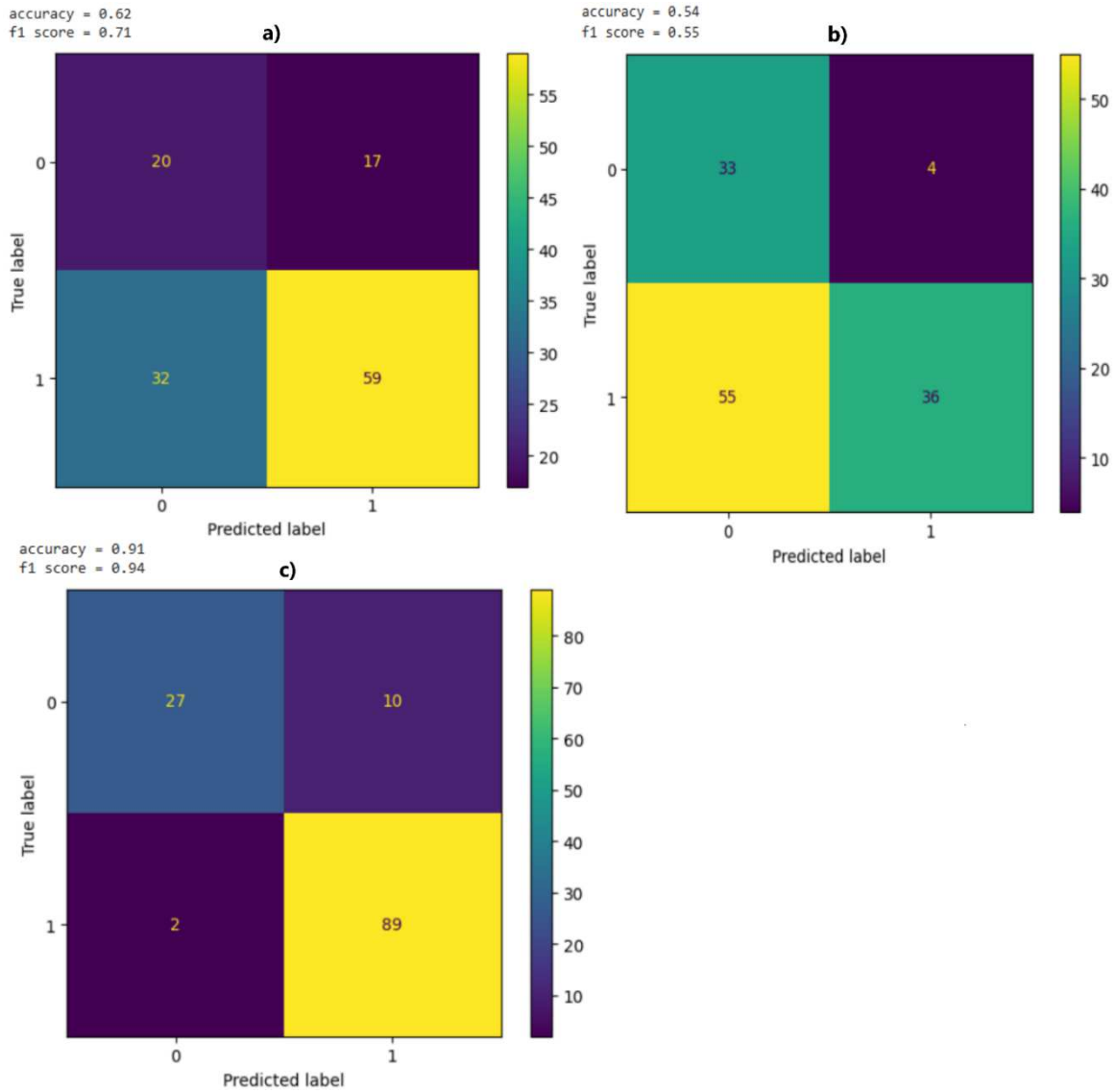

Figure 3.8. Models' comparison, a) initial CNN architecture, b) initial CNN architecture with different hyperparameters, c) VGG-16 model

The *ReduceLROnPlateau* function is a learning rate scheduler commonly used in training ML models. It adjusts the learning rate based on the performance of a specified metric – validation loss in this case. In theory, this technique helps to fine-tune the learning process, especially when the models stagnate. Another so-called *callback,* a method that helps in fine-tuning a learning process of a model, called 'early stopping' is used. It's purpose is to stop overfitting the model when the chosen metric (again validation loss) is not improving over the iterations. Those callback functions are not used in the following sections, since they were used only to observe patterns in different model training approaches. Momentum in the

SGD optimization algorithm is a technique used to accelerate convergence and smooth the optimization process. It introduces a velocity vector that accumulates the gradients' direction over time, helping the optimizer overcome small, inconsistent gradient fluctuations and push through shallow or noisy regions in the loss landscape.

In Figure 3.8., the VGG-16 pre-trained model significantly outperforms the other two models achieving an accuracy of 91% and the F1 score of 94%. In the notebook *'Brain_tumor_detection_v3_grid_search.ipynb'* [25], the grid search method is implemented to find the optimal combination of the learning rate, max. epochs, L2 regularization factor and the optimization algorithm momentum. One iteration of the algorithm takes 30 seconds to finalize, and with 100 iterations the whole process would take almost an hour. The code below shows how initial parameters are chosen (Figure 3.9.).

```
def create_n_random_entities(n: int) -> SortedSet:
  entities = SortedSet()
  for i in range(n):
    lr = round(random.uniform(0.001, 0.1),3)
    epochs = random.randint(5, 20)
    l2 = round(random.uniform(0.001, 0.1), 3)
    mom = round(random.uniform(0, 1), 3)
    entities.add(ParamsEntity("SGD",
                              lr,
                              epochs,
                              l2,
                              mom))
  return entities
```

Figure 3.9. Grid search method of finding the best CNN hyperparameters

Given that the entire process is both time-consuming and exhausting, there is no deeper logic in the selection of these parameters. Duplicates are permitted, and each entity in the population (in this case, a sorted list) lacks knowledge about the other entities. Furthermore, the process relies heavily on chance when selecting these N entities. Each entity represents a unique combination of one float value for the learning rate, one integer value for the maximum number of epochs, and float values for the L2 regularization factor and the momentum. The grid search method serves as a great motivation for using something more intelligent while seeking the optimal, or almost semi-optimal combination of the mentioned

parameters. In the next section, such methods are introduced and implemented on the same problem – a process of finding the best hyperparameters for the previously defined CNN model. Unlike grid search, which exhaustively evaluates every combination of hyperparameters in a predefined grid, evolutionary computing algorithms (EC) are way more effective and adaptable in complex and high-dimensional optimization problems where relationships between hyperparameters and model performance are not well understood. Also, by leveraging a population of solutions and evolving them over generations (iterations), EC algorithms can achieve better results in less time compared to e.g. the grid search mentioned above. By introducing some knowledge into the population, the global best solution is informing in which way should the algorithm converge and the randomness is helping to explore the whole search space, EC algorithms are designed to maintain diversity within the population, which helps prevent premature convergence, or getting stuck in the local optima. Some optimization algorithms tend to get stuck in the local optima while never discovering the global optimum, but all the details will be discussed in the following section.

4. Evolutionary computing in brain tumor detection

This final section dives into the application of EC in three critical areas of brain tumor detection: input image segmentation, hyperparameter tuning, and loss function optimization. Each of these study goals addresses a unique aspect of the problem space, emphasizing the effectiveness and adaptability of EC. First, the use of EC in input image segmentation highlights its capability to optimize complex, non-linear boundaries in medical imaging data. Motivated by many papers on the same topic, EC is commonly used for the image enhancement process after which it is very easy to segment the image into various classes. Second, hyperparameter tuning, often a time-intensive and computationally expensive process, can benefit significantly from the efficiency of EC algorithms. Lastly, the optimization function of the loss function for the CNN model opens plenty of possibilities for EC algorithms to show their performance. By employing EC in these three domains, this work underscores its role as a powerful and adaptable tool in not only the field of medicine or machine learning, but also in other fields like aerodynamics, fluid dynamics, telecommunications, robotics, physics, logistics and transportation, and similar. With continual progress of computational technologies and hardware, EC algorithms are even more applicable. On the other hand, it is not wise to use EC to solve problems where efficient and exact algorithms are available. For an NP-hard problem where state-of-the-art exact algorithms cannot solve the problem within the required search time, the use of EC algorithms and metaheuristics is justified. For example, finding optimal hyperparameters for CNN is considered an NP-hard problem since it involves searching through a high-dimensional space of possible configurations. The interactions between these hyperparameters can lead to exponential growth. Furthermore, there is no known algorithm that can guarantee finding the global optimum in polynomial time for hyperparameter tuning.

4.1. Study 1 – input image segmentation

The first study covers the first primary objective of this work – the usage of EC in the brain MRI image segmenting process. It is important to note that the dataset used in this work (section 3.1.) was not originally tailored for the segmentation task. The labels provided in the dataset were binary (0 or 1), indicating the presence or absence of the target class, rather than detailed annotations for pixel-level segmentation. As a result, the segmentation task was defined as isolating the brain matter from the skull region – also known as skull stripping. Accurate segmentation of brain tissue by removal of non-brain tissues like skull, muscle/skin, and cerebrospinal fluid is an important task since they produce a lot of noise on the CNN's input. The skull-stripping method involves a sequence of steps, starting with image enhancement using the PSO algorithm to boost performance. This is followed by background removal, histogram-based thresholding with maximum divergence to extract the brain region, and morphological operations to eliminate non-brain tissues. The whole research and final implementation are publicly available on GitHub with all the other materials under the name *'Brain_tumor_detection_v1_preprocessing.ipynb'* [25]. Gorai A. and Ghosh A. [26] introduced PSO-based automatic image enhancement techniques specifically designed for grayscale images. Their results were compared against linear contrast stretching, histogram equalization, and genetic algorithm (GA) based image enhancement approach. In most cases, the PSO-based method outperformed these techniques, highlighting its effectiveness. One key advantage of the PSO algorithm is its ability to produce better results through proper parameter tuning, a flexibility not available in methods like contrast stretching and histogram equalization, which yield only a single enhanced image for a given input. A similar method is used in this work. The enhancement process can be denoted as follows:

$$g(i,j) = T[f(i,j)]$$

where *f(i,j)* is the original image and *g(i,j)* enhanced image. T is the transformation function. Local enhancement methods apply transformations to a pixel by taking into account the intensity distribution of its surrounding neighboring pixels. The transformation function is defined as follows:

$$g\ (i,j) = \frac{k * D}{\sigma(i,j) + b}[f(i,j) - c * m(i,j)] + m(i,j)^a$$

Where D is the global mean and σ(i,j) is the local standard deviation of the $(i,j)^{th}$ pixel of the input image over an $n\ x\ n$ window. $m(i,j)$ is the local mean of the $(i,j)^{th}$ pixel over the same window. Parameters, namely a, b, c and k are introduced in the transformation function to produce large variations in the processed image. For the evaluation of the image enhancement process without human intervention, the objective function used the sum of combined three performance measures: entropy value, sum of edge intensity and number of edges. The work [26] showcases that maximizing those measures maximizes the enhancement process and quality. The PSO algorithm is used to find the optimal combination of a, b, c and k parameters, where the enhanced image has maximized the mentioned evaluation function. Particle Swarm Optimization is a stochastic population-based metaheuristic. It mimics the social behavior of swarms, such as flocks of birds or schools of fish. In the basic PSO algorithm, a swarm consists of N particles that navigate within a D-dimensional search space. Each particle i represents a potential solution to the problem and is described by the vector xi. Each particle possesses its own position and velocity, which define its movement direction and step size within the search space. Optimization takes advantage of the communication between the particles - the best position visited by the whole swarm or by particles neighborhood $gbest$ and memory of remembering its own best position $pbest_i$. There are many topologies associated with the swarm's neighborhood, such as chain topology or graph topology. Depending on the neighborhood structure, a leader refers to the particle that guides another particle's search toward a better solution. To sum it up, each particle has the following:

