METAHEURISTIC ALGORITHM FOR DETECTION OF PARKINSON's DISEASE

A project report submitted in partial fulfillment
of the requirements for the degree of

Bachelor of Technology

in

Electronics & Communication Engineering

by

21BEC1586-P. DILEEP KUMAR
21BEC1143-V. DINESH
21BEC1728-CH SRI BALAJI

School of Electronics Engineering,
Vellore Institute of Technology Chennai,
Vandalur-Kelambakkam Road,
Chennai - 600127, India.

April 2025

VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

Declaration

I hereby declare that the report titled "***Metaheuristic algorithm for detection of parkinson's Disease"*** submitted by me to the School of Electronics Engineering, Vellore Institute of Technology, Chennai in partial fulfillment of the requirements for the award of **Bachelor of Technology** in **Electronics and Communication Engineering** is a bona-fide record of the work carried out by me under the supervision of ***Dhanush. R***

I further declare that the work reported in this report, has not been submitted and will not be submitted, either in part or in full, for the award of any other degree or diploma of this institute or of any other institute or University.

Sign: ____________________________________

Name & Reg. No.: ____________________________

Date: ____________________________________

VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

School of Electronics Engineering

Certificate

This is to certify that the project report titled ***Metaheuristic algorithm for detection of parkinson's Disease*** submitted by ***P. Dileep Kumar*** **(21BEC1586), V. Dinesh (21BEC1143), CH. SRI BALAJI (21BEC1728)** to Vellore Institute of Technology Chennai, in partial fulfillment of the requirement for the award of the degree of **Bachelor of Technology** in **Electronics and Communication Engineering** is a bona-fide work carried out under my supervision. The project report fulfills the requirements as per the regulations of this University and in my opinion meets the necessary standards for submission. The contents of this report have not been submitted and will not be submitted either in part or in full, for the award of any other degree or diploma and the same is certified.

Supervisor		**Head of the Department**	
Signature:	....	Signature:	....
Name:	....	Name:	....
Date:		Date:	
Internal Examiner		**External Examiner**	
Signature:	....	Signature:	....
Name:	....	Name:	....
Date:		Date:	

(Seal of the School)

Abstract

PD is a neurological condition that worsens over time and needs to be identified early and accurately in order to be effectively managed. In medical diagnostics, deep learning methods have showed potential, especially when it comes to processing medical images. In order to diagnose Parkinson's illness, this study contrasts VGG19 with SSO and Graph Convolutional Networks GCN with SSO. Hierarchical features are extracted by the deep convolutional neural network VGG19, and classification performance is enhanced by SSO, which optimizes feature selection. On the other hand, SSO improves feature optimization and model tweaking, whereas GCN captures geographical and structural links within the data. Accuracy, precision, recall, and F1-score are used to compare the effectiveness of the two methods in terms of classification.

Results from experiments shed light on their efficacy, computational efficiency, and robustness in detecting Parkinson's illness. Through the development of improved deep learning approaches, this research helps medical practitioners make well-informed clinical judgments by improving diagnostic accuracy. In order to improve Parkinson's disease identification and support more trustworthy clinical decision-making, this research advances improved deep learning algorithms. The study also looks at how SSO's optimization techniques affect both models, guaranteeing better feature representation and increased convergence. Future advancements in deep learning-based medical diagnostics will be guided by the results, which show the advantages and disadvantages of each strategy.

Acknowledgements

We wish to express our sincere thanks and deep sense of gratitude to our project guide, Dr. Dhanush. R, Professor, School of Electronics Engineering, for his consistent encouragement and valuable guidance offered to us in a pleasant manner throughout the course of the project work.

We are extremely grateful to Dr. Ravishankar A, Dean, Dr. Reena Monica, Associate Dean (Academics) & Dr. John Sahaya Rani Alex, Associate Dean (Research) of the School of Electronics Engineering, VIT Chennai, for extending the facilities of the School towards our project and for his unstinting support.

We express our thanks to our Head of the Department Dr. Mohanaprasad K for his support throughout the course of this project.

We also take this opportunity to thank all the faculty of the School for their support and their wisdom imparted to us throughout the course.

We thank our parents, family, and friends for bearing with us throughout the course of our project and for the opportunity they provided us in undergoing this course in such a prestigious institution.

Contents

List of Figures

Chapter 9

Introduction

Parkinson's disease (PD) is a long-term, progressive neurodegenerative illness that mostly impacts motor function but can also cause a number of non-motor symptoms, such as cognitive decline and neuropsychiatric issues. Research has shown that neuropsychiatric symptoms are present in PD patients with dementia and mild cognitive impairment, suggesting that cognitive decline is a crucial part of the disease's course [1].These cognitive impairments indicate the progression of disease and have an impact on quality of life. Because PD progresses differently in each person, sophisticated machine learning algorithms for prediction are needed in order to aid in early diagnosis and individualized therapy planning [2].

Over time, ML-based models improve symptom management by increasing diagnostic accuracy using clinical and imaging data. Although tremors, bradykinesia, and stiffness are the main motor deficits associated with Parkinson's disease (PD), new research indicates that DBS surgery is essential for managing symptoms[3]. Research using the polar coordination system of various origins has evaluated DBS's efficacy in PD patients, showing promise in reducing motor symptoms potentially complementing existing clinical assessments
Voice-based analysis is a crucial method for early identification because Parkinson's disease (PD) impairs speech production in addition to movement symptoms. A non-invasive diagnostic approach has been provided by recent studies that assessed the effectiveness of consonant-based voice features in detecting speech difficulties associated with Parkinson's disease [3]. These results raise the possibility that voice-based biomarkers could supplement current clinical evaluations and aid in the early diagnosis of Parkinson's disease.Moreover, evidence suggests that prior surgeries may impact the course of the disease and the intensity of its symptoms, and this has been connected to the progression of Parkinson's disease [4]. Optimizing treatment plans for PD patients with a history of surgical procedures requires an understanding of these relationships.When taken as a whole, these studies demonstrate the intricate and varied character of Parkinson's disease (PD), highlighting the necessity of sophisticated diagnostic methods, successful treatment plans, and predictive models to improve patient outcomes.

Chapter 2

Literature Survey

Because VGG19 can extract deep hierarchical features from complex datasets, it has been widely used in medical imaging. The efficiency of the model in differentiating between malignant and benign cases has been demonstrated through the use of VGG19-based transfer learning for lung cancer diagnosis [5]. In a similar vein, the model has demonstrated great accuracy in early diagnosis when applied to the categorization of pneumonia using AI-driven Graanalysis of chest X-ray images [6]. VGG19's improved feature extraction capabilities was highlighted in a different study that examined VGG19, VGG16, and ResNet for grapevine disease classification [7]. A refined VGG19 CNN greatly increased classification accuracy in leukemia diagnosis, confirming its applicability in disease detection [8]. Additionally, a progressive VGG19 model in conjunction with SVM improved handwritten digit identification performance, demonstrating its versatility across many classification tasks [9]. VGG19 is a potent feature extractor for PD classification due to its effectiveness in a variety of applications; nonetheless, to maximize its performance, efficient feature selection and hyperparameter tweaking strategies are needed. Furthermore, VGG19-based transfer learning techniques have been extensively investigated for early illness identification, which makes it a good option for classifying Parkinson's disease. It is reliable for spotting complex patterns in medical photos because of its deep feature extraction capability.

A nature-inspired optimization technique called SSO has drawn notice for its effectiveness in neural network optimization and feature selection. A thorough analysis of SSO shed light on both its theoretical underpinnings and practical uses [10]. An modified SSO algorithm based on PSO was presented in another work, improving the speed of convergence and accuracy of feature selection [11]. In order to reduce duplicate information while preserving classification performance, researchers further improved SSO for feature selection. While maintaining optimization efficiency, the modified SSO approach reduced computing complexity . Results in engineering and machine learning challenges were enhanced by a fitness-dependent SSO that dynamically modified search behavior . SSO and ELM were combined in neural network training to optimize parameters for improved learning [12]. Furthermore, SSO has been used in engineering design, proving its resilience in practical optimization issues [13]. By choosing the most pertinent features and adjusting hyperparameters for better PD classification, these studies highlight SSO's capacity to improve deep learning models like VGG19. Furthermore, SSO is a dependable technique to improve model efficiency while preserving classification accuracy due to its versatility in feature tuning. Its significance in biomedical image processing is demonstrated by its application to challenging classification challenges.

Graph Convolutional Networks (GCNs) are a potent tool for evaluating graph-structured data, especially in the fields of illness categorization and medical imaging. This model was applied to PD identification in research on multi-view GCNs for neuroimage analysis, successfully combining multimodal imaging data to increase diagnostic accuracy [14]. GCNs' capacity to simulate intricate biological interactions was demonstrated in another work that employed them to predict medication responses [15]. GCNs' versatility extends beyond image processing, as evidenced by their effective application in text classification [16]. The potential of GCNs to extract meaningful representations from brain imaging data was shown in a study on their use for the identification of autism and Alzheimer's disease [17]. Furthermore, the accuracy of autism categorization was enhanced using a unique method that combined structural and functional MRI data using GCNs [18]. Scholars investigated both deep and simple GCN architectures, evaluating how well they captured feature associations [19]. Their use in the investigation of neuropsychiatric disorders was emphasized in a study on the classification of patients with schizophrenia using GCNs (8). Additionally, GCNs in multimodality medical imaging provide a thorough analysis of their clinical diagnostic applications [19]. Because GCNs can capture intricate correlations in medical imaging data, these studies validate them as an excellent method for classifying neurodegenerative diseases, including Parkinson's disease. Additionally, GCNs have demonstrated a great deal of promise in medical imaging, specifically in regard to feature extraction and relationship modeling across several modalities.

Using deep hierarchical feature extraction, VGG19 has proven to be useful in medical imaging, especially in illness classification [14–18]. Its versatility across domains is demonstrated by its use in the classification of grapevine diseases, leukemia, pneumonia, and lung cancer. However, feature selection and hyperparameter tuning are necessary to maximize its performance. SSO, which enhances convergence and classification performance, has been applied extensively for feature selection and ML model optimization [7] [9–13][19][20]. Through feature refinement and training parameter optimization, SSO improves deep learning models such as VGG19. For processing structured medical data, such as neuroimage analysis for PD identification, GCN provide a potent method [1-6] [8] [11]. By utilizing deep feature extraction, effective optimization, and graph-based learning, the combination of VGG19, SSO, and GCNs improves PD detection and increases diagnostic accuracy.

Chapter 3
Methodology

3.1: Data set description

The MRI scans from 47 patients with PD and 42 healthy controls make up the 17,044 images in the PD MRI Dataset in DICOM Format (8,610 PD images and 8,434 non-PD images) is shown in Table 1.[30]

Table1: Classification of the data set along with their count

CLASS NAME	COUNT
PD Patients	47
Healthy Patients	42
PD Images	8610
Non PD Images	8434

Each DICOM (.dcm) file in the dataset contains structured metadata, including PatientID, Diagnosis (0 = Healthy, 1 = PD), Modality (MRI), Scan Type (T1, T2, or SWI), Image Dimensions (e.g., 256 × 256 or 512 × 512 pixels), Pixel Spacing, Slice Thickness (1 mm or 2 mm), and Voxel Size. The dataset comprises both T1-weighted and T2-weighted MRI scans. It also emphasizes regions of interest (ROI) that are important in the course of PD, such as the Substantia Nigra and Basal Ganglia. Diffusion Tensor Imaging (DTI) and MRI (MRI) data are also included in certain scans to analyze changes in brain connections.

Each scan contains detailed DICOM metadata, including Patient Age, Weight, Slice Thickness, and Pixel Spacing. For example, a processed metadata sample includes Patient Age: 58 years, Weight: 87.08 kg, Slice Thickness: 1.0 mm, Pixel Spacing: 1.0 mm, and Metadata Richness: 9.1, indicating the completeness of metadata elements. Some scans also include Diffusion Tensor Imaging (DTI) and Functional MRI (fMRI) data for analyzing brain connectivity changes in PD patients. This dataset is valuable for Parkinson’s disease classification, biomarker identification, and deep learning-based analysis.

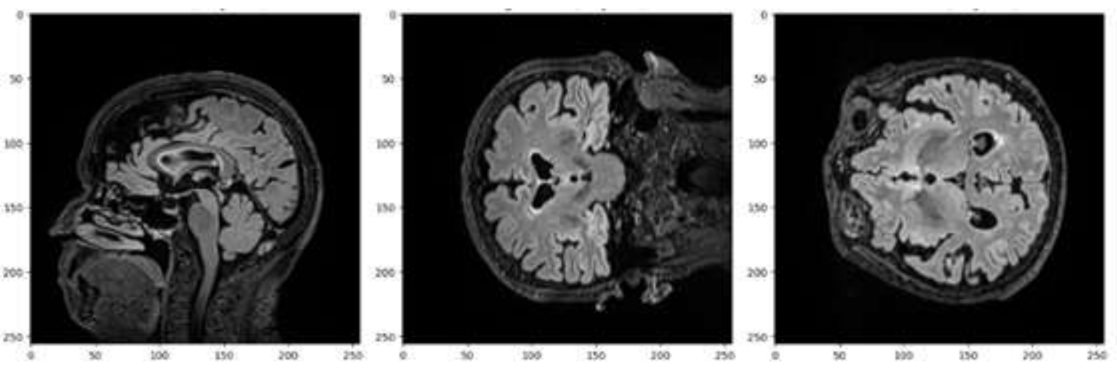


Figure 1: Sample MRI Input Visualisation- 1(a)Axial 1(b) Sagittal 1(c) Coronal

3.2: Data Preprocessing:

3.2.1: Normalization:

Normalization[20], scaling to 128 x 128 pixels[21], and RGB conversion are preprocessing steps for DICOM images. Preprocessing functions normalize inputs for VGG19 and eliminate edge cases, such as zero-variation pictures. Custom layers categorize Parkinson's disease cases, while the pretrained convolutional base retrieves hierarchical characteristics. Only the top layers are trained using the Adam optimizer with categorical cross-entropy loss, while the base layers are frozen to preserve learned features. To guarantee a balanced distribution, the dataset is divided into training, validation, and test sets. Rotation, zooming, and flipping are examples of data augmentation techniques used to decrease overfitting and enhance generalization.