- X-vector of the current position in the search space
- V-vector of a gradient or velocity for the particle
- P-vector of the best solution found so far by the particle
- P-fitness value of the p-vector

The algorithm starts by defining the size of a swarm and creating particles with randomly initialized starting positions. This step is crucial for the balance between exploration and exploitation of a search space. All particles can explore different regions, which increases the likelihood of finding the global optimum and reduces premature convergence. In the implemented solution, this was achieved by creating a random, four-dimensional vector with values from 0 up to 0.5. And a similar thing for velocity vectors, with values from -1 up to 1 (Figure 4.1.)

```
class PSOptimizer:
  def __init__(self, img, w, c1, c2, num_particles=20, is_global_neighborhood=True, kernel_size=3):
    self.img = img
    self.kernel_size = kernel_size
    self.w = w
    self.c1 = c1
    self.c2 = c2
    self.num_particles = num_particles
    self.is_global_neighborhood = is_global_neighborhood

    self.population = [np.random.uniform(0, 0.5, 4) for _ in range(num_particles)]
    self.velocities = [np.random.uniform(-VELOCITY_LIMIT, VELOCITY_LIMIT, 4) for _ in range(num_particles)]

    self.personal_best = np.copy(self.population)
    self.personal_best_fitness = np.zeros(num_particles)

    self.global_best = None
    self.global_best_fitness = 0

    self.fimg = None

    self._evaluate_particles()
```

Figure 4.1. PSO algorithm constructor method

The initial vector and velocities are four-dimensional because there are four hyperparameters the algorithm is trying to optimize, *a*, *b*, *c* and *k*. After the population is initialized, in the algorithm we can decide whether to use a global neighborhood or a local neighborhood with a chain topology (every particle is linked to its left and right neighbor). For simplicity, only the global neighborhood is used in this study. The comparison and detailed explanation of how the algorithm performs regarding those neighborhoods will be discussed in the following sections. Next, the algorithm has to evaluate each particle. That is done by the *img_enhance* method which takes the image and parameters *a*, *b*, *c* and *k* for the input, and return the enhanced image and fitness value for the enhanced image (Figure 4.2.) The logic for getting the fitness function and enhancing an input image is exactly the same as the one described above with a mathematical background.

```
def img_enhance(img, a=0.1, b=0.2, c=0.3, k=1.5, n=121):
  # get local mean of (i,j)
  kernel = np.ones((n, n))
  local_mean = convolve(img, kernel, mode='reflect')

  # get local std of (i,j)
  std = pow(np.sum(abs(img - local_mean) ** 2) / (img.shape[0]*img.shape[1]
- 1), 0.5)

  # get global mean of pixels within MxN
  D = np.mean(img)

  # final result
  g = (k * (D / (std + b))) * (img - c * local_mean) + local_mean ** a

  # calculate fitness score
  hist, _ = np.histogram(g, bins=256, range=(0, 256), density=True)
  hist = hist[hist > 0]
  image_entropy = entropy(hist, base=2)
  sobel_edges = sobel(g)
  edge_intensity = np.sum(sobel_edges)
  edges_count = np.sum(sobel_edges > 0.1)
  fitness = np.log(image_entropy * edge_intensity * edges_count)

  return g, fitness
```

Figure 4.2. Image enhancing method with fitness function calculation

The most important part – particle evolution, of a PSO algorithm can now be explained. With every new iteration algorithms work as follows:

1. Define two random vectors r_1 and r_2: introducing stochasticity to avoid deterministic behavior, helping to escape local optima
2. Cognitive component, c, is calculated by the equation

$$c = c_1 * r_1 * (p_{best} - x_i)$$

3. Social component, s, is calculated by the equation, where the global neighborhood is previously defined

$$s = c_2 * r_2 * (g_{best} - x_i)$$

4. The new velocity is updated as following

$$v_{t+1} = w * v_t + c + s$$

5. The new particle position is updated as following

$$x_{t+1} = x_t + v_{t+1}$$

6. After step 5, every particle is evaluated once again and if their current position is better (defined by fitness function), the following logic applies

$$if\ fitness(x_i) > fintess(p_{best_i}):$$
$$p_{best_i} = x_i$$

7. Finally, if the particle position is better than the global best position so far, the following logic applies

$$if\ fitness(x_i) > fintess(g_{best}):$$
$$g_{best} = x_i$$

The algorithm is performing steps 1 to 7 until the stopping criteria is met (Figure 4.3.), which might be a maximum number of iterations, big enough fitness function of global best position, time limit or something similar. In this case, the algorithm is built with only 3 maximum iterations. Three iterations were enough to prove the point (evolutionary computing can be used for image segmentation) and to enhance the input image with decent results. Also, keeping in mind that the mentioned dataset has almost 700 images, algorithm has to perform three iterations for each image with 20 different particles.

Random initialization of the whole swarm ;
Repeat
  Evaluate $f(x_i)$;
  For all particles i
    Update velocities:
      $v_i(t) = v_i(t-1) + \rho_1 \times (p_i - x_i(t-1)) + \rho_2 \times (p_g - x_i(t-1))$;
    Move to the new position: $x_i(t) = x_i(t-1) + v_i(t)$;
    If $f(x_i) < f(pbest_i)$ **Then** $pbest_i = x_i$;
    If $f(x_i) < f(gbest)$ **Then** $gbest = x_i$;
    Update(x_i, v_i) ;
  EndFor
Until Stopping criteria

Figure 4.3. Pseudocode of PSO algorithm

Parameters mentioned in the steps above, w, c_1 and c_2, are parameters of the PSO algorithm where their proper tuning can allow the algorithm to adapt to different optimization landscapes. E.g. if there is no social component ($c_2 = 0$), the particle's movement depends only on the best position discovered on their own. Without the social component particles cannot effectively share information and converge into a global solution

. The social component is also called an exploitation factor, hence if the c_2 factor is high, the algorithm converges towards the global best solution which might not be the global optimum. On the other hand, the cognitive factor (c_1) promotes algorithm exploration and where each particle is trying to get closer to their so far best position. If the cognitive factor is zero, the algorithm converges faster but with a high risk of not finding the global optimum. Lastly, the third parameter w is called the weight inertia factor or in some literature, learning rate. It defines how fast particles will adapt to their new position, or how fast they will learn. The optimal combination of those parameters is crucial for algorithm efficiency and good convergence. In this work, the mentioned parameters were set as follows:

$$w = 0.6, c_1 = 2.05, c_2 = 1.7$$

After the PSO algorithm outputs the best parameters for the input image, an enhanced image is used in the further process of skull stripping. The enhanced image is normalized, and by using the Otsu's thresholding method an image is separated into two distinct classes: a foreground class and a background class. Otsu's method is based on finding the optimal threshold value to separate foreground pixels from the background pixels in the bimodal histogram made from the grayscale image. The mentioned threshold needs to minimize the variance within each class and maximize the variance between the classes. After the thresholding process, a connected components analysis is performed to separate the groups of contiguous pixels in the binary image to then identify the largest connected component (skull or brain tissue) within the image. After the previous step, the brain mask is formed simply by setting all the pixels beside the largest object as a background (white), figure below c) (Figure 4.4). Next, the small holes inside the brain mask are filled using morphological opening and median blurring, as shown in figure d) (Figure 4.4). In the last step, the final brain mask is applied to the initial image giving the final result.

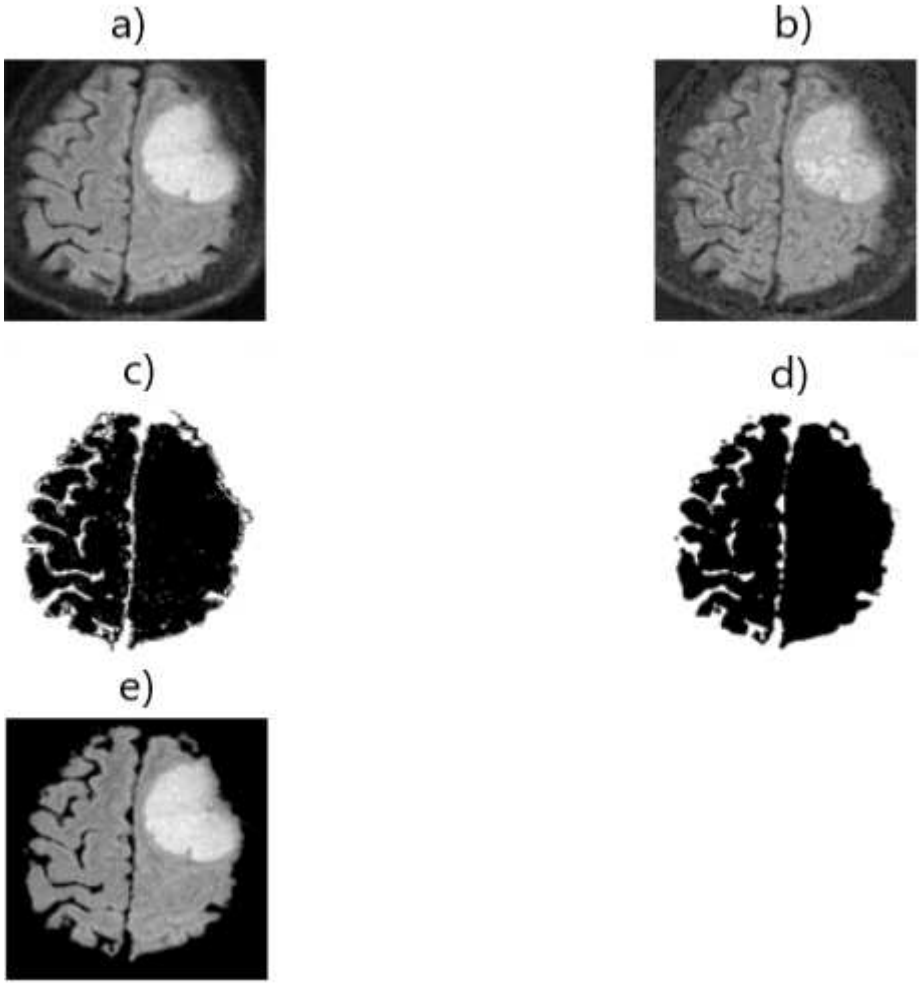

Figure 4.4. Skull stripping process; a) original image, b) enhanced image by PSO algorithm, c) initial brain mask, d) smoothed brain mask, e) final image [25]

To compare how well the proposed method works, the same model was trained on the same split of data (511 train images, and 128 test images), but the processing method was different in 6 scenarios [25]. For simplicity, the lightweight CNN model with only one convolution layer was used as reference. The 6 different scenarios are:

1. No processing, colored images
2. No processing, grayscale images
3. The first processing method, colored images
4. Proposed PSO processing method, colored images
5. Combination of the third and fourth methods, colored images

6. Combination of the fourth and third methods, colored images

The first scenario is used to define why we need image processing at the input of the CNN model, and the second scenario is used to observe is the color within MRI scans significant or not for tumor detection. For each scenario, the metric achieved on test data will be presented as an accuracy score, F1 score and confusion matrix.