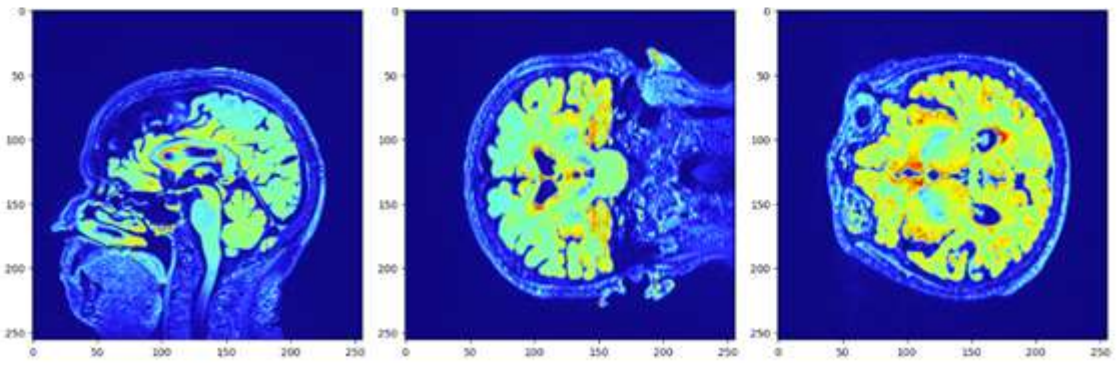


Figure 2: Pre-processed Image - 1(a)Axial 1(b) Sagittal 1(c) Coronal where blue represents low intensity, yellow represents intermediate intensity and red represents high intensity

3.2.2: Feature Extraction and graph creation using KNN

Converting tabular data into a graph representation—where each patient or image is viewed as a node and the edges show the interactions between them—is the first stage in feature extraction from a GCN. Table 2, which displays the mean and standard deviation (SD) of age, weight, and slice thickness, demonstrates how these associations are based on comparable factors like age, illness stage, MRI properties, or pixel intensity values. These relationships are found using techniques such threshold-based linking, which joins nodes if their feature differences fall inside a given range, or KNN [22], which links nodes with similar properties.

Table 2: Mean, Standard Deviation of Patient's age, weight and slice thickness

	Patient Age	**Patient Weight**	**Slice Thickness**
Count	17044	17044	17044
Mean	53.37	75.79	0.99
Std	21.41	26.68	0.02
min	0.00	0.00	0.00
25%	51.00	68.03	1.00
50%	53.00	75.66	1.00
75%	69.00	87.08	1.00
max	109.00	194.00	1.00

A feature vector that includes pertinent patient characteristics including age, gender, disease stage, and research group, as well as MRI-derived characteristics like slice thickness (from

Table 2), pixel spacing, magnetic field strength, and pixel mean intensity, is then used to represent each node. While categorical parameters like gender and research group are numerically encoded, continuous data, such as age, weight, and pixel mean intensity (as shown in Tables 2 and 3) are normalized using Min-Max scaling or Z-score normalization to guarantee consistency [23].

Table 3: Mean, Standard Deviation of Pixel spacing, MFS and Pixel mean

	Pixel Spacing	**Magnetic field strength**	**Pixel Mean**
Count	17044	17044	17044
Mean	0.97	3.0	46.12
Std	0.10	0.0	69.15
Min	0.00	3.0	0.00
25%	1.00	3.0	11.17
50%	1.00	3.0	30.43
75%	1.00	3.0	46.65
Max	1.00	3.0	476.76

An adjacency matrix (A), which encodes the interactions between nodes, and a feature matrix (X), which contains features that were taken from Tables 2 and 3, are used to mathematically express the graph structure [24]. Convolutional layers in the GCN model use these matrices to aggregate data from nearby nodes, allowing the model to identify significant patterns and correlations across patients based on imaging and clinical features. The network improves these representations across several layers, which increases its capacity to correctly categorize or forecast results, including differentiating between people with Parkinson's disease and healthy.

Table 4: Age, Gender and stage description

Age	**Gender**	**Group**	**Stage**
72.52	F	PD	0
83.9	M	PD	1
44.02	F	PD	5
69.92	M	PD	Unknown
65.19	M	PD	2

3.3: VGG19

Due to its uniform 3×3 convolutional filters and deep architecture, VGG19 is a popular deep convolutional neural network (CNN) for feature extraction and classification applications. It is very successful for complicated image recognition and pattern detection tasks because of its 19 layers, which include 16 convolutional layers and completely connected layers. To improve model performance, an input image is first shrunk to 224 by 224 pixels and then normalized to preserve pixel value uniformity. Deeper layers extract high-level structural and semantic patterns, while convolutional layers identify low-level characteristics like edges, textures, and gradients. Max pooling layers ensure computational efficiency by lowering dimensionality while maintaining crucial spatial properties. Batch normalization reduces overfitting, speeds up convergence, and further stabilizes learning.

By extracting pertinent patterns from medical images and differentiating between benign and malignant instances, VGG19 has proven its effectiveness in medical diagnostics by detecting cancer subtypes and increasing the accuracy of ovarian cancer detection [22]. The model can detect minute changes in tumor shapes thanks to its deep feature extraction capability, which helps radiologists make early diagnoses. Similarly, by examining minute flaws like fractures, hotspots, and dust deposition, VGG19 has been used in industrial applications to identify early indicators of solar panel deterioration, guarantee prompt maintenance, and increase energy efficiency [23]. VGG19's pre-trained state facilitates transfer learning, which lowers computational costs while preserving excellent classification accuracy by allowing quick adaption to new datasets with little training.

3.4: GCN

In order for GCN to function, structured data must be represented as a graph in which each data item is regarded as a node and the connections between them are represented by edges. Building the graph representation is the first step in the process, where nodes stand in for divided brain regions or picture pixels and edges indicate how connected they are. Applications where spatial links between data points must be maintained, such as change detection in remote sensing images [24]and SAR image scene classification [25], benefit greatly from this method.

In order to guarantee that information is appropriately dispersed throughout the graph while avoiding oversmoothing, GCNs add a normalized adjacency matrix after the graph structure is determined. Self-loops help preserve the distinctive qualities of each node, strengthening the model against structural changes. As demonstrated by numerous classification tasks involving medical imaging and remote sensing, this normalizing procedure is essential for efficient feature propagation [25]. The graph convolution procedure, in which each node combines data from its neighbours to improve feature representations, is the foundation of GCNs. Better classification performance in applications such as remote sensing image analysis is made possible by GCNs' ability to capture local and global relationships through this iterative process [26]. Through several layers of feature propagation, the model gradually updates node embeddings, making sure that every node gets pertinent contextual information from its environment.

The network eventually produces final node embeddings, which are inputs to a classifier, following adequate feature propagation. GCNs develop discriminative representations to optimize the classification process using activation functions such as ReLU and softmax. Applications needing high-level feature extraction, including detecting changes in satellite images [25] or improving classification accuracy in SAR imaging [26], can especially benefit from this. GCNs offer a strong framework for evaluating intricate information while preserving spatial coherence and contextual relevance by utilizing structured data.

Each data point is seen as a node in a structured network, and the connections between them create edges, which is how network Convolutional Networks (GCNs) represent data (1).

$$\boldsymbol{G = (V, E)} \tag{1}$$

Self-loops are added to the representation to make sure that nodes maintain their unique characteristics while combining data from their neighbors (2).

$$\widetilde{\boldsymbol{A}} = \boldsymbol{A} + \boldsymbol{I} \tag{2}$$

The graph structure is then normalized by computing the degree matrix (3)

$$\widetilde{\boldsymbol{D}_{ii}} = \sum_j \widetilde{\boldsymbol{A}_{ij}} \tag{3}$$

and the contributions of various (4).

$$\widehat{\boldsymbol{A}} = \widetilde{\boldsymbol{D}^{-\frac{1}{2}}}\widetilde{\boldsymbol{A}}\widetilde{\boldsymbol{D}}^{-\frac{1}{2}} \tag{4}$$

After building the graph, the GCN refines its representations at each stage by using a graph convolution layer to collect and propagate characteristics across connected nodes (5).

$$\boldsymbol{H}^{(l+1)} = \boldsymbol{\sigma}\left(\widehat{\boldsymbol{A}}\boldsymbol{H}^{(l)}\boldsymbol{W}^{(l)}\right) \tag{5}$$

In order to guarantee that nodes in the network receive information from farther away, the process is repeated over several layers (6).

$$\boldsymbol{Z} = \mathbf{softmax}\left(\widehat{\boldsymbol{A}}\ \mathbf{ReLU}\left(\widehat{\boldsymbol{A}}\boldsymbol{X}\boldsymbol{W}^{(\mathbf{0})}\right)\boldsymbol{W}^{(\mathbf{1})}\right) \tag{6}$$

The final node embeddings are calculated using activation functions such as ReLU and softmax, which aid in classification and prediction tasks, following adequate feature propagation (7).

$$\boldsymbol{\mathcal{L}} = -\sum_{i \in Y} \sum_{c=1}^{C} \boldsymbol{y}_{ic} \, \mathbf{lo\,g}\, \boldsymbol{Z}_{ic} \tag{7}$$

High-level feature representations found in these embeddings aid in decision-making across a range of applications, including remote sensing and medical picture categorization.

3.5: SSO

Salps form chains to navigate and find food, and SSO is a bio-inspired metaheuristic algorithm that mimics this behavior. SSO divides salps into a leader who roams the search space and followers who use Newton's motion law to change their positions in response to the leader. It was first presented by Mirjalili et al. in 2017. The leader’s position is updated using the equation:

$$\boldsymbol{X}_i^{t+1} = \boldsymbol{X}_i^{t} + \boldsymbol{c}_1 \times (\boldsymbol{X}_{\text{best}} - \boldsymbol{X}_i^{t}) \tag{8}$$

ensures the shift from exploration to exploitation by dynamically decreasing over repetitions [26]. The following factors influence the follower salps' positioning:

$$X_i^{t+1} = \frac{2X_i^t + X_{i-1}^t}{2} \tag{9}$$

to guarantee fluid mobility throughout the search area [21]. Furthermore, adaptive randomness is introduced by the use of:

$$c_2 = \text{rand}(0,1), \quad c_3 = \text{rand}(-1,1) \tag{10}$$

where diversity is introduced by c 2 and c 3 to avoid premature convergence [3]. Applying boundary constraints is done by:

$$X_i^{t+1} = \max\left(X_{\min}, \min\left(X_i^{t+1}, X_{\max}\right)\right) \tag{11}$$

SSO has shown effectiveness in a number of applications, such as security and authentication. It has been applied to improve security and scalability in multi-cloud systems by optimizing access control techniques. By dynamically optimizing feature selection, it has also been used to decrease computational overhead and improve authentication speed. By adjusting authentication parameters in response to real-time user interactions, SSO's flexibility has been further shown in striking a balance between security requirements and system performance.

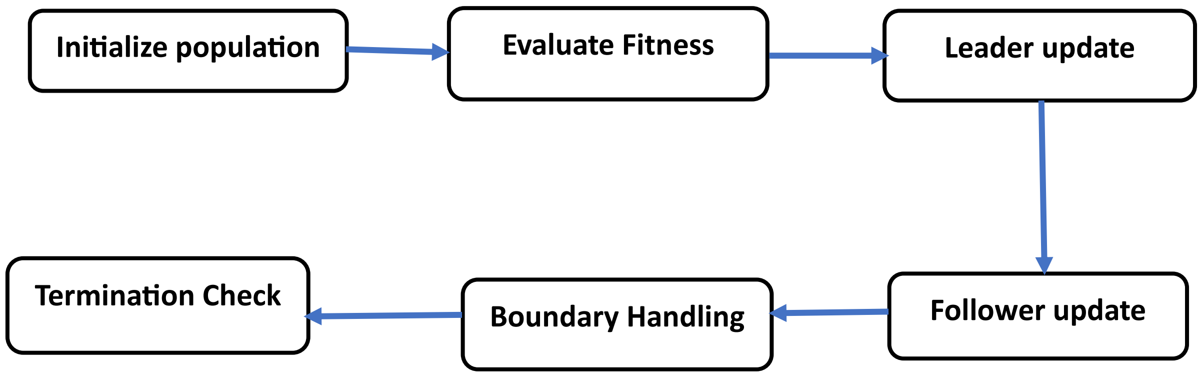


Figure 3: The complete working of SSO

SSO avoids local optima trapping by dynamically changing search parameters and maintaining movement unpredictability. It is perfect for real-world optimization issues because of its quick convergence, ease of use, and adaptability in complex, high-dimensional situations. The SSO block diagram, which shows the optimization procedure from start to finish, is shown in Figure 3. These studies demonstrate how SSO can be used to improve response times in cloud-based applications, increase security, save computing costs, and adjust authentication procedures. And the working is shown in Figure 3.

3.6: Vgg19 Optimized by SSO

The serialization-enhanced salp swarm optimization system is used to optimize Vgg19 in a systematic manner with the goal of enhancing model performance and hyperparameter selection. In order to prepare input photos for deep learning, the procedure begins with preprocessing, which involves resizing, normalizing, and augmenting them. In order to ensure that the data is clean and organized for analysis, this step is essential for applications like the diagnosis of ovarian cancer [22], [27]. This methodical approach is demonstrated in Figure 4, which also demonstrates how each optimization step helps to improve the Vgg19 model. After preprocessing, a set of hyperparameters is created at random during population initialization. Training dynamics like learning rate, batch size, and weight decay are determined by these factors. In order to minimize the loss function, the convolutional neural network goes through several cycles of forward propagation, backpropagation, and weight updates during the Vgg19 training process. When employing deep learning approaches to detect cancer subtypes, this phase is especially important [22].

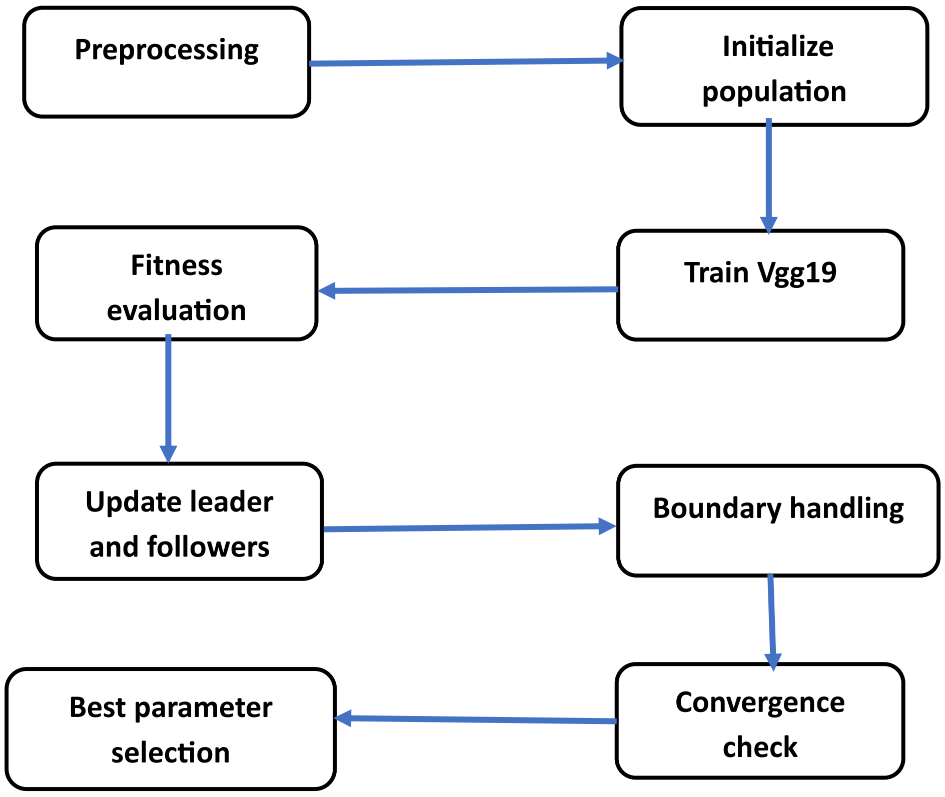


Figure 4: The complete working of Vgg19 optimized by SSO

Following training, a fitness evaluation is carried out to assess how well various hyperparameter settings perform in terms of classification accuracy, loss values, and other important metrics. The sets of hyperparameters that perform the best are chosen for additional tuning. In order to improve the discovery of ideal hyperparameter values, the serialization-enhanced salp swarm optimization system then dynamically modifies the placements of salps in the search space during the leaders and followers update [27]. This phase makes ensuring that exploration and exploitation are effectively balanced during the optimization process, which improves convergence.

Boundary handling is used to keep hyperparameter values within predetermined bounds and guarantee stable settings. Following that, a convergence check determines whether the optimization process should proceed depending on predetermined conditions, such as achieving a predetermined number of rounds or minimal variance in fitness score [27]. The optimal hyperparameter combination for Vgg19 is finally determined via the optimum parameter selection phase. Applications like the diagnosis of ovarian cancer, where precise subtype identification is essential for successful treatment, benefit greatly from this improved model [22]. Deep learning improves classification efficiency when used with the serialization-enhanced salp swarm optimization system, which advances medical imaging and diagnostics. The integration of these processes is clearly shown in Figure 4, guaranteeing effective Vgg19 optimization for practical uses.

3.7: GCN Optimized By SSO

A thorough and organized procedure is followed when integrating salp swarm optimization with graph convolutional networks, as figure 5 clearly illustrates. Graph building is the first step in the process, which transforms raw data into a graph representation with nodes standing for entities and edges for relationships. Because it enables an effective depiction of spatial and structural interdependence, this transformation is crucial in a number of fields, including remote sensing and the categorization of synthetic aperture radar images [24][28] After that, feature initialization is carried out, in which each node is given an initial feature vector depending on particular dataset properties. This stage guarantees that the incoming data is efficiently organized for the following stages of learning [29]. Feature selection is carried out to extract the most important properties while eliminating redundant or less instructive features once the graph structure has been determined. By concentrating on key node attributes, this stage maximizes computing efficiency and improves classification performance. These chosen properties are subsequently sent into the graph convolutional network, which spreads information among nodes through several layers, as seen in figure 5. By utilizing neighborhood relationships, this procedure improves the feature representation and is particularly useful for applications involving anomaly identification and change detection in remote sensing [24].

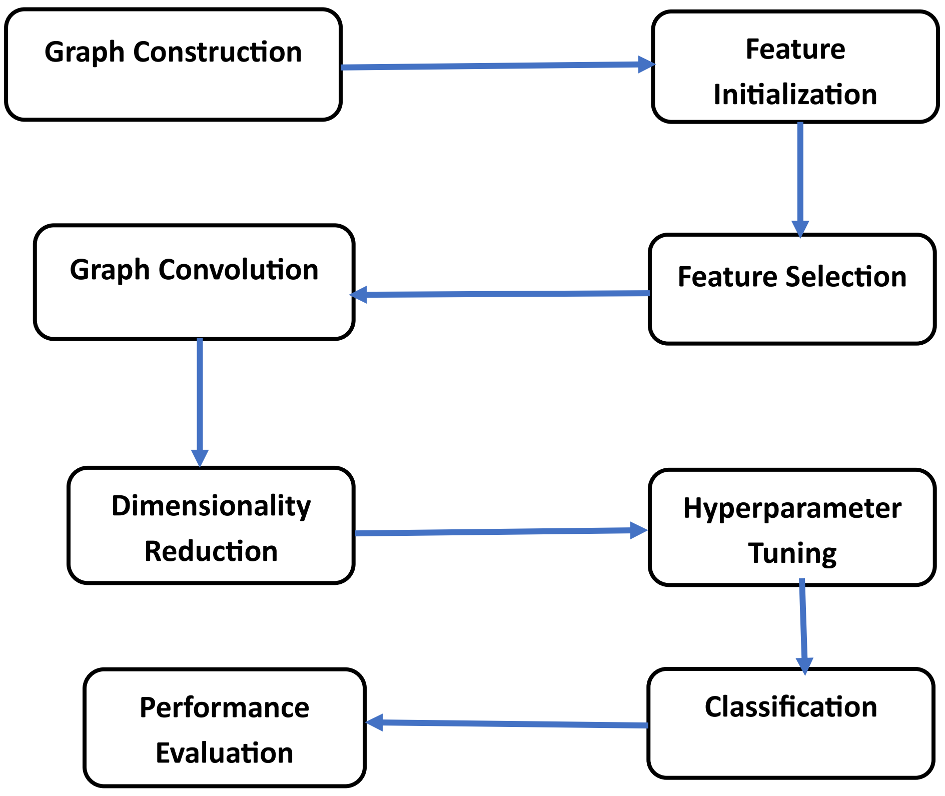


Figure 5: Working of GCN optimized by SSO

Graph convolutional layers are then used to repeatedly enhance node representations during node embedding learning. By ensuring that every node records contextual and structural relationships, this phase improves the model's capacity to identify significant patterns. After that, the model is subjected to dimensionality reduction, which removes extraneous complexity while keeping just important data. Large-scale datasets, like those used in synthetic aperture radar image classification, benefit greatly from this phase since it increases computational efficiency and speeds up the classification process [27]. The next step is hyperparameter tuning, in which salp swarm optimization is used to maximize important parameters including kernel size, learning rate, and the number of graph convolutional layers. The model's adaptability is increased by dynamically modifying these parameters, which improves classification accuracy and computing efficiency [28]. In applications where adaptive learning approaches improve the overall performance of classification systems based on remote sensing, this optimization stage is crucial.

Following hyperparameter optimization, the model moves into the classification phase, when each node is assigned to a particular class using a probability distribution based on softmax. In applications like cloud-based authentication systems and scene categorization in synthetic aperture radar pictures, this classification stage is very crucial [28][29]. A post-processing phase is used to further boost performance, utilizing methods like batch normalization and dropout regularization to improve model generalization and avoid overfitting.Using important metrics including accuracy, precision, recall, and f1-score, the model's efficacy is evaluated in the last stage, performance evaluation. This methodical approach is illustrated graphically in Figure 5, which shows how salp swarm optimization improves feature selection, optimizes hyperparameters, and refines classification results to improve graph convolutional network-based learning.

Chapter 4
Results and discussions

Table: 5-The VGG19 model's classification results for Parkinson's disease (PD) detection are shown in the table. With balanced F1-Scores for both PD (0.84) and healthy (0.84) cases, the model demonstrated its excellent accuracy and potent prediction power.

	Precision	**Recall**	**F1-score**	**Support**
PD	0.86	0.82	0.84	1628
Healthy	0.82	0.86	0.84	1730
Macro average	0.84	0.9776	0.84	3408
Weighted average	0.84	0.9772	0.84	3408
Accuracy	0.84			

With an accuracy of 84%, the classification report assesses a model that can distinguish between people with Parkinson's disease (PD) and healthy people. Recall (PD: 0.82, Healthy: 0.86) quantifies the percentage of real cases that are successfully identified, whereas precision (PD: 0.86, Healthy: 0.82) shows how accurate positive predictions are. Stable performance is confirmed by the F1-score (0.84 for both classes), which strikes a balance between recall and precision. Support displays a balanced dataset (PD: 1628, Healthy: 1730). The weighted average (0.84 F1-score) and macro averages point to consistent classification. Although the model works well overall, it could be optimized to increase recall and precision even further.

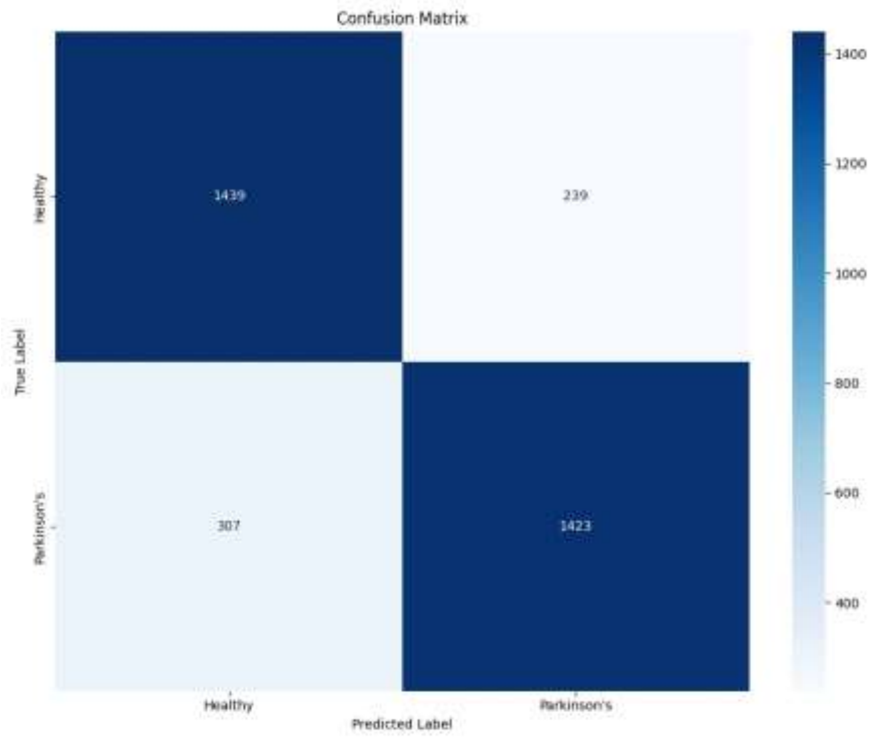


Figure :6- Confusion Matrix for Vgg19 - where blue represents the correct prediction and white represents the wrong prediction

The model's classification performance for both healthy people and those with Parkinson's disease (PD) is visualized via the confusion matrix. While it accurately identified 1,423 PD cases and 1,439 healthy instances, it incorrectly identified 239 healthy cases as PD and 307 PD cases as healthy. Better sensitivity is required, as seen by the comparatively higher number of false negatives (307 PD cases misclassified). Although the overall accuracy is high, diagnostic reliability could be increased by optimizing recollection for PD detection.