1. Scenario: no processing, colored images

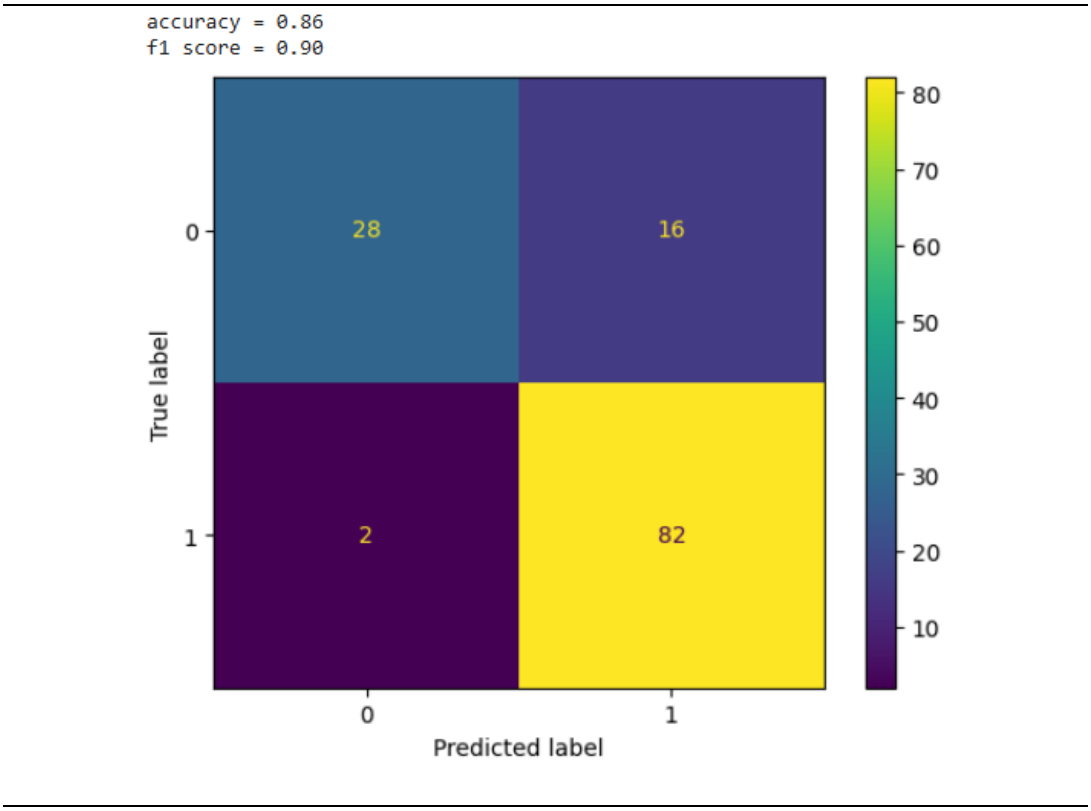

Figure 4.5.1. Final metrics

2. Scenario: no processing, grayscale images

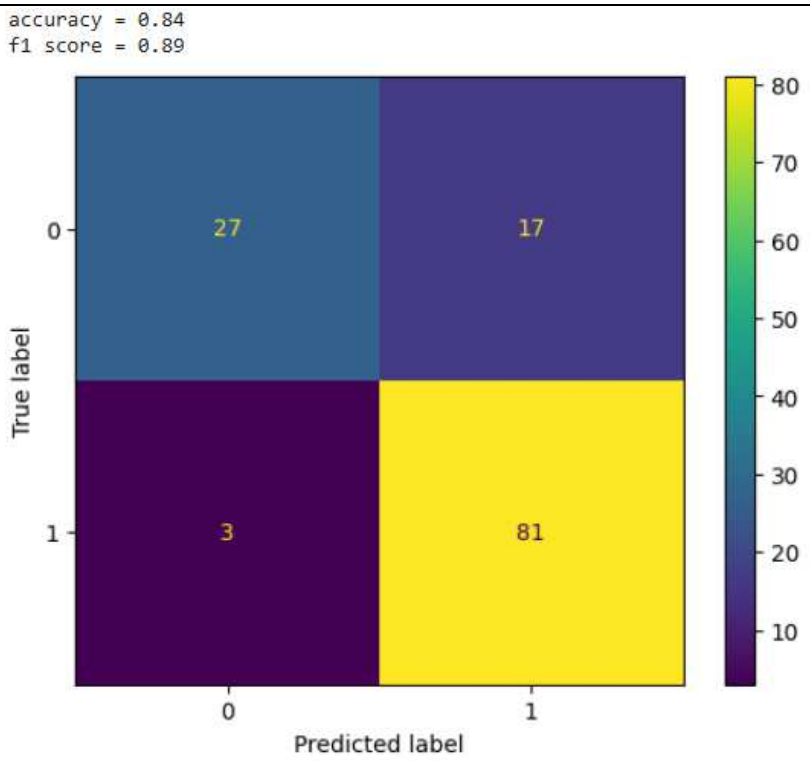

Figure 4.5.2. Final metrics

3. Scenario: first processing method, colored images

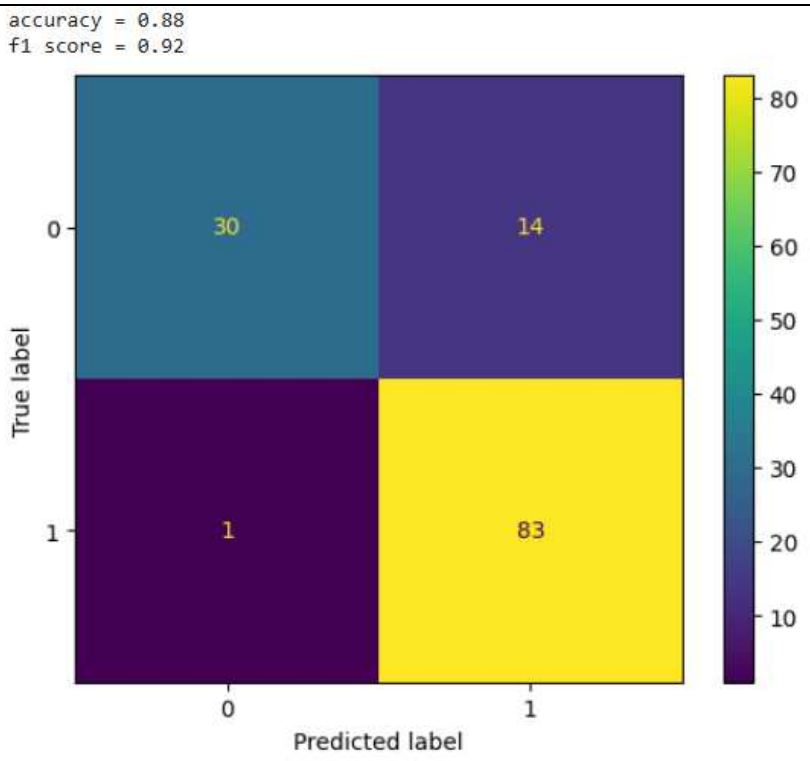

Figure 4.5.3. Final metrics

4. Scenario: proposed PSO processing method, colored images

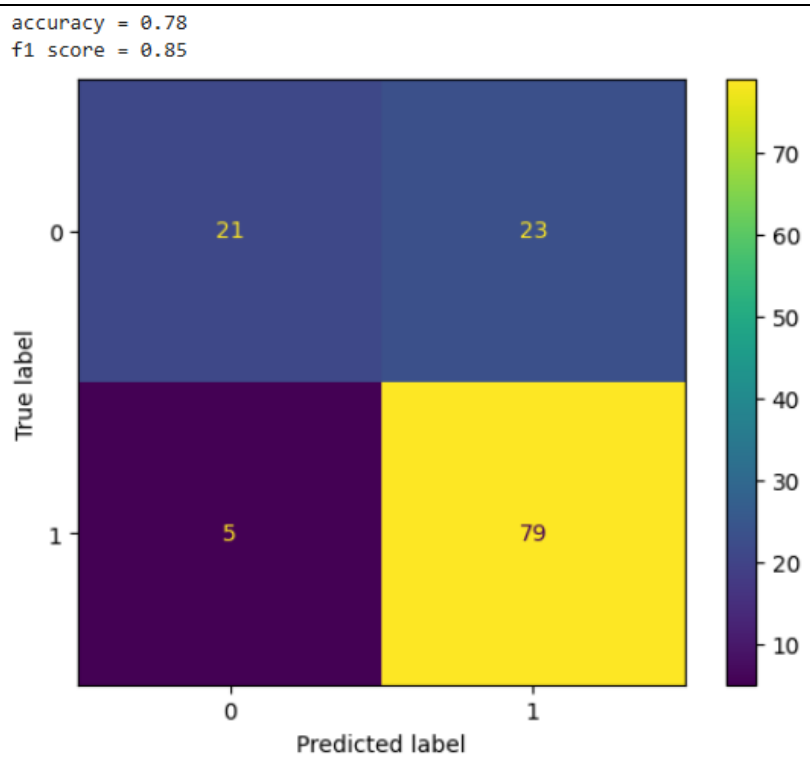

Figure 4.5.4. Final metrics

5. Combination of the third and fourth scenario, colored images

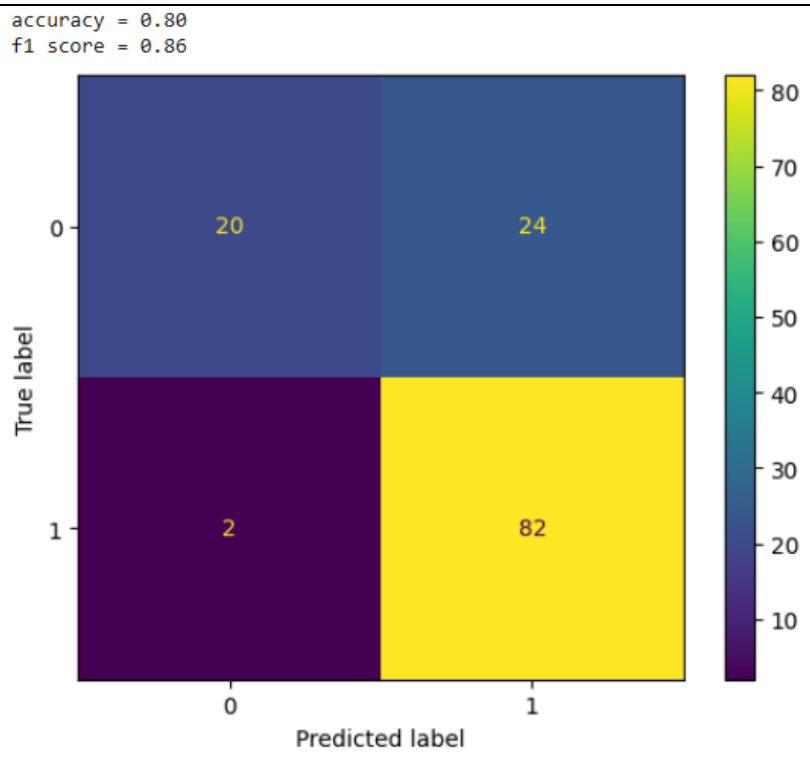

Figure 4.5.5. Final metrics

6. Combination of the fourth and third scenario, colored images

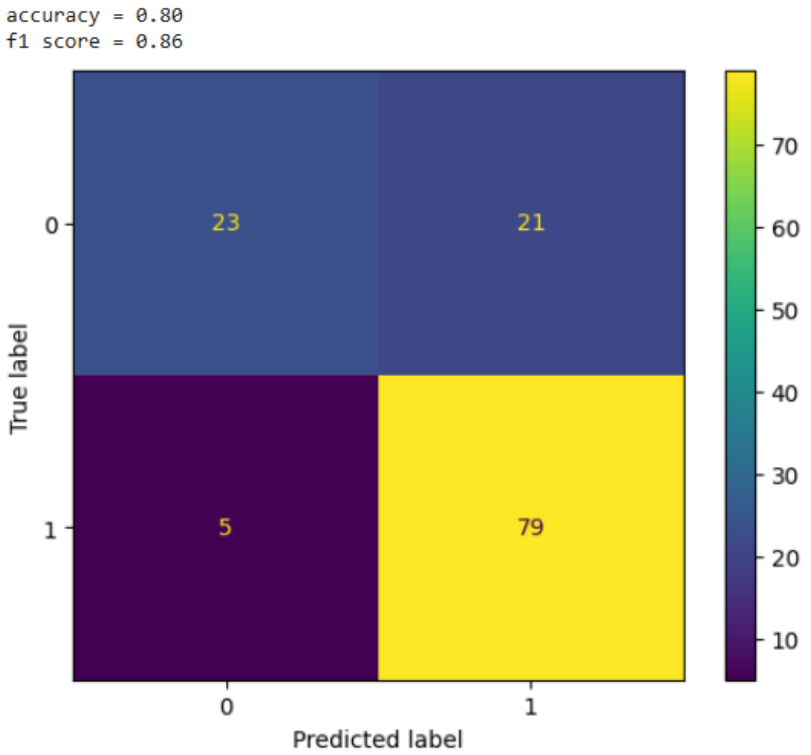

Figure 4.5.6. Final metrics

Table 4.6. Scenarios comparison

	Scenario description	Accuracy	F1 score
Scenario 1	No processing, colored images	0.86	0.90
Scenario 2	no processing, grayscale images	0.84	0.89
Scenario 3	first processing method, colored images	0.88	0.92
Scenario 4	proposed PSO processing method, colored images	0.78	0.85
Scenario 5	Combination of the third and fourth scenario, colored images	0.80	0.86
Scenario 6	Combination of the fourth and third scenario, colored images	0.80	0.86

The Table 4.6. shows the metrics for all scenarios in this study. Turning images to the grayscale did not improve any metrics (scenario 1 versus scenario 2). Thus, for the following scenarios only images with color are used. The third scenario is a simple processing method that finds the largest contour within the image and resizes the original image to a uniform size to fit the object within the largest contour perfectly. Simply put, the method is zooming images to remove noise that is not part of the brain from the background which is not a part of a brain (Figure 4.7.).

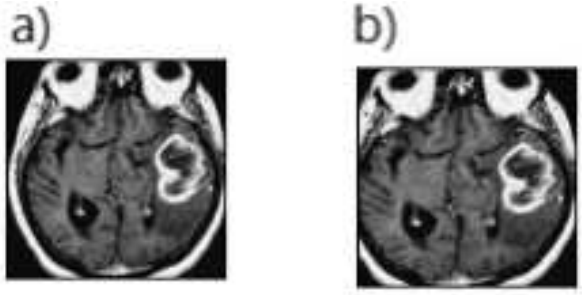

Figure 4.7. Brain zooming method; a) original image, b) processed image

The scenario above outperformed the model with non-processed images by 2% in accuracy and F1 score. The fourth scenario is a proposed PSO processing method, which performed significantly worse even against the non-processed gray images. The figure below shows original images a), and those images processed with the proposed method b) (Figure 4.8.) It can be seen that for some samples algorithm works perfectly, it removes the skull and noise area around brain tissue. But, for some samples, the algorithm got confused and removed parts of a brain tissue, leaving the skull intact. The reason for that is difficulty in separating the background from the foreground. As mentioned in section 3.2., the dataset contains both T1-weighted and T2-weighted MRI scans of different resolutions and quality. The proposed algorithm for skull stripping works well for only one type of MRI scan, since in T2 gray matter is bright and white matter is darker than gray matter, while on the T1 scans gray matter is dark and white matter is bright. The bone is usually bright in both scans. The logic behind the skull stripping algorithm is to find the biggest connected object in the background of an image and remove it. If the logic is applied to the T2 MRI scans, where cerebrospinal fluid also appears bright, the algorithm will have trouble defining what is the background and vice versa. Hence, the algorithm removes some parts of brain tissue with the skull and the cerebrospinal fluid.

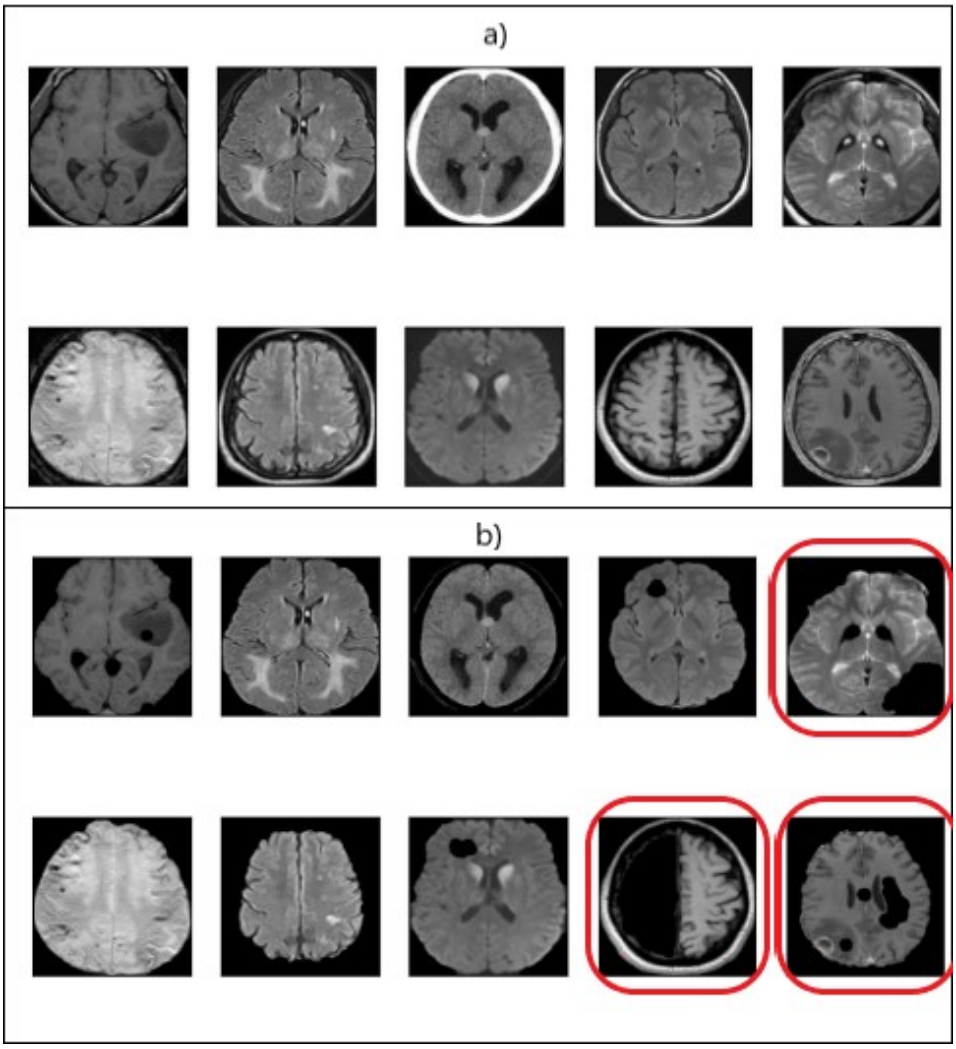