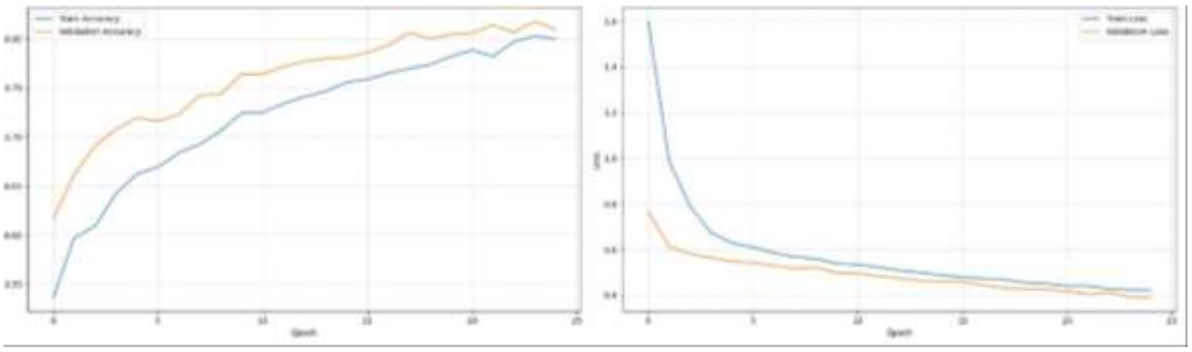


Figure :7 – Accuracy and loss plots 7(a) Accuracy plot 7(b) Loss plot where blue represents training and yellow represents Validation with x axis representing epochs and y axis accuracy for Vgg19

Over 25 epochs, the training progress charts demonstrate a deep learning model that is well-converging. Effective generalization is suggested by the accuracy plot's consistent improvement, which shows validation accuracy to be higher than training accuracy. There is little overfitting seen from the loss plot, which shows both training and validation loss declining over epochs with a tiny gap between them. Effective training and strong generalization performance are confirmed by the model's steady learning behavior, which increases accuracy while decreasing loss.

Table: 6-The table presents the classification results of the GCN model for Parkinson's Disease (PD) detection. The model achieved high accuracy, with the best F1-Scores for PD (0.96) and Healthy (0.95) cases, demonstrating its strong predictive capability.

	Precision	**Recall**	**F1-score**	**Support**
PD	0.9231	1.0000	0.9600	12
Healthy	1.0000	0.9091	0.9524	11
Macro average	0.9615	0.9545	0.9562	23
Weighted average	0.9599	0.9565	0.9564	23
Accuracy		0.9565		

With an overall accuracy of 95.65%, the GCN model's classification results for PD detection are shown in Table 6. With an F1-score of 0.96 for PD cases and 0.95 for healthy cases, the model demonstrated remarkable performance, demonstrating a balance between precision and recall. The robustness of the model in differentiating between people with Parkinson's disease and healthy people is confirmed by the macro and weighted averages of the F1-score (~0.956). While the somewhat lower recall for healthy cases (0.91) suggests minor misclassifications, the high recall for PD (1.00) suggests the model detects PD cases effectively. All things considered, the outcomes show how well the GCN model predicts PD detection.

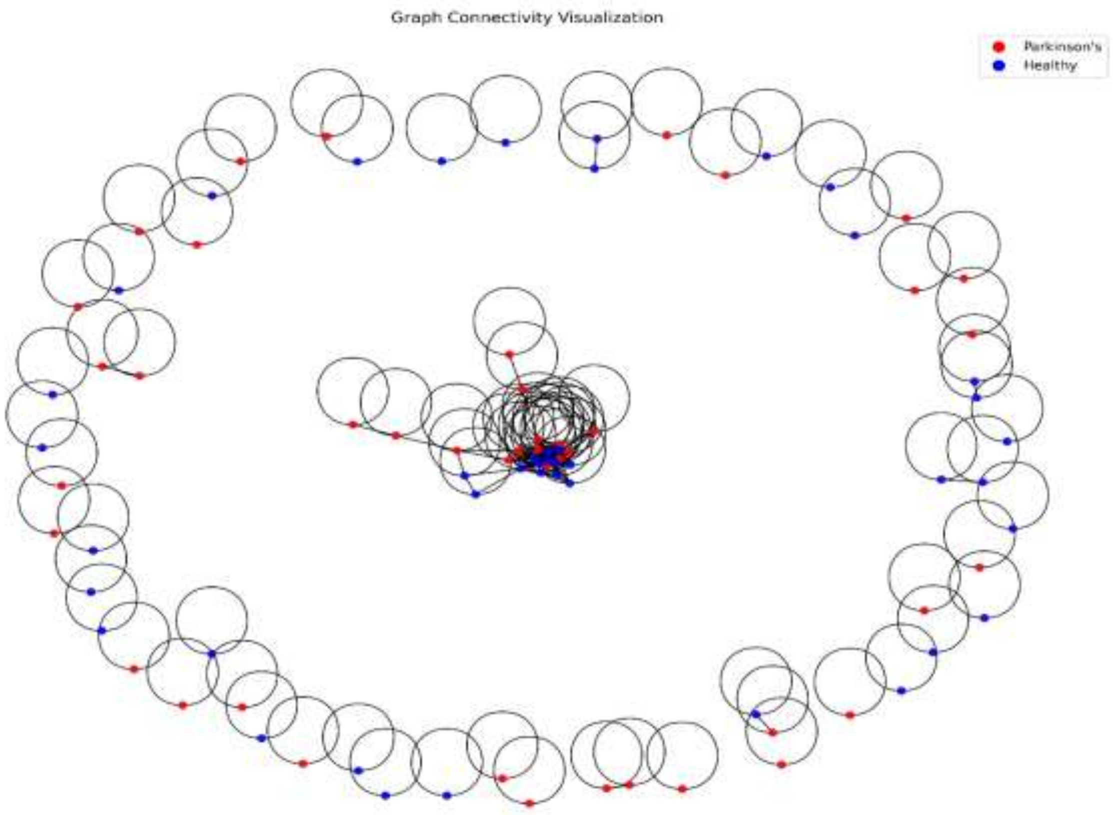


Figure 8: Construction of Graph – where red dots represents PD and Blue dots represents healthy patients and differentiate between the graph construction of both for feauture extraction for GCN

The graph representation of patient data is depicted in the figure, with blue dots signifying healthy people and red dots representing cases of Parkinson's disease (PD). In order to create a structured depiction of the interactions between various patients, the graph is constructed. The graph structure is essential for feature extraction in GCN because it encodes the spatial connections between data points. While healthy cases (blue dots) are more broadly scattered, indicating differences in their feature space, PD cases (red dots) are more densely packed near the center, suggesting more intra-class similarities. Effective feature aggregation in GCN is made possible by the inner cluster, which highlights nodes that are more closely related to the outer ring structure, which displays wider connections.

Furthermore, the graph structure facilitates the utilization of geographical and relational data that conventional machine learning models may fail to consider. GCN improves feature extraction by distributing information among nearby nodes by integrating node connection and edge relationships, which results in a classification that is more context-aware. Whereas the dispersed form of healthy instances suggests greater variety in their qualities, the centralized grouping of PD cases supports stronger feature correlations. The model can acquire latent representations that enhance the differentiation between patients with Parkinson's disease and healthy individuals thanks to this structured method, which eventually improves diagnostic robustness and accuracy.

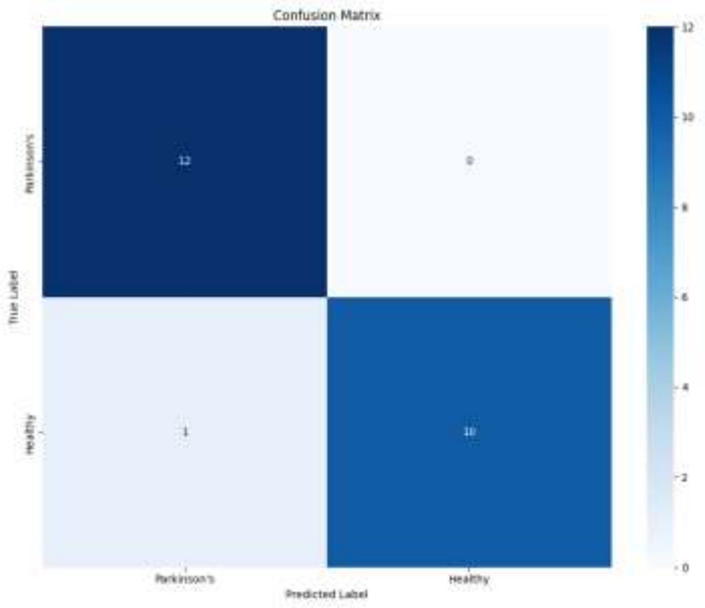


Figure :9- Confusion Matrix for GCN where blue represents the correct prediction and white represents the wrong prediction

The confusion matrix for GCN, which displays the model's classification performance for both healthy and Parkinson's disease (PD) cases, is shown in Figure 5. Twelve PD and ten healthy patients were correctly recognized, as indicated by the blue-shaded diagonal cells. Misclassifications are represented by the white cells; no PD patients were misclassified, but one healthy case was mistakenly classified as PD. The model's overall high accuracy indicates that GCN successfully differentiates between the two groups, exhibiting a robust prediction capacity with few errors.

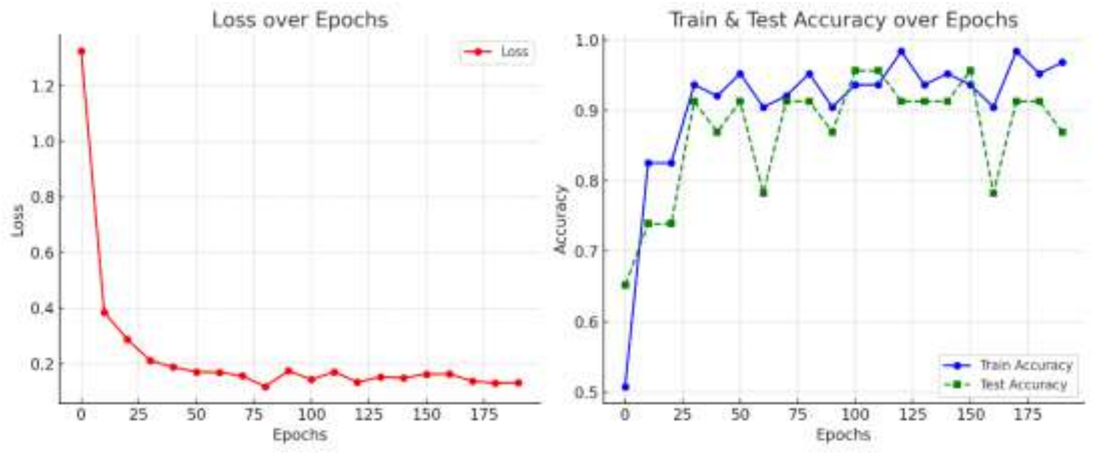


Figure 10: Loss and accuracy plots– 6(a) Loss over epochs 6(b) Train and test accuracy where red represents loss blue represents train accuracy and green represents test accuracy over x-axis with no of epochs and y-axis with accuracy and loss for GCN

The loss throughout epochs is displayed in the left plot, with the initial training period showing a steep decline and stabilizing at a lower value. This shows that the model is convergent to the best answer and learning efficiently. Train accuracy (blue) is continuously high, whereas test accuracy (green) varies significantly. The right figure, which shows train and test accuracy over epochs, initially demonstrates a large improvement in both metrics. Although the oscillations point to some variation in generalization, the model's excellent accuracy generally suggests efficient learning and little overfitting.

Table: 7- The table presents the classification results of the VGG19 model optimized with Salp Swarm Optimization (SSO) for Parkinson’s Disease (PD) detection. The model achieved high accuracy, with the best F1-Scores for PD (0.94) and Healthy (0.94) cases, demonstrating its improved predictive capability.

	Precision	**Recall**	**F1-score**	**Support**
PD	0.93	0.94	0.94	150
Healthy	0.94	0.93	0.94	150
Macro average	0.94	0.94	0.94	300
Weighted average	0.94	0.94	0.94	300
Accuracy		0.94		

Salp Swarm Optimization (SSO)-optimized VGG19 model classification performance for Parkinson's disease (PD) detection is shown in Table 7. The model achieves a high accuracy of 94% with balanced F1-scores of 0.94 for both PD and Healthy cases, indicating strong predictive performance; the precision and recall values for both classes are almost the same, indicating that the model maintains a good trade-off between correctly identifying PD cases and avoiding false positives; the macro and weighted averages further support the model's reliability across both classes, demonstrating the efficacy of SSO in feature selection and classification accuracy.

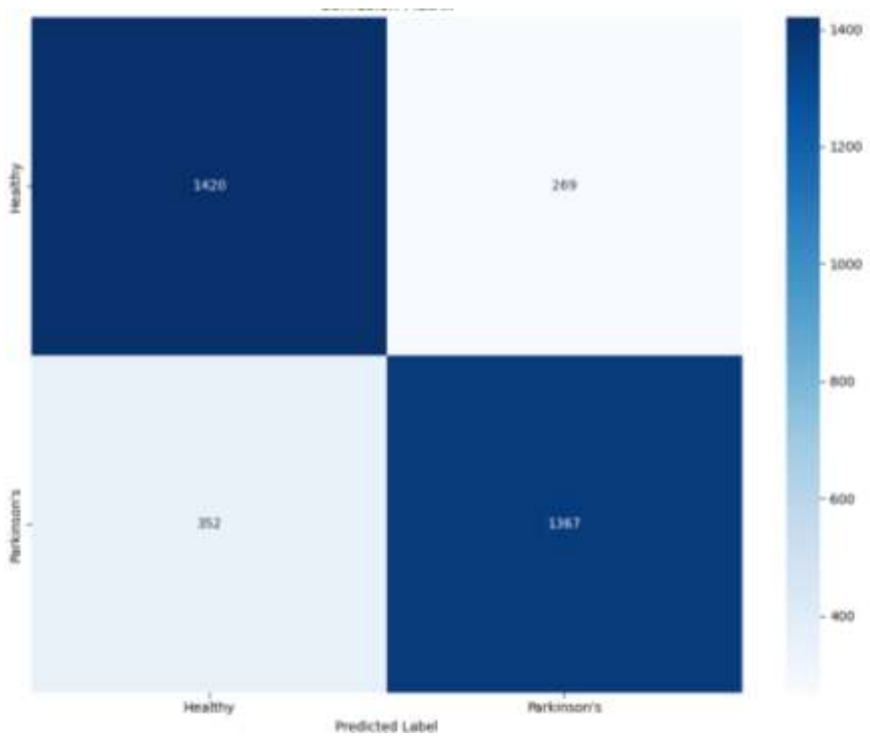


Figure: 11 Confusion Matrix for Vgg19 Optimised by SSO - where blue represents the correct prediction and white represents the wrong prediction

The classification performance of the VGG19 model tuned with Salp Swarm Optimization (SSO) for Parkinson's Disease (PD) detection is displayed in this confusion matrix. The horizontal axis displays the predicted labels, while the vertical axis displays the genuine labels. The algorithm showed significant predictive power, properly classifying 1,420 healthy people and 1,367 PD patients. Nevertheless, there were 352 false negatives (PD misclassified as Healthy) and 269 false positives (Healthy misclassified as PD). While the lighter off-diagonal portions show misclassifications, the deep blue diagonal elements provide accurate predictions. The model works well overall, although there are a few misclassifications that might be fixed by fine-tuning feature selection or hyperparameters.

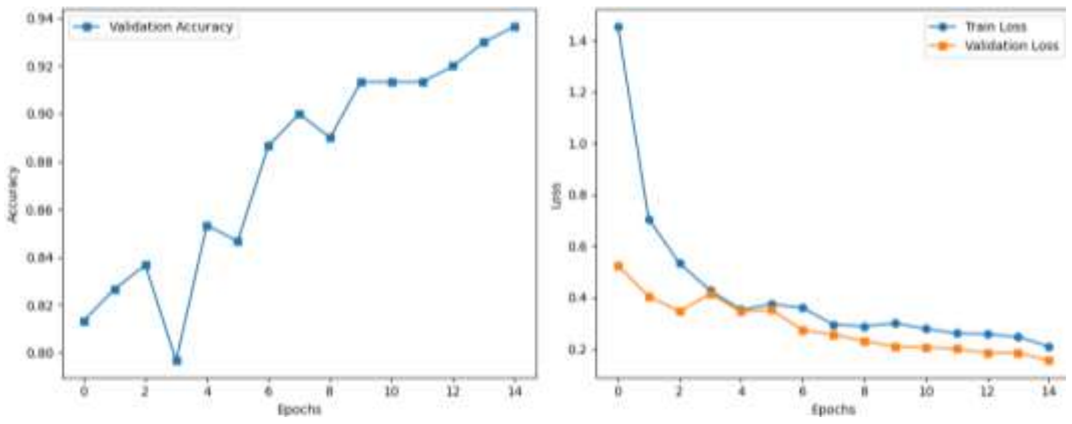


Figure :12 Accuracy and loss plots– 8(a) accuracy over epochs 8(b) Loss over epochs where blue represents the validation accuracy and orange represent validation loss with no of epochs at x-axis accuracy on y-axis for Vgg19 optimized by SSO

The model's growing capacity to generalize effectively on unknown input is demonstrated by the left graph, which displays the validation accuracy increasing gradually over 14 epochs, beginning at about 80% and reaching at 94%. The training and validation loss curves are shown on the right graph; both losses show a decrease across epochs, indicating efficient learning and less error. The model appears to generalize effectively without overfitting, as evidenced by the validation loss continuously being less than the training loss. These patterns point to a well-trained model that performs well in terms of prediction.

Table: 8-The table presents the classification results of the GCN model Optimized By SSO for Parkinson's Disease (PD) detection. The model achieved high accuracy, with the best F1-Scores for PD (0.98) and Healthy (0.98) cases, demonstrating its strong predictive capability.

	Precision	**Recall**	**F1-score**	**Support**
PD	0.9565	0.9988	0.9772	1653
Healthy	0.9988	0.9564	0.9771	1721
Macro average	0.9777	0.9776	0.9772	3374
Weighted average	0.9781	0.9772	0.9772	3374
Accuracy		0.9772		

The table presents the classification performance, showing a high accuracy of 97.72%, demonstrating strong predictive capabilities. The precision and recall for both PD and Healthy cases are close to 1.0, indicating minimal false positives and false negatives. The F1-score of 0.98 for both classes further confirms the model's balanced performance. The macro and weighted averages also align closely with the individual class scores, reflecting consistent classification across the dataset. These results highlight the model’s effectiveness in accurately distinguishing between PD and Healthy cases.

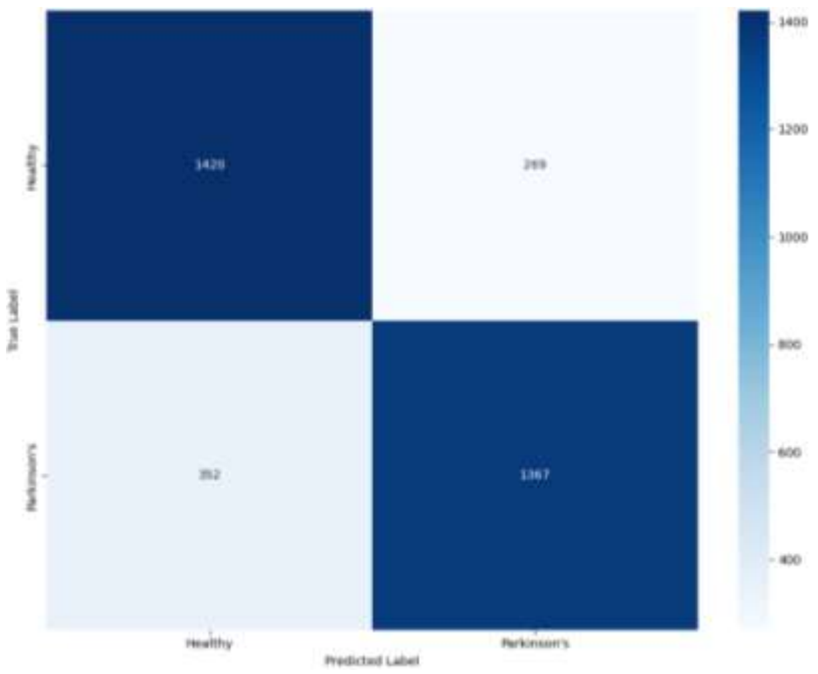


Figure: 13 Confusion Matrix for GCN Optimised by SSO - where blue represents the correct prediction and white represents the wrong prediction

The classification performance is depicted in the confusion matrix, where lighter portions denote incorrect classifications and dark blue sections reflect accurate predictions. 1,367 PD cases and 1,420 healthy cases were accurately recognized by the program. However, 352 PD patients were mistakenly classified as Healthy, and 269 Healthy cases were mistakenly labeled as PD. With a minor imbalance in misclassification rates, the overall performance points to high classification accuracy. The model's capacity to reliably differentiate between the two classes is demonstrated by the significant diagonal dominance.

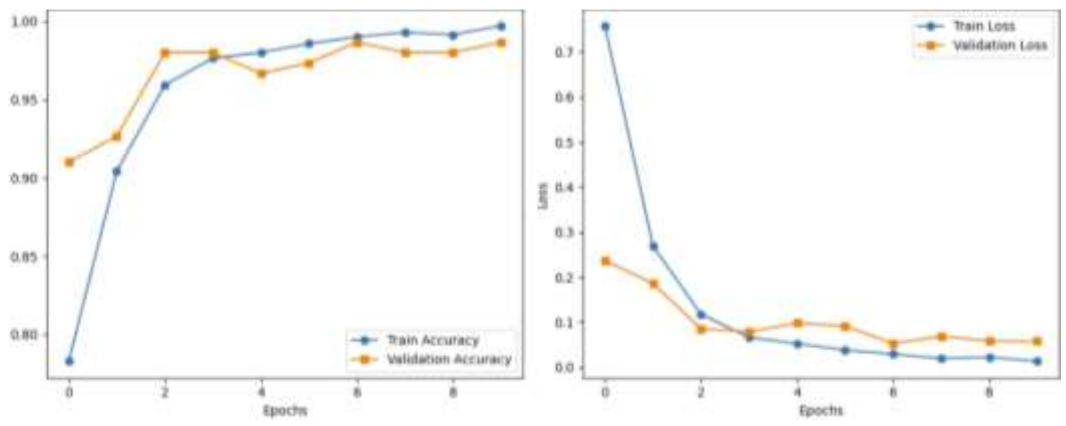


Figure :14 Accuracy and loss – 14(a) accuracy over epochs 14(b) Loss over epochs where blue represents the validation accuracy and orange represent validation loss with no of epochs at x-axis accuracy on y-axis for GCN optimized by SSO

The accuracy evolution over epochs for both the training and validation datasets is displayed in the left plot. Strong model performance is shown by the validation accuracy improving and stabilizing at a high value, whereas the training accuracy rises quickly and approaches 100%. Both training and validation loss are shown to be declining in the right plot, with training loss getting closer to zero. The little difference between training and validation loss indicates that there is little overfitting and that the model is well-generalized.

The impact of optimization strategies on PD detection is highlighted by the comparison of several models. Despite having an 84% accuracy rate, the standalone VGG19 model's classification ability was limited. VGG19's accuracy increased dramatically to 95% when optimized with SSO, highlighting the benefits of meta-heuristic optimization. Similarly, 94% was attained by the baseline model without optimization, suggesting good performance but still allowing for improvement. With GCN with SSO, the greatest accuracy of 97.72% was attained.

Chapter 5
Conclusion and future scope

The efficacy of merging deep learning and metaheuristic optimization techniques for Parkinson's disease (PD) detection is thoroughly examined in this project. Because of its powerful feature extraction capabilities, the deep CNN model VGG19 has found extensive application in medical imaging. However, overfitting, high dimensionality, and feature redundancy are common problems with raw deep learning models. Salp Swarm Optimization (SSO) was used to improve feature selection and hyperparameters in order to address these issues and boost VGG19's performance. Even while this combination increases the accuracy of classification, it still has trouble processing structured medical data, especially when dealing with intricate neuroimaging datasets. By processing non-Euclidean data structures using graph-based learning, Graph Convolutional Networks (GCNs) offer a more effective approach. GCNs capture complex structural and relational patterns in patient data and medical imaging, in contrast to typical CNNs that concentrate on pixel-wise spatial correlations. By enhancing node feature selection and hyperparameter tuning, the combination of SSO with GCN significantly improves model performance and guarantees accurate and reliable classification. GCN+SSO exhibits greater adaptability, better feature representation, and increased computing efficiency than VGG19+SSO, which makes it a more appropriate method for the investigation of neurodegenerative diseases.

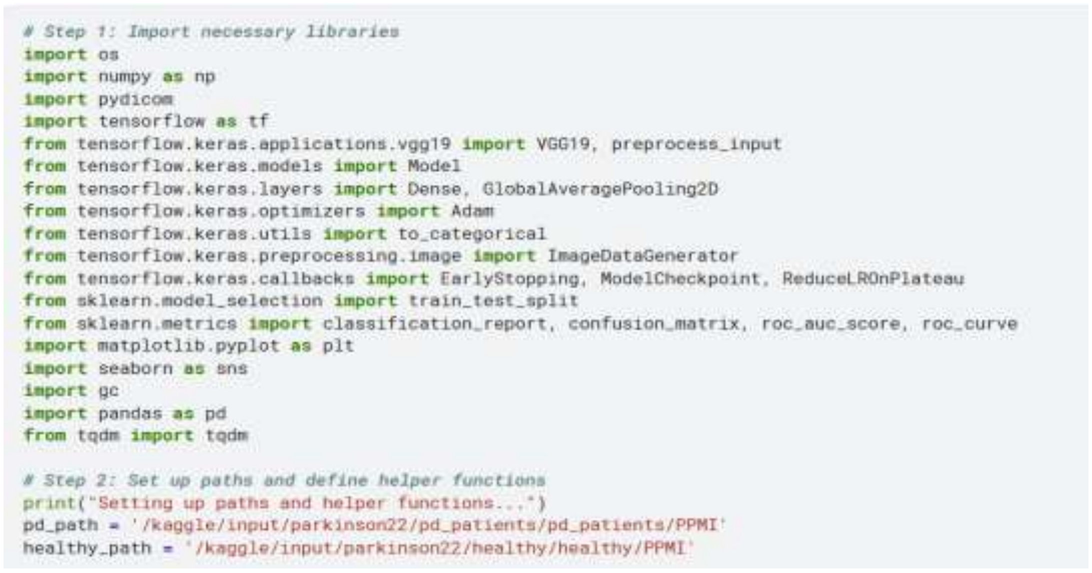
After a thorough comparison, this study finds that GCN performs better in PD classification than conventional CNN-based models like VGG19 when optimized with SSO. GCN is extremely useful in medical imaging applications, notably for examining brain connection patterns in patients with Parkinson's disease (PD), due to its capacity to describe spatial dependencies and leverage graph-structured relationships. According to the results, GCN+SSO is a better method for managing intricate medical datasets, enhancing feature selection, and raising classification precision. To further improve automated PD detection and classification, future studies can investigate hybrid models that combine several deep learning architectures, multimodal medical data fusion, and improved metaheuristic optimization strategies.

The outcomes show how well optimization strategies work to increase the precision of PD detection. With an accuracy of 84%, the standalone VGG19 model demonstrated its baseline performance. The accuracy of VGG19 increased dramatically to 95% when optimized with SSO, demonstrating the advantages of meta-heuristic optimization. A non-optimized model performed well but might be improved, achieving 94%. GCN improved by SSO achieved the maximum accuracy of 97.72%, demonstrating its outstanding feature extraction and classification capabilities. All models' continuously strong F1-scores attest to their dependability, and the enhanced methods offer more accurate and effective PD detection. These results highlight how crucial optimization is to improving deep learning models for medical diagnosis, which will increase early detection and healthcare results.