Figure 4.8. PSO processing method visualization; a) original images, b) processed images Samples where algorithm removed a part of a brain tissue are highlighted with red circles

In addition, for the following studies, the chosen segmentation method is brain zooming (scenario 3). The mentioned method performed well in this study (Figure 4.6.) and it was well-suitable for the chosen dataset used in this work. It is worth noting though that separating images into classes (e.g. T1, T2) before their segmentation would improve the results.

4.2. Study 2 – hyperparameter tuning

As shown in section 3.4., finding the optimal hyperparameters for the specified CNN model is challenging. Some computational intelligence is needed to speed up the process and EC fits here perfectly. Three EC algorithms are introduced here to find the optimal combination of the learning rate, maximum number of iterations, the L2 regularization factor and the momentum factor for the SGD optimizer. Those algorithms are, namely:

- Particle Swarm Optimization (PSO)
- Moth Flame Optimizer (MFO)
- Non-linear Levy Chaotic Moth Flame Optimizer (NLCMFO)

A PSO algorithm was already introduced in the first study. The whole implementation can be found in the *PSO.py* file [25]. The code is quite similar to the code for the first study. Some of the differences are: different velocity and position limits and stagnation control (if the algorithm have the same solution for the N iterations, stop the iterations). Optimization class looks like this:

```
class CNN_optimization:
    def __init__(self):
        # load data
        X, y = load_data()
        self.X_train, self.y_train, self.X_test,
                  self.y_test = train_test_split(X, y, test_size=0.2)

    def get_metrics(self, N: np.ndarray) -> tuple:
        optimizer = keras.optimizers.SGD(learning_rate=N[0], momentum=N[3])
        model = get_CNN_model(optimizer, l2_reg = N[2])
        return train(model, self.X_train, self.X_test, self.y_train, self.y_test,
epochs=int(N[1]), verbose=False)
```

Figure 4.9. CNN optimization class with *get_metrics* method

A *get_metrics* function will train a proposed CNN model (introduced in the third section), using the SGD optimizer with selected hyperparameters as an input to the method, and test the model using the test data. The method returns the accuracy and the F1 binary score. The mentioned method is used for all of three EC algorithms introduced in this section and called when the evaluation of a particle's or a moth's position is needed. It is important to note that the train-test split is made only once, at the beginning of the optimization process. In any other scenario, it would not make sense. Two versions of the PSO algorithm are observed in

this study, one with the global neighborhood and another with a chain topology neighborhood. In general, it is observed that PSO algorithm with global neighborhood can easily converge too early in some of the local optima. In this study, the number of iterations is limited to 30 due to the limited computational power and resources. The first version with the global neighborhood with the following parameters:

- Swarm size: 20
- Max. iterations: 30
- c1: 2
- c2: 1.75
- w: 0.5

The algorithm's initial best position overall had an F1 score of 87%, whereas the last best position had an F1 score of 93%. The PSO algorithm with global neighborhood optimized the mentioned hyperparameters and boosted the F1 score by more than 6% within 15 iterations. The algorithm stagnated on the 15th iteration (Figure 4.10.)