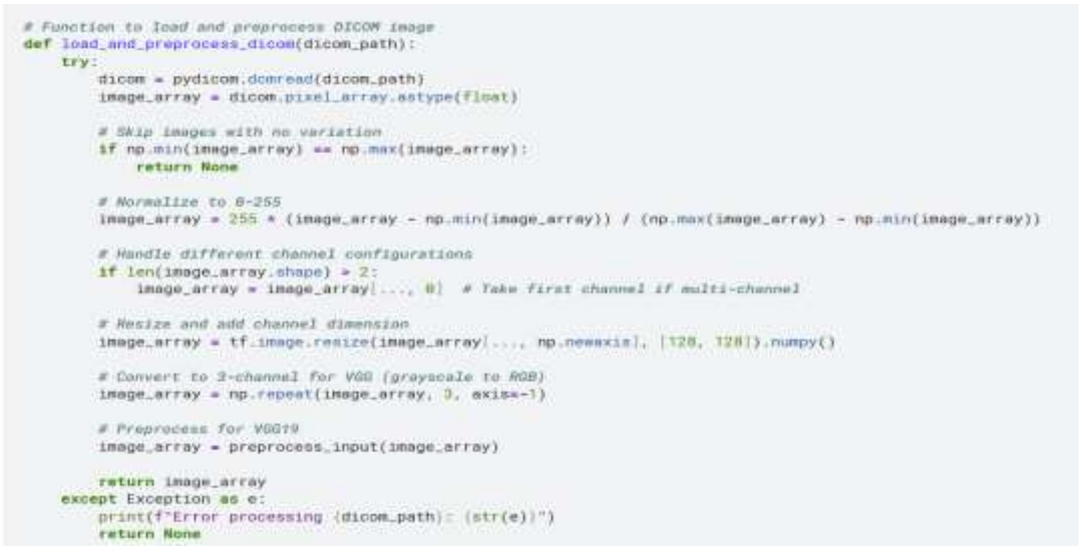
A number of prospective developments to improve the precision and effectiveness of PD detection are included in the research's future scope. First, feature selection and classification performance could be further enhanced by including additional meta-heuristic algorithms like Ant Colony Optimization (ACO) or Particle Swarm Optimization (PSO). Second, adding more varied and sizable samples to the dataset can enhance the model's ability to generalize across various demographic groups. Third, the accuracy of diagnosis can be improved by combining speech and text-based features with multi-modal data, such as MRI, fMRI, and genetic biomarkers. Fourth, early and easily accessible PD screening can be facilitated by real-time implementation in clinical settings through cloud-based frameworks or edge computing. Lastly, by using explainable AI (XAI) methodologies, model interpretability can be enhanced, which would facilitate healthcare professionals' adoption and trust of AI-driven diagnostic tools.

Chapter 6
Appendix

6.1: Vgg19



```python
# Step 1: Import necessary libraries
import os
import numpy as np
import pydicom
import tensorflow as tf
from tensorflow.keras.applications.vgg19 import VGG19, preprocess_input
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint, ReduceLROnPlateau
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix, roc_auc_score, roc_curve
import matplotlib.pyplot as plt
import seaborn as sns
import gc
import pandas as pd
from tqdm import tqdm

# Step 2: Set up paths and define helper functions
print("Setting up paths and helper functions...")
pd_path = '/kaggle/input/parkinson22/pd_patients/pd_patients/PPMI'
healthy_path = '/kaggle/input/parkinson22/healthy/healthy/PPMI'
```



```python
# Function to load and preprocess DICOM image
def load_and_preprocess_dicom(dicom_path):
    try:
        dicom = pydicom.dcmread(dicom_path)
        image_array = dicom.pixel_array.astype(float)

        # Skip images with no variation
        if np.min(image_array) == np.max(image_array):
            return None

        # Normalize to 0-255
        image_array = 255 * (image_array - np.min(image_array)) / (np.max(image_array) - np.min(image_array))

        # Handle different channel configurations
        if len(image_array.shape) > 2:
            image_array = image_array[..., 0]  # Take first channel if multi-channel

        # Resize and add channel dimension
        image_array = tf.image.resize(image_array[..., np.newaxis], [128, 128]).numpy()

        # Convert to 3-channel for VGG (grayscale to RGB)
        image_array = np.repeat(image_array, 3, axis=-1)

        # Preprocess for VGG19
        image_array = preprocess_input(image_array)

        return image_array
    except Exception as e:
        print(f"Error processing {dicom_path}: {str(e)}")
        return None
```

```python
# Generator function for dataset batches
def dataset_generator(file_list, batch_size=32, augment=False):
    datagen = ImageDataGenerator(
        rotation_range=20,
        zoom_range=0.2,
        horizontal_flip=True,
        vertical_flip=False,
        fill_mode='nearest'
    ) if augment else None

    while True:
        np.random.shuffle(file_list)
        for start in range(0, len(file_list), batch_size):
            X_batch = []
            y_batch = []
            end = min(start + batch_size, len(file_list))

            for file_path, label in file_list[start:end]:
                image = load_and_preprocess_dicom(file_path)
                if image is not None:
                    X_batch.append(image)
                    y_batch.append(label)

            if X_batch:
                X_batch_array = np.array(X_batch)
                y_batch_array = to_categorical(y_batch, num_classes=2)

                if augment and datagen:
                    # Only yield the first batch from the generator to match expected behavior
                    for augmented_batch in datagen.flow(X_batch_array, y_batch_array, batch_size=len(X_batch_array)):
                        yield augmented_batch
                        break
                else:
                    yield X_batch_array, y_batch_array
```

```python
# Step 3: Build and compile the model
print("Building the VGG19-based model...")
def build_model(input_shape=(128, 128, 3)):
    base_model = VGG19(weights='imagenet', include_top=False, input_shape=input_shape)

    # Freeze base model layers
    for layer in base_model.layers:
        layer.trainable = False

    x = base_model.output
    x = GlobalAveragePooling2D()(x)
    x = Dense(512, activation='relu')(x)
    x = tf.keras.layers.Dropout(0.5)(x)
    output = Dense(2, activation='softmax')(x)

    model = Model(inputs=base_model.input, outputs=output)

    model.compile(
        optimizer=Adam(learning_rate=0.0001),
        loss='categorical_crossentropy',
        metrics=['accuracy', tf.keras.metrics.AUC(name='auc')]
    )

    return model

model = build_model()
model.summary()
```

```
# Step 4: Train the model
print("Starting training...")
batch_size = 32
train_gen = dataset_generator(train_files, batch_size=batch_size, augment=True)
val_gen = dataset_generator(val_files, batch_size=batch_size, augment=False)

# Calculate steps
steps_per_epoch = len(train_files) // batch_size
validation_steps = max(1, len(val_files) // batch_size)

# Callbacks for training
callbacks = [
    EarlyStopping(monitor='val_loss', patience=7, restore_best_weights=True, verbose=1),
    ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=3, min_lr=1e-6, verbose=1),
    ModelCheckpoint('best_parkinson_model.keras', monitor='val_auc', mode='max', save_best_only=True, verbose=1)
]

# Train the model
history = model.fit(
    train_gen,
    steps_per_epoch=steps_per_epoch,
    epochs=25,
    validation_data=val_gen,
    validation_steps=validation_steps,
    callbacks=callbacks,
    verbose=1
)

# Clean up memory
gc.collect()
```

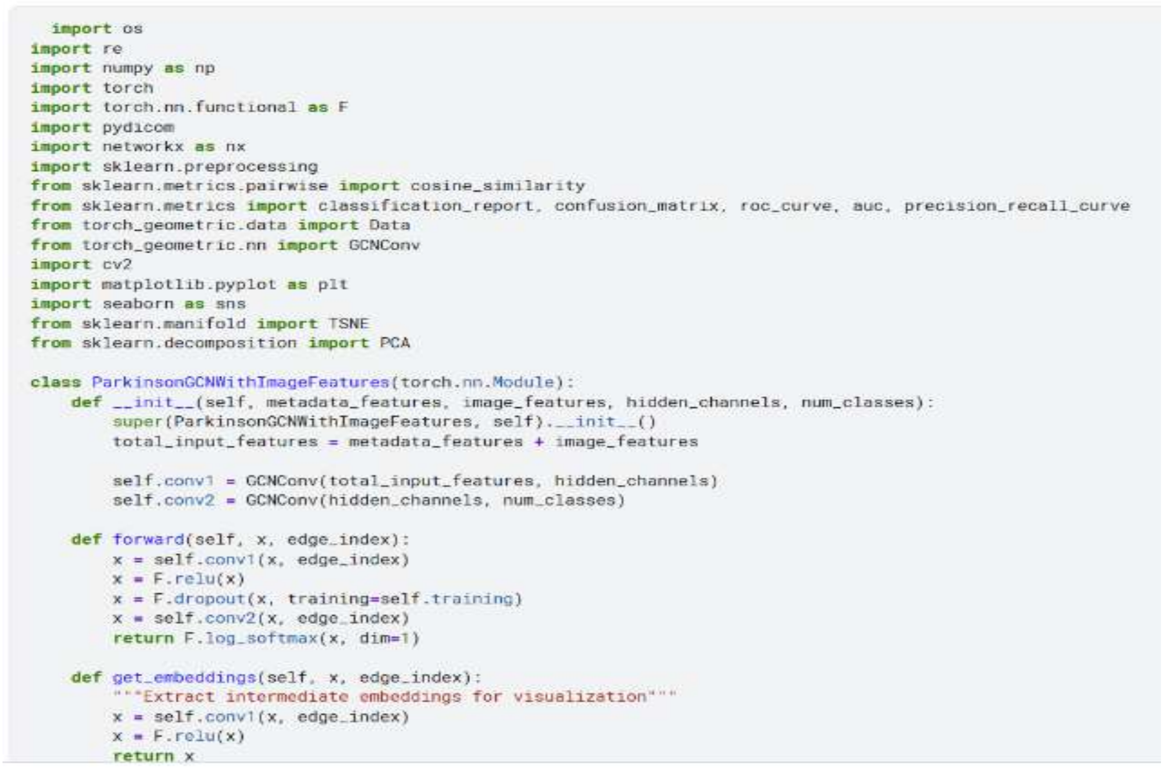
```
def evaluate_on_test_set(model, test_files, batch_size=32):
    all_y_true = []
    all_y_pred = []
    all_X_test = []

    # Process test files in batches
    for i in range(0, len(test_files), batch_size):
        batch_files = test_files[i:i+batch_size]
        X_batch = []
        y_batch = []

        for file_path, label in batch_files:
            image = load_and_preprocess_dicom(file_path)
            if image is not None:
                X_batch.append(image)
                y_batch.append(label)

        if X_batch:
            X_batch = np.array(X_batch)
            y_batch = np.array(y_batch)

            y_pred = model.predict(X_batch)

            all_X_test.extend(X_batch)
            all_y_true.extend(y_batch)
            all_y_pred.extend(y_pred)

    all_X_test = np.array(all_X_test)
    all_y_true = np.array(all_y_true)
    all_y_pred = np.array(all_y_pred)

    return all_X_test, all_y_true, all_y_pred

# Evaluate on full test set
X_test, y_true, y_pred_prob = evaluate_on_test_set(model, test_files)
y_pred_class = np.argmax(y_pred_prob, axis=1)
y_true_categorical = to_categorical(y_true, num_classes=2)
```

```
# Calculate metrics
test_loss, test_accuracy, test_auc = model.evaluate(X_test, y_true_categorical, verbose=1)
print(f"\nTest metrics:")
print(f"Test accuracy: {test_accuracy:.4f}")
print(f"Test AUC: {test_auc:.4f}")

# Print classification report
print("\nClassification Report:")
print(classification_report(y_true, y_pred_class, target_names=['Healthy', 'Parkinson\'s']))

# Generate confusion matrix
cm = confusion_matrix(y_true, y_pred_class)
print("\nConfusion Matrix:")
print(cm)

# Step 6: Visualize results
print("Generating visualizations...")
```

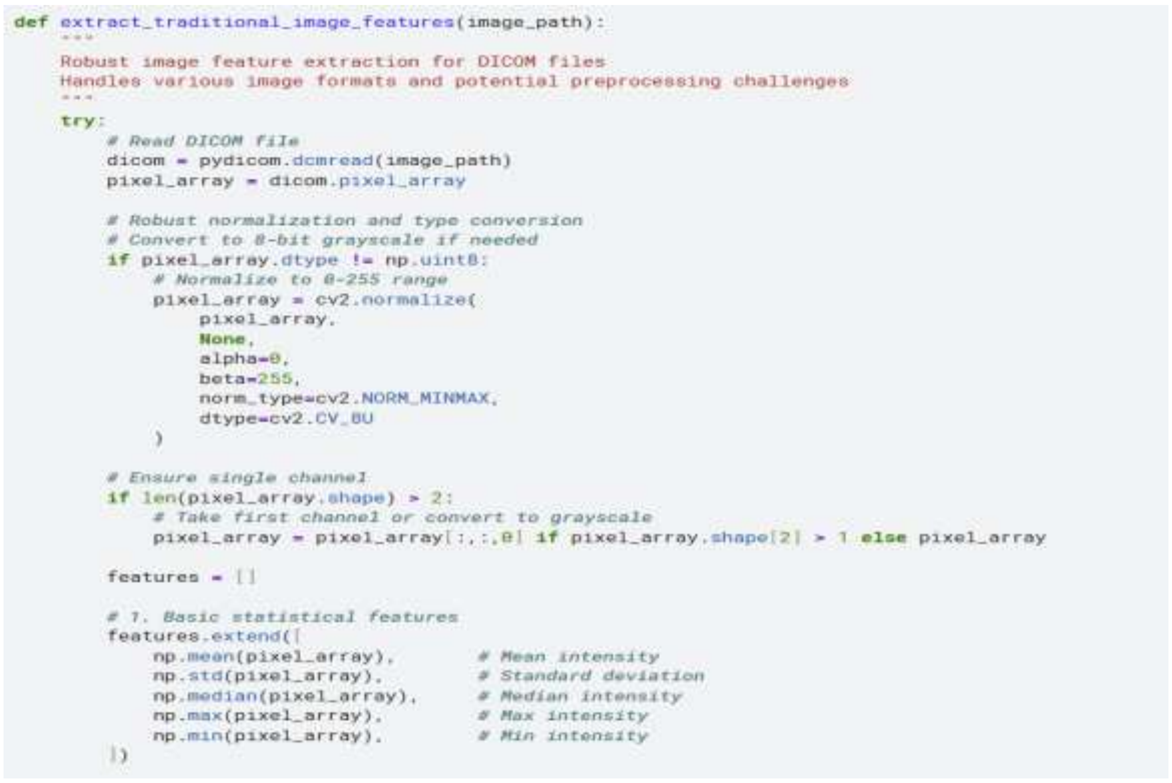
```
# Plot training history
def plot_training_history(history):
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 6))

    # Accuracy plot
    ax1.plot(history.history['accuracy'], label='Train Accuracy')
    ax1.plot(history.history['val_accuracy'], label='Validation Accuracy')
    ax1.set_title('Model Accuracy over Epochs')
    ax1.set_xlabel('Epoch')
    ax1.set_ylabel('Accuracy')
    ax1.legend()
    ax1.grid(True, linestyle='--', alpha=0.6)

    # Loss plot
    ax2.plot(history.history['loss'], label='Train Loss')
    ax2.plot(history.history['val_loss'], label='Validation Loss')
    ax2.set_title('Model Loss over Epochs')
    ax2.set_xlabel('Epoch')
    ax2.set_ylabel('Loss')
    ax2.legend()
    ax2.grid(True, linestyle='--', alpha=0.6)

    plt.tight_layout()
    plt.savefig('training_history.png')
    plt.close()

    # Plot AUC
    plt.figure(figsize=(10, 6))
    plt.plot(history.history['auc'], label='Train AUC')
    plt.plot(history.history['val_auc'], label='Validation AUC')
    plt.title('Model AUC over Epochs')
    plt.xlabel('Epoch')
    plt.ylabel('AUC')
    plt.legend()
    plt.grid(True, linestyle='--', alpha=0.6)
    plt.savefig('auc_history.png')
    plt.close()
    plt.subplot(1, 3, 1)
    acc_diff = [t - v for t, v in zip(history.history['accuracy'], history.history['val_accuracy'])]
    plt.plot(acc_diff)
    plt.axhline(y=0, color='r', linestyle='--')
    plt.title('Train-Val Accuracy Difference')
    plt.xlabel('Epoch')
    plt.ylabel('Difference')
    plt.grid(True, linestyle='--', alpha=0.6)

    plt.subplot(1, 3, 2)
    loss_diff = [t - v for t, v in zip(history.history['loss'], history.history['val_loss'])]
    plt.plot(loss_diff)
    plt.axhline(y=0, color='r', linestyle='--')
    plt.title('Train-Val Loss Difference')
    plt.xlabel('Epoch')
    plt.ylabel('Difference')
    plt.grid(True, linestyle='--', alpha=0.6)

    plt.subplot(1, 3, 3)
    auc_diff = [t - v for t, v in zip(history.history['auc'], history.history['val_auc'])]
    plt.plot(auc_diff)
    plt.axhline(y=0, color='r', linestyle='--')
    plt.title('Train-Val AUC Difference')
    plt.xlabel('Epoch')
    plt.ylabel('Difference')
    plt.grid(True, linestyle='--', alpha=0.6)

    plt.tight_layout()
    plt.savefig('train_val_differences.png')
    plt.close()
```

```
print("Collecting dataset files...")
def collect_files(pd_path, healthy_path):
    pd_files = []
    healthy_files = []

    # Collect PD files
    for subj in tqdm(os.listdir(pd_path), desc="Collecting PD files"):
        subj_dir = os.path.join(pd_path, subj)
        if os.path.isdir(subj_dir):
            for f in os.listdir(subj_dir):
                if f.lower().endswith('.dcm'):
                    pd_files.append((os.path.join(subj_dir, f), 1))

    # Collect healthy files
    for subj in tqdm(os.listdir(healthy_path), desc="Collecting healthy files"):
        subj_dir = os.path.join(healthy_path, subj)
        if os.path.isdir(subj_dir):
            for f in os.listdir(subj_dir):
                if f.lower().endswith('.dcm'):
                    healthy_files.append((os.path.join(subj_dir, f), 0))

    print(f"Found {len(pd_files)} Parkinson's files and {len(healthy_files)} healthy control files")
    return pd_files, healthy_files

pd_files, healthy_files = collect_files(pd_path, healthy_path)
all_files = pd_files + healthy_files
np.random.shuffle(all_files)

# Split into train, validation, and test sets
train_files, test_files = train_test_split(all_files, test_size=0.2, random_state=42)
train_files, val_files = train_test_split(train_files, test_size=0.25, random_state=42)

print(f"Training set: {len(train_files)} files")
print(f"Validation set: {len(val_files)} files")
print(f"Test set: {len(test_files)} files")
```

6.2: GCN



```
  import os
import re
import numpy as np
import torch
import torch.nn.functional as F
import pydicom
import networkx as nx
import sklearn.preprocessing
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.metrics import classification_report, confusion_matrix, roc_curve, auc, precision_recall_curve
from torch_geometric.data import Data
from torch_geometric.nn import GCNConv
import cv2
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.manifold import TSNE
from sklearn.decomposition import PCA

class ParkinsonGCNWithImageFeatures(torch.nn.Module):
    def __init__(self, metadata_features, image_features, hidden_channels, num_classes):
        super(ParkinsonGCNWithImageFeatures, self).__init__()
        total_input_features = metadata_features + image_features

        self.conv1 = GCNConv(total_input_features, hidden_channels)
        self.conv2 = GCNConv(hidden_channels, num_classes)

    def forward(self, x, edge_index):
        x = self.conv1(x, edge_index)
        x = F.relu(x)
        x = F.dropout(x, training=self.training)
        x = self.conv2(x, edge_index)
        return F.log_softmax(x, dim=1)

    def get_embeddings(self, x, edge_index):
        """Extract intermediate embeddings for visualization"""
        x = self.conv1(x, edge_index)
        x = F.relu(x)
        return x
```



```
def extract_traditional_image_features(image_path):
    """
    Robust image feature extraction for DICOM files
    Handles various image formats and potential preprocessing challenges
    """
    try:
        # Read DICOM file
        dicom = pydicom.dcmread(image_path)
        pixel_array = dicom.pixel_array

        # Robust normalization and type conversion
        # Convert to 8-bit grayscale if needed
        if pixel_array.dtype != np.uint8:
            # Normalize to 0-255 range
            pixel_array = cv2.normalize(
                pixel_array,
                None,
                alpha=0,
                beta=255,
                norm_type=cv2.NORM_MINMAX,
                dtype=cv2.CV_8U
            )

        # Ensure single channel
        if len(pixel_array.shape) > 2:
            # Take first channel or convert to grayscale
            pixel_array = pixel_array[:,:,0] if pixel_array.shape[2] > 1 else pixel_array

        features = []

        # 1. Basic statistical features
        features.extend([
            np.mean(pixel_array),        # Mean intensity
            np.std(pixel_array),         # Standard deviation
            np.median(pixel_array),      # Median intensity
            np.max(pixel_array),         # Max intensity
            np.min(pixel_array),         # Min intensity
        ])
```

```python
        # 2. Histogram features with more robust binning
        hist = cv2.calcHist([pixel_array], [0], None, [32], [0, 256])
        hist_normalized = hist / np.sum(hist)  # Normalize histogram
        features.extend(hist_normalized.flatten())

        # 3. Edge detection (Sobel operator)
        try:
            sobel_x = cv2.Sobel(pixel_array, cv2.CV_64F, 1, 0, ksize=3)
            sobel_y = cv2.Sobel(pixel_array, cv2.CV_64F, 0, 1, ksize=3)
            features.extend([
                np.mean(np.abs(sobel_x)),
                np.mean(np.abs(sobel_y)),
                np.std(sobel_x),
                np.std(sobel_y)
            ])
        except Exception as e:
            # Fallback if Sobel fails
            features.extend([0, 0, 0, 0])

        # 4. Robust texture features using alternative methods
        try:
            # Variance of Laplacian as blur detection
            laplacian_var = cv2.Laplacian(pixel_array, cv2.CV_64F).var()
            features.append(laplacian_var)
        except Exception:
            features.append(0)

        # 5. Entropy calculation
        try:
            histogram = cv2.calcHist([pixel_array], [0], None, [256], [0, 256])
            histogram_normalized = histogram / np.sum(histogram)
            entropy = -np.sum([p * np.log2(p) if p > 0 else 0 for p in histogram_normalized])
            features.append(entropy)
        except Exception:
            features.append(0)

        # Ensure consistent feature vector length
        while len(features) < 64:
            features.append(0.0)
```

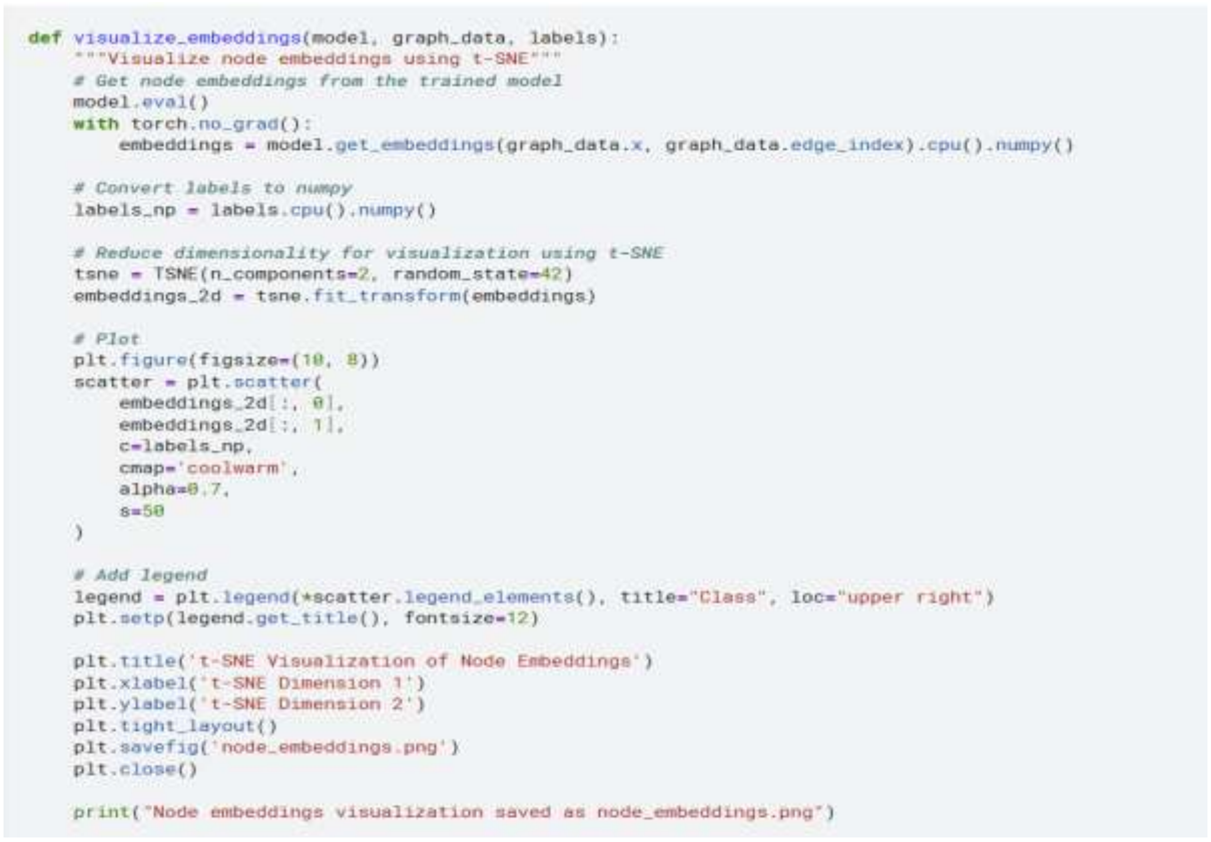
```python
    except Exception as e:
        print(f"Comprehensive error in feature extraction from {image_path}: {e}")
        return np.zeros(64)  # Return zero vector if extraction completely fails

def extract_advanced_dicom_metadata(dicom_path, extract_images=False):
    """
    Enhanced metadata extraction with optional image feature extraction
    """
    features = []
    image_features = []
    processed_files = 0
    unique_features = set()

    print(f"Scanning directory: {dicom_path}")

    for root, _, files in os.walk(dicom_path):
        for filename in files:
            if filename.lower().endswith('.dcm'):
                full_path = os.path.join(root, filename)
                try:
                    ds = pydicom.dcmread(full_path)

                    # Metadata feature vector (same as previous implementation)
                    feature_vector = [
                        parse_dicom_age(ds.get('PatientAge', '0Y')),
                        float(ds.get('PatientWeight', 0) or 0),
                        float(ds.get('SliceThickness', 0) or 0),
                        float(ds.get('PixelSpacing', [0,0])[0] if ds.get('PixelSpacing') else 0),
                        float(len(ds.dir())),
                        float(ds.get('Rows', 0) or 0),
                        float(ds.get('Columns', 0) or 0),
                    ]

                    # Image feature extraction (optional)
                    if extract_images:
                        image_feat = extract_traditional_image_features(full_path)
                        image_features.append(image_feat)
                    # Create a hashable representation
                    feature_tuple = tuple(feature_vector)

                    if feature_tuple not in unique_features:
                        features.append(feature_vector)
                        unique_features.add(feature_tuple)

                    processed_files += 1

                    if processed_files % 1000 == 0:
                        print(f"Processed {processed_files} DICOM files")

                except Exception as e:
                    print(f"Error processing {filename}: {e}")

    # Standardize features
    features_array = np.array(features)
    scaler = sklearn.preprocessing.StandardScaler()
    features_scaled = scaler.fit_transform(features_array)

    if extract_images:
        # Standardize and combine metadata and image features
        image_features_array = np.array(image_features)
        image_scaler = sklearn.preprocessing.StandardScaler()
        image_features_scaled = image_scaler.fit_transform(image_features_array)

        return features_scaled, image_features_scaled

    return features_scaled
```

```
def parse_dicom_age(age_str):
    """
    Parse DICOM age string with robust error handling
    """
    try:
        match = re.match(r'(\d+)([A-Z])', str(age_str))
        if match:
            value, unit = match.groups()
            value = int(value)

            if unit == 'Y':
                return float(value)
            elif unit == 'M':
                return float(value / 12)
            elif unit == 'W':
                return float(value / 52)
            elif unit == 'D':
                return float(value / 365)
        return 0.0
    except Exception:
        return 0.0
```