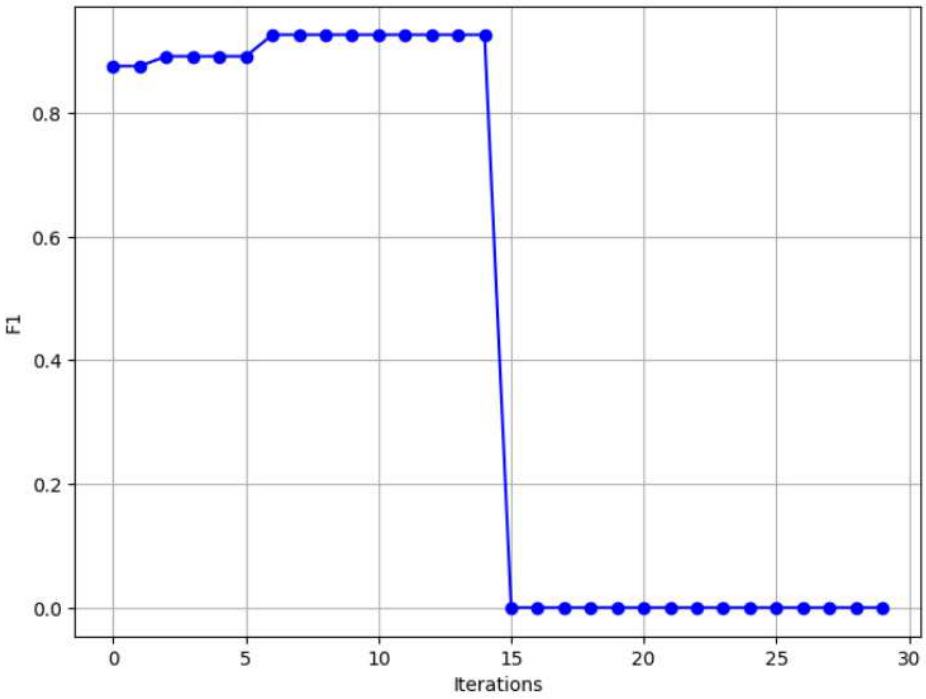

Figure 4.10. PSO with global neighborhood score over iterations

Another interesting fact to observe in these algorithms is the movement of a single particle. The graph below represents the F1 score of every position of the first particle in the swarm (Figure 4.11.) From the movement of the selected particle, by the 6th iteration, the particle was exploring the search space. After it found its personal best position, it tried to focus close to that position (iterations 6 to 11).

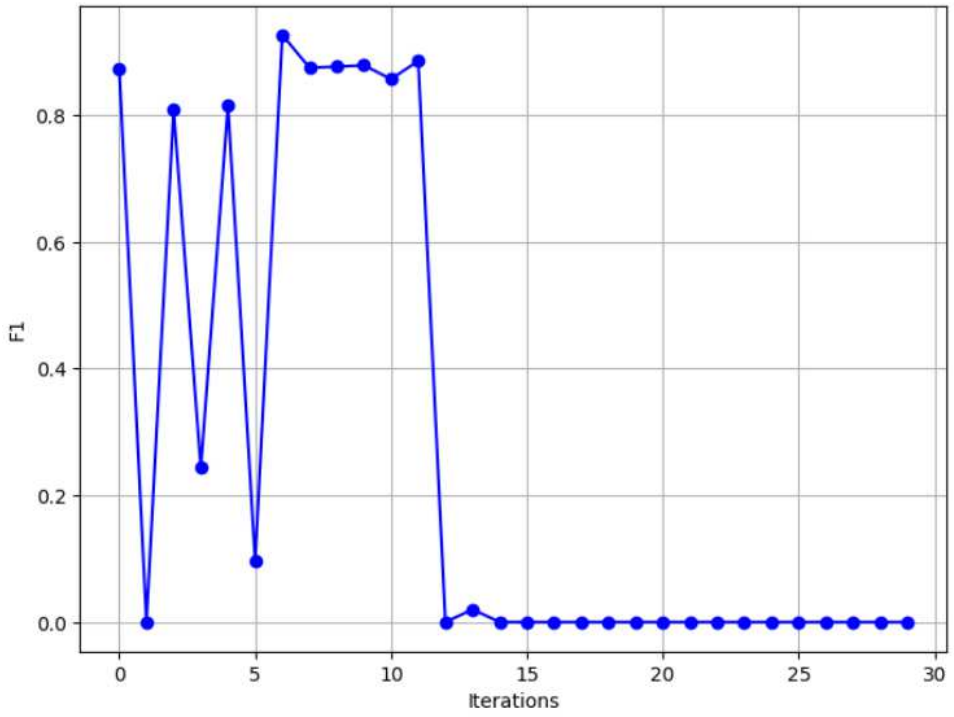

Figure 4.11. Score over iterations for a single particle within a swarm

The whole experiment with the PSO algorithm is available on GitHub by the name of *Brain_tumor_detection_PSO.ipynb.* Another instance of the PSO algorithm, but with a local neighborhood showed a similar performance. The algorithm's parameters were the same, and the algorithm stagnated once again on the 15th iteration (Figure 4.12). The F1 score rose from 84% to 90%. The only difference was the stability of a single particle since it was updated based only on its left and right neighbors (Figure 4.13). For this problem, there was no significant difference between the two versions of the same algorithm. The difference would be visible only with a higher number of iterations. Unfortunately, it takes around 10 minutes for every iteration of the algorithm on the computational resources used for this work. For the future reference, a bigger number of iterations would surely be a good path to pursue.

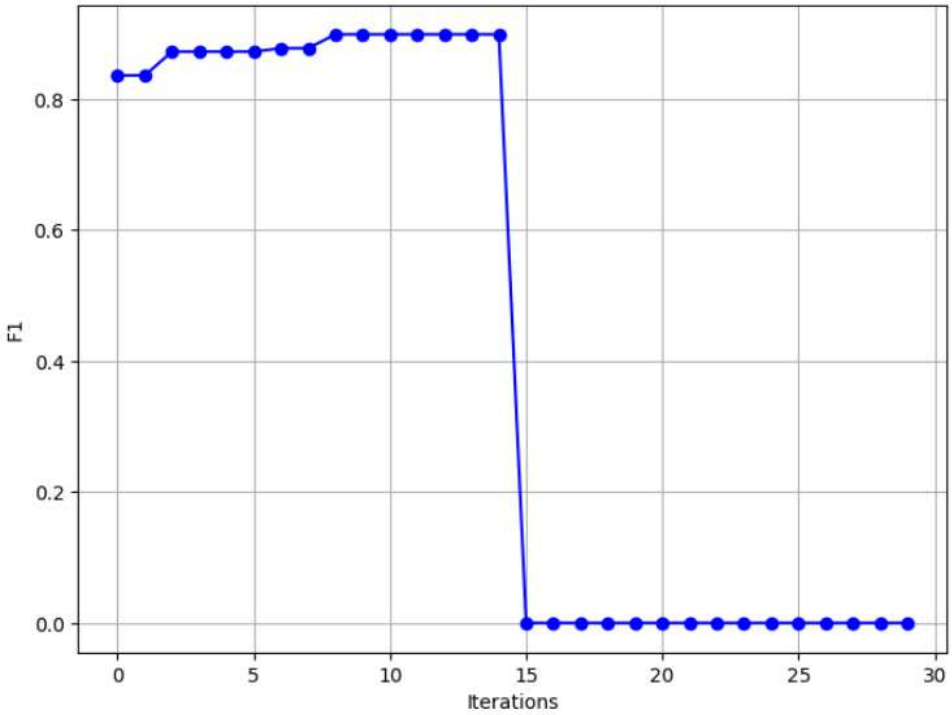

Figure 4.12. PSO with local neighborhood score over iterations

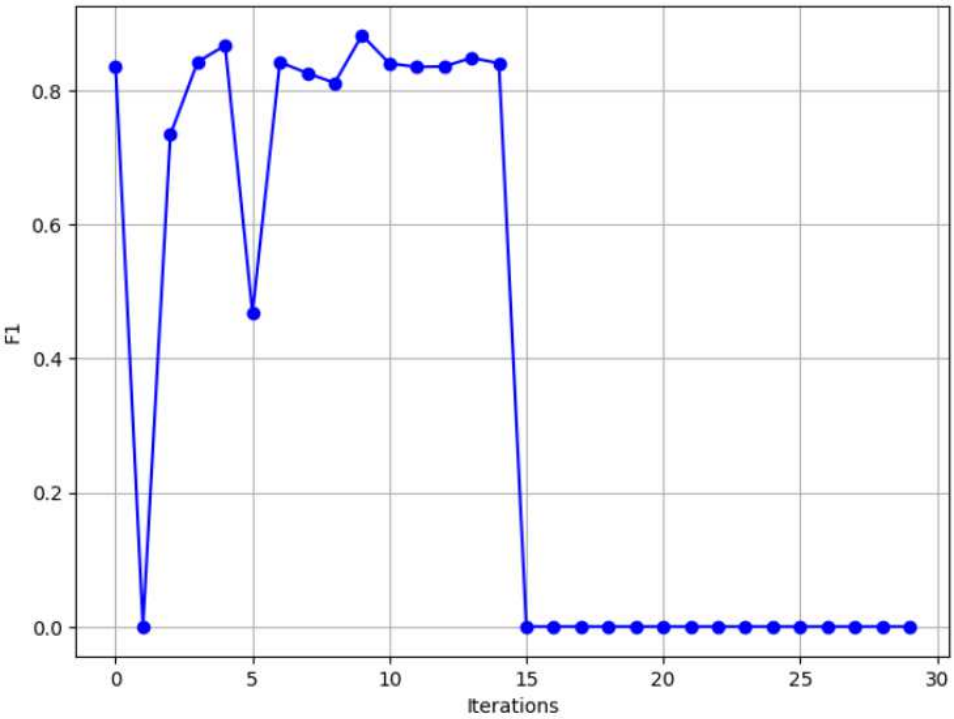

Figure 4.13. Score over iterations for a single particle within a swarm

Another algorithm used in this study was introduced by Dehkordi et al. [19] in their paper where it outperformed other optimization algorithms such as PSO, GA, GWO – Nonlinear Levy Chaotic Moth Flame Optimizer (NLCMFO). But before explaining that part, it is necessary to understand the base version of the algorithm, a simple Moth Flame Optimizer (MFO).

MFO, population-based algorithm introduced by Mirjalili [27], mimics the transverse orientation for particle navigation. The mentioned technique is used by moths at night. In this algorithm, moth behavior is presented as an optimization technique where every moth moves following the spiral movement around the flames. The algorithm begins similarly to PSO, by randomly producing moths in the search space and evaluating their position considering the optimal position with the flame. The moths are represented as search agents that explore the search space, and the flames are represented as their flags or pins in their way to find a better solution in the search space (Figure 4.14).

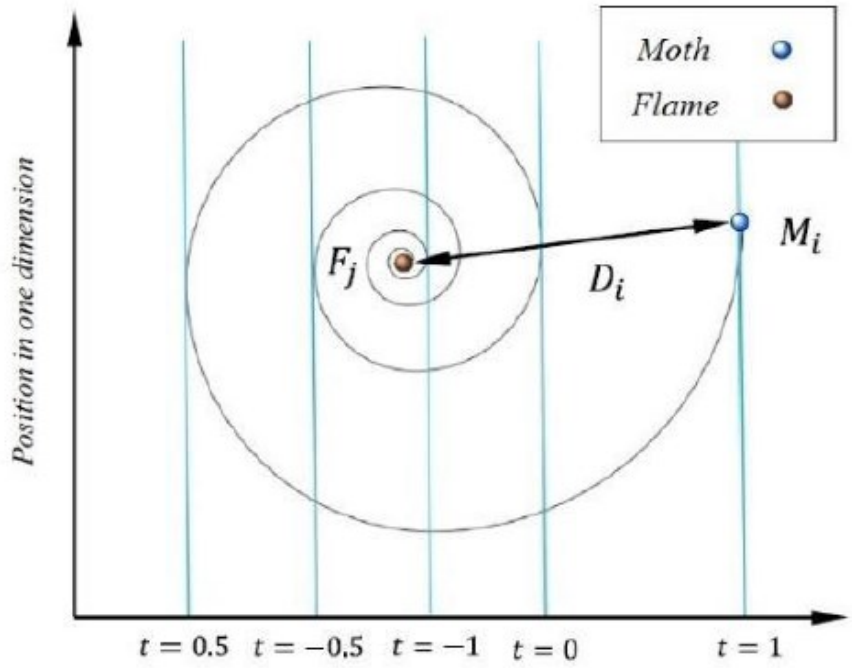

Figure 4.14. The spiral movement of a moth Mi towards the flame Fj

The spiral motion of the moths around the flame ensures the balance between exploration and exploitation in the MFO algorithm. A flame number is calculated dynamically over the iterations following the equation:

$$flame\_no = round(N - l * \frac{N - l}{T})$$

Where N represents the maximum number of flames (given to the algorithm on the start), l is a current iteration and T is a maximal number of iterations (also given to the algorithm on the start). The moth's position is updated by the following equation:

$$S(M_i, F_j) = D_i * e^{bt} * \cos(2\pi t) + F_j$$

Where the new position of a moth Mi regarding the flame Fj, is represented as $S(M_i, F_j)$. D_i is the distance between i^{th} moth and j^{th} flame, b is a constant value used to define the logarithmic helix space, and t represents a random number between [-1, 1]. The equation for getting the t value is following:

$$t = (a - 1) * rand + 1$$

Where a is calculated for every iteration following the equation:

$$a = -1 + T_i * \left(\frac{-1}{T}\right)$$

Here, T_i represents the i^{th} iteration, and T represents a maximal number of iterations. This helps the algorithm to know when to explore space and when to converge to the current best solution. The implemented version of the algorithm was built on top of these equations. Furthermore, a stagnation check was added and another check to see if the moth went out of the search space. If true, its position was set within the borders of a search space. A Python implementation of the MFO algorithm used NumPy arrays to store the moth's and flame's positions. Where sorting those arrays would present the best so-far positions within the search space. The flames are updated based on the best solution so far, and moths are updated based on the logarithmic spiral. The process is repeated until the convergence or stopping criterion is met. In this study, the stopping criteria is a maximal number of iterations – 10. The algorithm is robust to getting stuck in local optima because of the dynamic number of flames which gradually reduces over iterations. The mentioned behavior prevents premature convergence. Another key factor is the simplicity of the algorithm, every moth has to know only his position (no personal best or global best like in PSO). An overview of the implementation is available in the files named *MFO.py* and *Brain_tumor_detection_MFO.ipynb* [25].

The initial parameters of the MFO algorithm were:

- number of moths: 30
- max. number of iterations: 10
- max number of flames: 15
- b: 1

The optimization method boosted the F1 score from 86% to 90%. The total execution time was around 150 minutes, similar to PSO's time for 15 iterations. The graph below shows the F1 score for the best position found in every iteration (Figure 4.15).

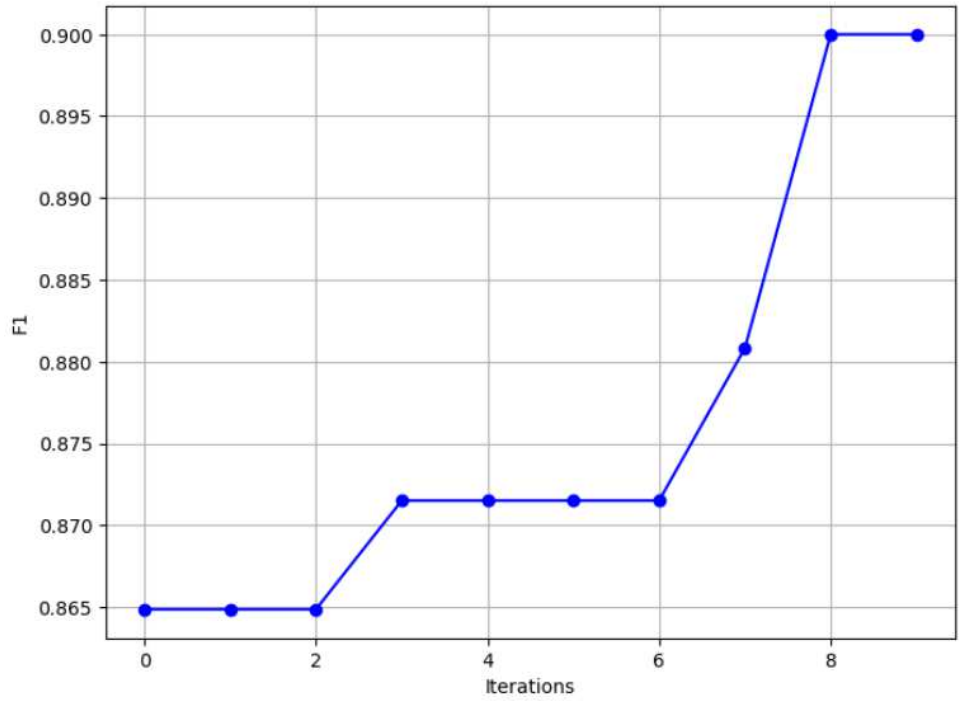

Figure 4.15. MFO score over iterations

As mentioned before, to solve some drawbacks mentioned in the paper [19], the NLCMFO algorithm has been introduced as quoted "*The primary purpose of the NLCMFO is to increase the performance of the standard MFO in two segments. The initial stage is to integrate MFO with Levy flight theory and chaotic maps, followed by employing the nonlinear weight coefficient parameter as a control variable between both the algorithm's exploration and exploitation processes.*"

Chaotic maps play a crucial role in optimization algorithms and not only NLCMFO one, by introducing deterministic, yet highly unpredictable behavior, making them a powerful alternative to randomness. Unlike random parameters, chaotic maps leverage the non-repetition and ergodicity of chaos to systematically and thoroughly explore the search space. The use of chaotic maps helps optimization techniques improve their exploration capabilities, covering diverse regions of the search space. This approach also helps in avoiding local optima and accelerates convergence by maintaining a dynamic and non-linear exploration pattern. In this thesis, the chosen chaotic map is a logistic map with the following equation:

$$X_{n+1} = r * X_n(1 - X_n)$$

Where r was a random value set to 3.99 in this study.

Levy flight theory, a stochastic step method was used to control the step size depending on the probability distribution function generated by the Levy distribution [28] followed by the equation:

$$L(x_i) \approx |x_i|^{1-\alpha}, \ 1 < \alpha \leq 2$$

Lewy flight is generally connected with small steps, and rarely with long jumps. Hence, it is more suitable for the optimization field than, for example, a uniform random search [29]. Overall, NLCMFO showed better performance and stability over the MFO algorithm both in the mentioned paper [19], and in this thesis. The algorithm implementation is very similar to the MFO algorithm implementation, with a few digressions:

- in the equation for the moth position update, instead of a b value the levy step over a multiplied by alpha (value of 1.8 in this work)
- in the same equation, instead of a t value (the current iteration of the algorithm), a logistic map has been applied to the randomly chosen number and constant r of 3.99

The final equation for updating the moth position is the following:

$$S'(M_i, F_j) = D_i * e^{levy(a*1.8)} * \cos(2\pi * \text{logMap}(x, r)) + F_j$$

Where x represents a random value chosen at the beginning of the algorithm (before the first iteration). The whole implementation is available by the name of *NLCMFO.py* and *Brain_tumor_detection_NLCMFO.ipynb* [25].

The algorithm improved F1 score by almost 10% over only ten iterations as shown in the graph below (Figure 4.16.)

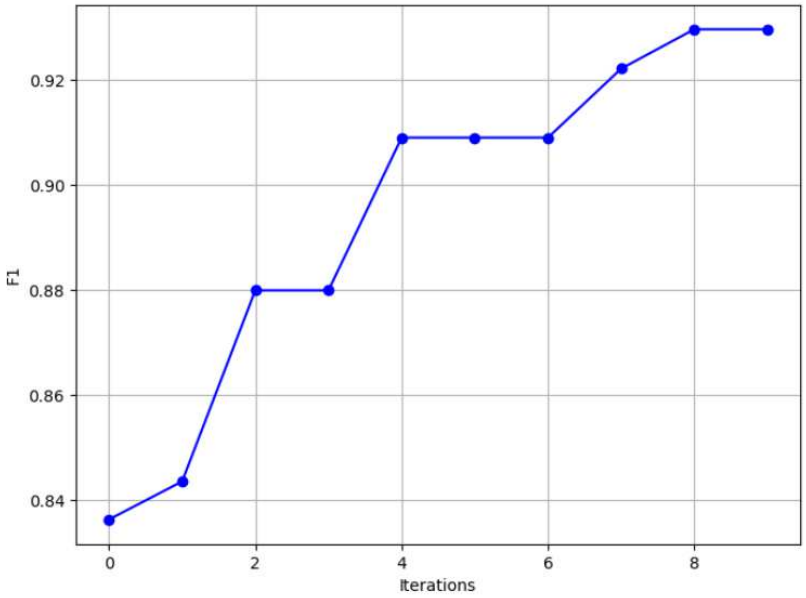

Figure 4.16. NLCMFO score over iterations

Accordingly, the algorithm found a better solution in almost every new iteration, which demonstrates the great ability to explore the given search space compared to PSO and MFO algorithms. The graph below shows the motion of a single moth over the same iterations (Figure 4.17). It can be observed that the specified moth improved its starting position (fourth iteration), and stayed close to the best, optimal position.

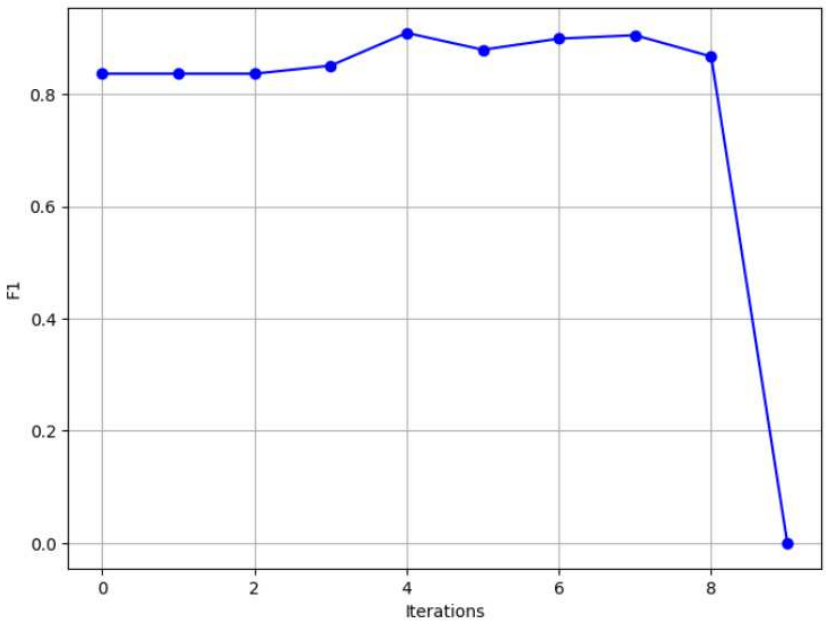

Figure 4.17. Single moth score over iterations

The next study will show an even better comparison between MFO and NLCMFO algorithms. In conclusion, all three algorithms have the same time and space complexity. The PSO algorithm uses a bit more memory, given the fact it has to store all particle's best positions and their velocities. But on the other hand, MFO and NLCMFO algorithms have to perform sorting operations. The NLCMFO algorithm showed better performance by boosting the F1 score by almost 10%, whereas other algorithms did the same by only 6%. All three algorithms are scalable for bigger problems and a greater number of iterations. For future reference, a better comparison of different search spaces is needed.

Table 4.18. Metrics of different CNN models on a subset

	Accuracy	F1 score
CNN – without EC	0.63	0.71
CNN – PSO	0.88	0.93
CNN - MFO	0.86	0.9
CNN - NLCMFO	0.89	0.93

4.3. Study 3 – CNNs loss function optimization

Traditional algorithms like Stochastic gradient descent (SGD), Adam, Adagrad, and RMSprop have become the standard for optimizing CNN loss function. However, these methods are often taken for granted, even though they rely on gradient information, making them prone to challenges such as getting stuck in local optimum, sensitivity to hyperparameters, or limited exploration. EC algorithms present a suitable alternative for optimizing CNN loss functions. Unlike gradient-based methods, EC algorithms are gradient-free, enabling them to effectively navigate rugged, high-dimensional search spaces without relying on the differentiability of the loss surface. By leveraging mechanisms such as population-based exploration, mutation, and crossover, EC algorithms can explore diverse regions of the search space, avoid local optima, and discover more robust solutions. This makes EC particularly suitable for optimizing CNN loss functions. Overall EC algorithm's time to converge is longer than in traditional gradient-based algorithms, but it may outperform when gradients are noisy, or the loss function surface is highly non-convex. EC methods have a high computational cost per iteration, and this was the case in this study as well. For simplicity, a simple artificial NN was created for the purposes of this study. A 10-dimensional input was connected to a hidden layer with five nodes, which was connected to an output layer with two nodes. Next, a dummy data was created according to the code below:

```
inputs = np.random.randn(5, 10)
labels = np.array([0, 1, 0, 1, 0])
```

Figure 4.19. Dummy data generating

For the study purpose, three EC versions were implemented again – PSO, MFO and NLCMFO algorithms. The implementation was exactly the same as the one in the study before, with a few minor modifications so it would fit the problem of this study. The purpose of this study was to observe how well EC algorithms are performing compared to the famous gradient-based algorithms. Furthermore, the main difference between PSO with a global neighborhood and PSO with a local neighborhood is shown, such as a difference between MFO and NLCMFO algorithms. For that purpose, a Python method was developed to compare the average loss of different optimizers over multiple iterations on the same ANN. The whole implementation is available in the file named *Loss_function_opt.ipynb* [25]. The

graph below displays the results of this study (figure 4.19.)

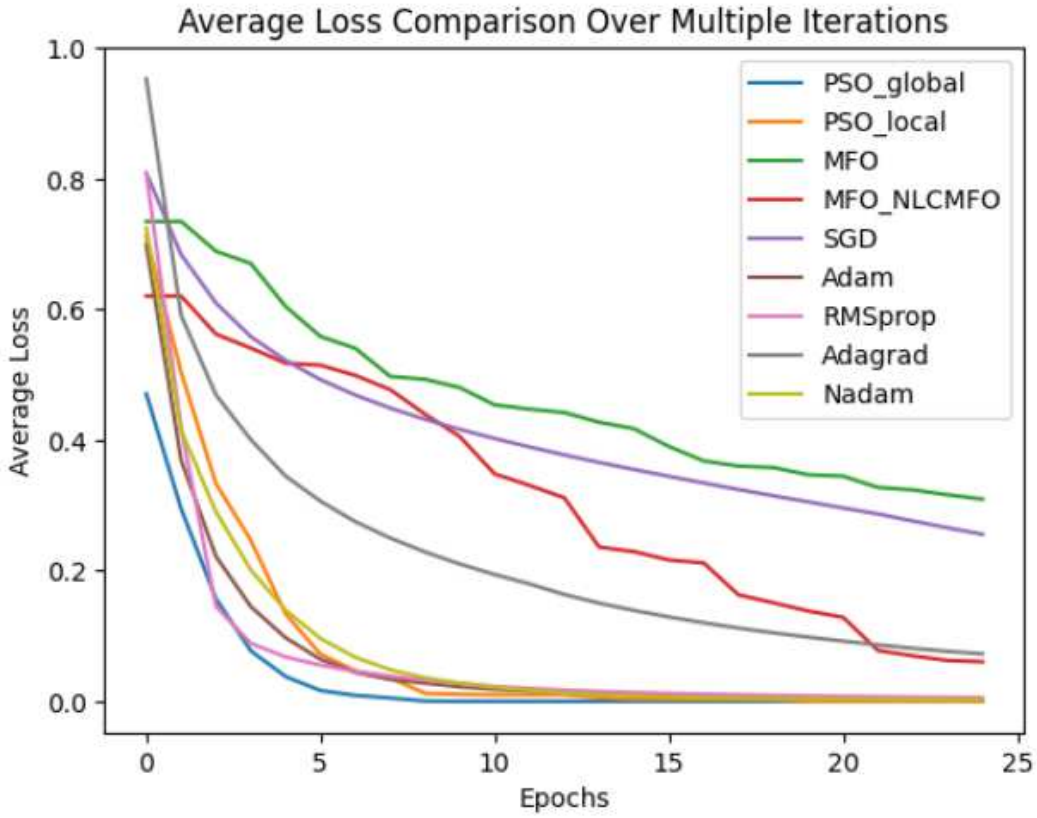

Figure 4.20. Average loss comparison on different optimizers over 25 epochs within multiple iterations

In case of this simple problem, the learning process of a simple ANN, PSO algorithm with a global neighborhood achieved very impressive results over 25 epochs within 10 different iterations. Every optimizer performed a learning step over 25 epoch 10 times, with the average displayed on the graph above. PSO algorithm with global neighborhood (light blue line) scored the lowest average loss and converged fastest. Within only five epochs, the algorithm converged. A PSO algorithm with a chain topology, scored impressive results as well, with a slightly slower convergence (orange line). A global neighborhood forces the algorithm to explore the search space more aggresively by guiding all particles toward the global best solution, which is why the algorithm converges faster. This is not always a good thing, if an algorithm is stuck in the local optimum, it may never get out of it. Green line shows the performance of a basic MFO algorithm is displayed. Clearly, it performance is the worst compared to other optimizers. However, the performance of the algorithm was boosed

significantly by the slight modification of the simple MFO algorithm, i.e. introduction oflogistic maps and levy flights. The NLCMFO algorithm outperformed two commonly used optimizers, SGD and Adagrad, within 20 epochs for this single problem. Because of limited resources and computational power, the mentioned comparison was not suitable for the optimization of the CNN used in this work. Hence, the optimizer used to train the mentioned CNN model was still SGD, a gradient-based method. The following section will show the comparison between the mentioned CNN model and the VGG-16 pre-trained model on a whole dataset.

4.4. Final CNN model

In the processes of input image segmentation and the model's loss function optimization, no EC was used for the reasons stated in previous sections. EC was used only to find optimal hyperparameters of a mentioned CNN model. The process was done on a smaller subset, which means those optimal hyperparameters should be tested on the whole dataset. From the third chapter and graph comparison (Figure 3.8.), the proposed model achieved F1 score of only 71% on a subset of the dataset. In the second study, the same model was optimized by three different optimization algorithms: PSO, MFO and NLCMFO. The NLCMFO algorithm found the following best hyperparameters:

- Learning rate: 0.0946
- Momentum: 0.5815
- Optimizer: SGD
- L2 regularization factor: 0
- Number of iterations: 20

With these it achieved the F1 score of 93%. In the final notebook, the mentioned CNN model with optimal hyperparameters is trained on the whole dataset (total of 3509 images) and compared to the VGG-16 pre-trained model. The results are shown in the table below (Table 4.20.)

Table 4.21. Introduced CNN and VGG-16 model final comparison

	Accuracy	F1 score
CNN	0.95442	0.97283
VGG-16	0.96011	0.97623

The CNN model showed similar performance as the pre-trained VGG-16 model on this specific data split. However, more important is the significant improvement of the F1 score within the CNN model with different hyperparameters. In fact, it ensures that using the optimal hyperparameters can boost the F1 score by more than 15%. Both models remain insufficiently accurate for the task of simple binary classification, primarily due to the imbalanced and noisy nature of the chosen dataset.

Conclusion

This thesis explores the potential of EC techniques was explored, specifically in the context of brain tumor detection using CNNs. The early detection of brain tumors is crucial for improving patient outcomes, as it enables timely intervention and treatment. AI models, particularly CNNs, have shown promising results in medical image analysis, offering accurate, efficient, and automated detection systems. The incorporation of EC methods, such as Particle Swarm Optimization (PSO), Moth Flame Optimization (MFO), and Nonlinear Levy Chaotic Moth-Flame Optimization (NLCMFO), has the potential to further enhance the performance of CNNs in this domain.

In this research, the latest state-of-the-art methods using EC techniques for brain tumor classification and detection were compared and implemented, shedding light on their applications and the improvements they bring. Various approaches were assessed, from optimizing CNN hyperparameters, enhancing segmentation and loss functions to achieving better convergence speeds and avoiding premature convergence. Studies such as Dehkordi et al. [19] and Zhang et al. [18] have demonstrated the effectiveness of EC methods in optimizing CNN architectures and training processes, achieving high accuracy rates in brain tumor detection.

Motivated by this fact, this research used a simple CNN model with two convolutional layers, max pooling, and dropout layers. The model achieved an F1 score of 71% on a subset of the dataset which has been previously merged and processed. Later, the same model achieved an F1 score of 97% with optimal hyperparameters on the whole dataset. The integration of evolutionary computing further improved the performance of the model. Firstly, the PSO was employed for image segmentation, but the results were insufficient for further use. Next, the CNN hyperparameters were fine-tuned using MFO, PSO, and NLCMFO. The NLCMFO algorithm significantly improved the F1 score, achieving an impressive performance boost of over 15%. Additionally, the research explored the optimization of the loss function using MFO, PSO, and NLCMFO, although computational limitations hindered the applicability to the CNN model. However, the results were very interesting and they confirmed the theoretical foundation behind the mentioned optimization algorithms.

A comparison between the CNN model optimized with NLCMFO hyperparameters and the pre-trained VGG-16 model showed similar performance, further validating the effectiveness of the proposed approach. While the results are promising, there is still considerable room for improvement and further research. Future work should focus on extending and cleansing the dataset, exploring 3D MRI scans, and adapting the models for real-time applications in clinical settings. Furthermore, optimizing these models for larger and more diverse datasets in high-performance computing environments could lead to even more accurate and robust tumor detection systems.

In conclusion, the integration of evolutionary computing with deep learning models such as CNNs has shown great potential for enhancing brain tumor detection, but continued research is necessary to fully leverage these techniques and deploy them effectively in clinical practice.

Literature

[1] Mayo Clinic, *Brain tumor: Overview*, (2024, December). URL: https://www.mayoclinic.org/diseases-conditions/brain-tumor/symptoms-causes/syc-20350084; accessed 21. December 2024.

[2] Judas M., Kostovic I., *Temelji neuroznanosti*, Zagreb: MD, 1997.

[3] Chang H., Valentino D.J., Duckwiler G.R., Toga A.W., *Segmentation of brain MR images using a charged fluid model,* IEEE, 54, 10 (2007), pages 1798-1813

[4] El-Dahshan et al., *Computer-aided diagnosis of human brain tumor through MRI: A survey and a new algorithm,* ScienceDirect, 41, 11 (2014), pages 5526-5545

[5] Lloyd-Jones G., *Radiology masterclass – MRI interpretation*, (September 2017). URL: https://www.radiologymasterclass.co.uk/tutorials/mri/mri_scan; accessed 21. December 2024.

[6] Sajedi H., Pardakhti N., *Age prediction based on brain MRI image: A survey,* Springer, 43, 8 (2019), pages 279

[7] Wiering M., Otterlo M., *Reinforcement learning*, Berlin: Springer, 2012.

[8] Goodfellow I., Bengio Y., Courville A., *Deep learning*, United States: MIT Press, 2016.

[9] Muller B. et al., *Neural networks: An introduction*, Berlin: Springer, 1991.

[10] Towards data science, *what's the role of weights and bias in a neural network?* (2020, July). URL: https://towardsdatascience.com/whats-the-role-of-weights-and-bias-in-a-neural-network-4cf7e9888a0f ; accessed 26. December 2024.

[11] El-Ghazali T., *Metaheuristics – From design to implementation*, United States: John Wiley & Sons, 2009.

[12] Darwish A. et al., *A survey of swarm and evolutionary computing approaches for deep learning*, Springer, 53, 1 (2019), pages 1767-1812

[13] Yao X, *Evolving artificial neural networks,* IEEE, 87, 9 (1999), pages 1423-1447

[14] Leung F. et al., *Tuning of the structure and parameters of a neural network using an improved genetic algorithm,* IEEE, 1, 1 (2002), pages 25-30

[15] Cheung B., Sable C., *Hybrid evolution of convolutional networks,* IEEE, 1 (2011), pages 293-297

[16] Fujino S., Mori N., Matsumoto K., *Deep convolutional networks for human sketches by means of the evolutionary deep learning,* IEEE (2017), pages 1-5

[17] Khalifa M. et al., *Particle swarm optimization for deep learning of convolution neural network,* IEEE (2017), pages 1-5

[18] Zhang Y. et al., *Pathological brain detection in MRI scanning by wavelet entropy and hybridization of biogeography-based optimization and particle swarm optimization*, Springerplus, 4, 1 (2015), pages 716

[19] Dehkordi A., Hashemi M., Neshat M., Mirjalili A., Sadiq A., *Brain tumor detection and classification using a new evolutionary CNN,* SSRN (2022), pages 53

[20] Mahesh, K., Renjit A., *Evolutionary intelligence for brain tumor recognition from MRI images: a critical study and review*. Springer, 11 (2018), pages 19-30

[21] Anaconda, *2022 State of Data Science* (2022, May). URL: https://www.anaconda.com/resources/whitepapers/state-of-data-science-report-2022 ; accessed: 6. January 2025.

[22] Chakrabarty N., Brain MRI Images for Brain Tumor Detection, Kaggle (2019). URL: https://www.kaggle.com/datasets/navoneel/brain-mri-images-for-brain-tumor-detection/code ; accessed: 6. January 2025.

[23] Kanchan S., Chakrabarty N., Bhuvaji S. et al., Brain Tumor Classification (MRI), Kaggle (2020). URL: https://www.kaggle.com/datasets/sartajbhuvaji/brain-tumor-classification-mri ; accessed: 6. January 2025.

[24] Keras official documentation, *ADAM optimizer class.* URL: https://keras.io/api/optimizers/adam/ *;* accessed: 6. January 2025.

[25] Github materials, *Brain tumor detection – Final thesis* (2025, February). URL: https://github.com/Jandjurinec/FinalThesis

[26] Gorai A., Ghosh A., *Gray-level image enhancement by Particle Swarm Optimization*, IEEE, 1 (2009), pages 72-77

[27] Mirjalili S., *Moth-flame optimization algorithm: A novel nature-inspired heuristic paradigm*, ScienceDirect, 89, 1 (2015), pages 228-249

[28] Siegrist K., 5.16: *The Lévy Distribution* (2022, April). URL: https://stats.libretexts.org/Bookshelves/Probability_Theory/Probability_Mathematical_Statistics_and_Stochastic_Processes_(Siegrist)/05%3A_Special_Distributions/5.16%3A_The_Levy_Distribution ; accessed: 12. January 2025.

[29] Faramarzi A. et al., *Marine Predators Algorithm: A nature-inspired metaheuristic,* ScienceDirect, 152, 1 (2020), pages 113377

Summary

Title: Evolutionary computing optimization of neural network for tumor detection in brain MR images

Key words: Brain tumor detection, Convolutional neural networks, Evolutionary computing

This thesis explores the application of evolutionary computing (EC) techniques to optimize convolutional neural networks (CNNs) for brain tumor detection in MRI scans. EC techniques were used for image preprocessing, hyperparameter tuning, and loss function optimization. Using a dataset of 3509 MRI scans, initial CNN F1 score was 71% (accuracy: 63%). Three EC algorithms—PSO, MFO, and NLCMFO—were tested, with NLCMFO boosting the F1 score to 97%. While EC-assisted skull stripping showed mixed results, simpler preprocessing proved more reliable. A final comparison between the optimized CNN and the pre-trained VGG-16 model showed similar performance. This result highlights the potential of evolutionary algorithms to enable custom CNNs to perform comparably to state-of-the-art pre-trained models in medical image analysis. Despite computational costs and dataset challenges, EC shows promise for enhancing medical image analysis, warranting further research.

Summary in Croatian

Naslov: Optimiranje neuronske mreže evolucijskim računanjem za detekciju tumora u MR slikama mozga

Ključne riječi: Detekcija tumora mozga, konvolucijske neuronske mreže, evolucijsko računarstvo

Rad istražuje primjenu tehnika evolucijskog računanja (ER) za optimizaciju konvolucijskih neuronskih mreža (CNN) u detekciji tumora mozga na MRI snimkama. ER metode korištene su za obradu slika na ulazu modela, podešavanje hiperparametara i optimizaciju funkcije gubitka. Na skupu podataka od 3509 MRI snimaka, početna F1 mjera CNN-a iznosila je 71% (točnost: 63%). Testirana su tri ER algoritma: PSO, MFO i NLCMFO – pri čemu je NLCMFO povećao F1 mjeru na 97%. ER metoda uklanjanja lubanje nije pokazala značajnije rezultate. Konačna usporedba optimiziranog CNN-a s unaprijed treniranim VGG-16 modelom pokazala je slične performanse. Navedeni rezultati naglašavaju potencijal ER algoritama za optimiranje CNN modela kako bi postigli rezultate usporedive sa *state-of-the-art* modelima. Unatoč računalnim troškovima i izazovima vezanim uz podatke, ER pruža čvrste temelje za daljnja istraživanja u medicinskoj analizi slika i općenito.

Abbreviations

ACO *Ant Colony Optimization*
ANN *Artificial Neural Network*
AI *Artificial Intelligence*
CNN *Convolutional Neural Network*
CT *Computed Tomography*
DL *Deep Learning*
DWT *Discrete Wavelet Transform*
EC *Evolutionary Computing*
FN *False Negative*
FP *False Positive*
GA *Genetic Algorithm*
GP *Genetic Programming*
GWO *Grey Wolf Optimization*
MFO *Moth Flame Optimization*
ML *Machine Learning*
MR *Magnetic Resonance*
MRI *Magnetic Resonance Imaging*
fMRI *Functional Magnetic Resonance Imaging*
MSE *Mean Squared Error*
NBTF *National Brain Tumor Foundation*
NLCMFO *Nonlinear Levy Chaotic Moth Flame Optimization*
NN *Neural Network*
NP *Non-deterministic Polynomial-time*
PC *Personal Computer*
PET *Positron Emission Tomography*
PSO *Particle Swarm Optimization*
ReLU *Rectified Linear Unit*
SVM *Support Vector Machine*
TN *True Negative*
TP *True Positive*
VGG *Visual Geometry Group*