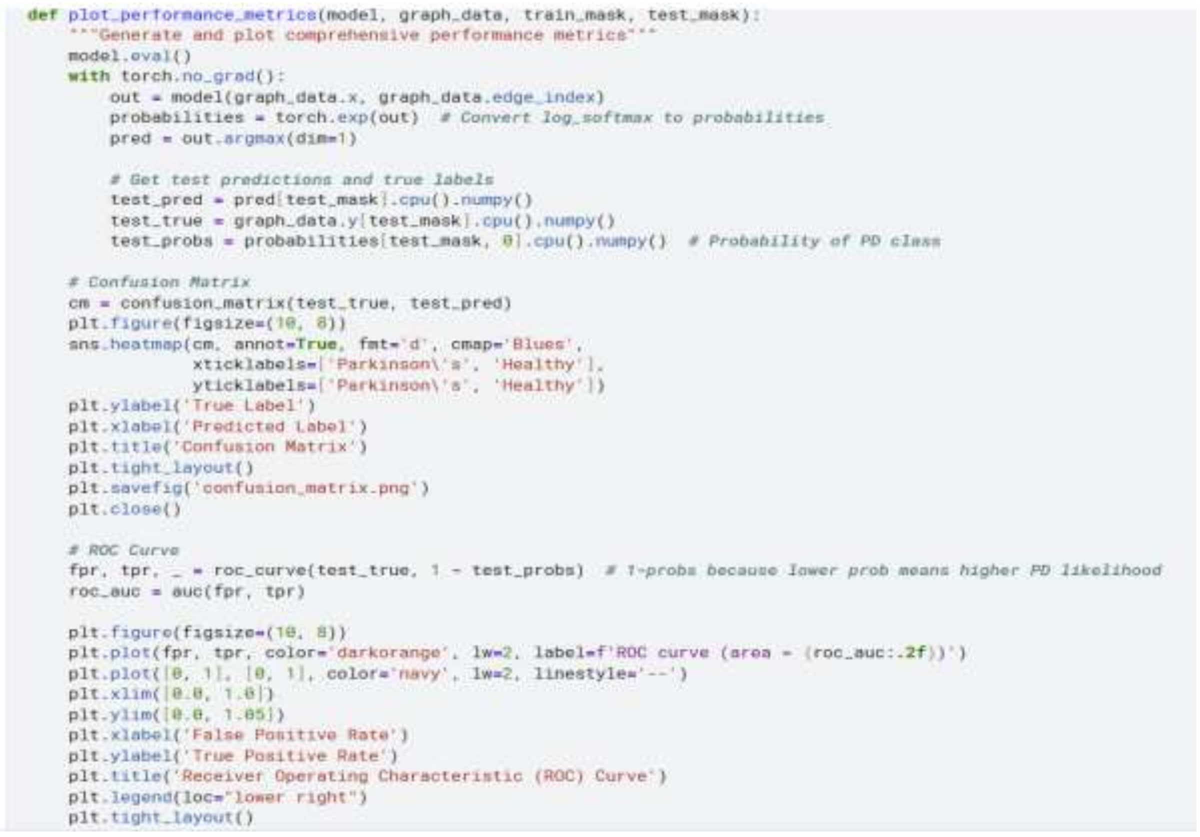
```python
def prepare_dataset(pd_path, healthy_path):
    """
    Prepare dataset for GCN with combined metadata and traditional image features
    """
    print("\nPreparing dataset:")

    # Extract metadata and image features
    pd_metadata, pd_image_features = extract_advanced_dicom_metadata(pd_path, extract_images=True)
    healthy_metadata, healthy_image_features = extract_advanced_dicom_metadata(healthy_path, extract_images=True)

    # Balance dataset
    min_samples = min(len(pd_metadata), len(healthy_metadata))

    np.random.seed(42)
    pd_indices = np.random.choice(len(pd_metadata), min_samples, replace=False)
    healthy_indices = np.random.choice(len(healthy_metadata), min_samples, replace=False)

    # Select balanced features
    pd_metadata_balanced = pd_metadata[pd_indices]
    pd_image_balanced = pd_image_features[pd_indices]
    healthy_metadata_balanced = healthy_metadata[healthy_indices]
    healthy_image_balanced = healthy_image_features[healthy_indices]

    # Combine metadata and image features
    all_metadata = np.vstack([pd_metadata_balanced, healthy_metadata_balanced])
    all_image_features = np.vstack([pd_image_balanced, healthy_image_balanced])
    combined_features = np.hstack([all_metadata, all_image_features])

    # Create labels
    pd_labels = torch.zeros(len(pd_metadata_balanced), dtype=torch.long)
    healthy_labels = torch.ones(len(healthy_metadata_balanced), dtype=torch.long)
    all_labels = torch.cat([pd_labels, healthy_labels])
```

```python
    # Create similarity graph
    similarity_matrix = cosine_similarity(combined_features)
    threshold = np.percentile(similarity_matrix, 95)
    adjacency_matrix = (similarity_matrix >= threshold).astype(int)

    G = nx.from_numpy_array(adjacency_matrix)

    # Convert to PyTorch Geometric Data
    edge_index = torch.tensor(list(G.edges)).t().contiguous()
    x = torch.tensor(combined_features, dtype=torch.float)

    graph_data = Data(x=x, edge_index=edge_index, y=all_labels)

    print("\nBalanced Dataset Summary:")
    print(f"Total samples: {len(all_labels)}")
    print(f"Parkinson's samples: {len(pd_labels)}")
    print(f"Healthy samples: {len(healthy_labels)}")
    print(f"Combined feature dimensions: {combined_features.shape}")

    return graph_data, G

def visualize_graph_structure(G, labels, title="Graph Connectivity Visualization"):
    """Visualize the constructed graph structure"""
    plt.figure(figsize=(12, 10))

    # Convert labels to numpy for easier manipulation
    labels_np = labels.numpy()

    # Set node colors based on labels (blue for healthy, red for PD)
    node_colors = ['blue' if label == 1 else 'red' for label in labels_np]

    # Spring layout for better visualization
    pos = nx.spring_layout(G, seed=42)
    # Plot
    nx.draw_networkx(
        G,
        pos=pos,
        node_color=node_colors,
        node_size=50,
        width=0.3,
        edge_color='gray',
        alpha=0.7,
        with_labels=False
    )

    # Add legend
    pd_patch = plt.Line2D([0], [0], marker='o', color='w', markerfacecolor='red', markersize=10, label="Parkinson's")
    healthy_patch = plt.Line2D([0], [0], marker='o', color='w', markerfacecolor='blue', markersize=10, label='Healthy')
    plt.legend(handles=[pd_patch, healthy_patch], loc='upper right')

    plt.title(title)
    plt.axis('off')
    plt.tight_layout()
    plt.savefig('graph_structure.png')
    plt.close()

    print("Graph visualization saved as graph_structure.png")
```

```
def visualize_feature_importance(model, feature_names=None):
    """Analyze feature importance based on model weights"""
    # Extract weights from the first layer
    weights = model.conv1.lin.weight.detach().cpu().numpy()

    # Sum absolute weights across all output nodes
    importance = np.abs(weights).sum(axis=0)

    # Normalize to get relative importance
    importance = importance / importance.sum()

    # Generate feature names if not provided
    if feature_names is None:
        metadata_names = ['Age', 'Weight', 'SliceThickness', 'PixelSpacing', 'DicomAttributes', 'Rows', 'Columns']
        image_names = [f'ImgFeat_{i+1}' for i in range(64)]
        feature_names = metadata_names + image_names

    # Sort features by importance
    sorted_idx = np.argsort(importance)[::-1]
    top_features = sorted_idx[:15]  # Top 15 features

    # Plot
    plt.figure(figsize=(12, 8))
    plt.barh(np.array(feature_names)[top_features], importance[top_features], color='skyblue')
    plt.xlabel('Relative Importance')
    plt.title('Top 15 Feature Importance')
    plt.gca().invert_yaxis()  # Display most important at the top
    plt.tight_layout()
    plt.savefig('feature_importance.png')
    plt.close()

    print("Feature importance visualization saved as feature_importance.png")

    return importance
```



```python
def visualize_embeddings(model, graph_data, labels):
    """Visualize node embeddings using t-SNE"""
    # Get node embeddings from the trained model
    model.eval()
    with torch.no_grad():
        embeddings = model.get_embeddings(graph_data.x, graph_data.edge_index).cpu().numpy()

    # Convert labels to numpy
    labels_np = labels.cpu().numpy()

    # Reduce dimensionality for visualization using t-SNE
    tsne = TSNE(n_components=2, random_state=42)
    embeddings_2d = tsne.fit_transform(embeddings)

    # Plot
    plt.figure(figsize=(10, 8))
    scatter = plt.scatter(
        embeddings_2d[:, 0],
        embeddings_2d[:, 1],
        c=labels_np,
        cmap='coolwarm',
        alpha=0.7,
        s=50
    )

    # Add legend
    legend = plt.legend(*scatter.legend_elements(), title="Class", loc="upper right")
    plt.setp(legend.get_title(), fontsize=12)

    plt.title('t-SNE Visualization of Node Embeddings')
    plt.xlabel('t-SNE Dimension 1')
    plt.ylabel('t-SNE Dimension 2')
    plt.tight_layout()
    plt.savefig('node_embeddings.png')
    plt.close()

    print("Node embeddings visualization saved as node_embeddings.png")
```



```python
def plot_performance_metrics(model, graph_data, train_mask, test_mask):
    """Generate and plot comprehensive performance metrics"""
    model.eval()
    with torch.no_grad():
        out = model(graph_data.x, graph_data.edge_index)
        probabilities = torch.exp(out)  # Convert log_softmax to probabilities
        pred = out.argmax(dim=1)

        # Get test predictions and true labels
        test_pred = pred[test_mask].cpu().numpy()
        test_true = graph_data.y[test_mask].cpu().numpy()
        test_probs = probabilities[test_mask, 0].cpu().numpy()  # Probability of PD class

    # Confusion Matrix
    cm = confusion_matrix(test_true, test_pred)
    plt.figure(figsize=(10, 8))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=['Parkinson\'s', 'Healthy'],
                yticklabels=['Parkinson\'s', 'Healthy'])
    plt.ylabel('True Label')
    plt.xlabel('Predicted Label')
    plt.title('Confusion Matrix')
    plt.tight_layout()
    plt.savefig('confusion_matrix.png')
    plt.close()

    # ROC Curve
    fpr, tpr, _ = roc_curve(test_true, 1 - test_probs)  # 1-probs because lower prob means higher PD likelihood
    roc_auc = auc(fpr, tpr)

    plt.figure(figsize=(10, 8))
    plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (area = {roc_auc:.2f})')
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver Operating Characteristic (ROC) Curve')
    plt.legend(loc="lower right")
    plt.tight_layout()
```

```
    plt.savefig('roc_curve.png')
    plt.close()

    # Precision-Recall Curve
    precision, recall, _ = precision_recall_curve(test_true, 1 - test_probs)
    pr_auc = auc(recall, precision)

    plt.figure(figsize=(10, 8))
    plt.plot(recall, precision, color='green', lw=2, label=f'PR curve (area = {pr_auc:.2f})')
    plt.xlabel('Recall')
    plt.ylabel('Precision')
    plt.title('Precision-Recall Curve')
    plt.legend(loc="lower left")
    plt.tight_layout()
    plt.savefig('precision_recall_curve.png')
    plt.close()

    # Training history visualization (requires training history to be collected during training)
    print("\nClassification Report:")
    print(classification_report(
        test_true,
        test_pred,
        target_names=['Parkinson\'s', 'Healthy'],
        digits=4
    ))

    print(f'ROC AUC: {roc_auc:.4f}')
    print(f'Precision-Recall AUC: {pr_auc:.4f}')

    print("Performance visualizations saved as confusion_matrix.png, roc_curve.png, and precision_recall_curve.png")

    # Return metrics dictionary
    return {
        'confusion_matrix': cm,
        'roc_auc': roc_auc,
        'pr_auc': pr_auc,
        'classification_report': classification_report(
            test_true, test_pred,
            target_names=['Parkinson\'s', 'Healthy'],
            digits=4,
```

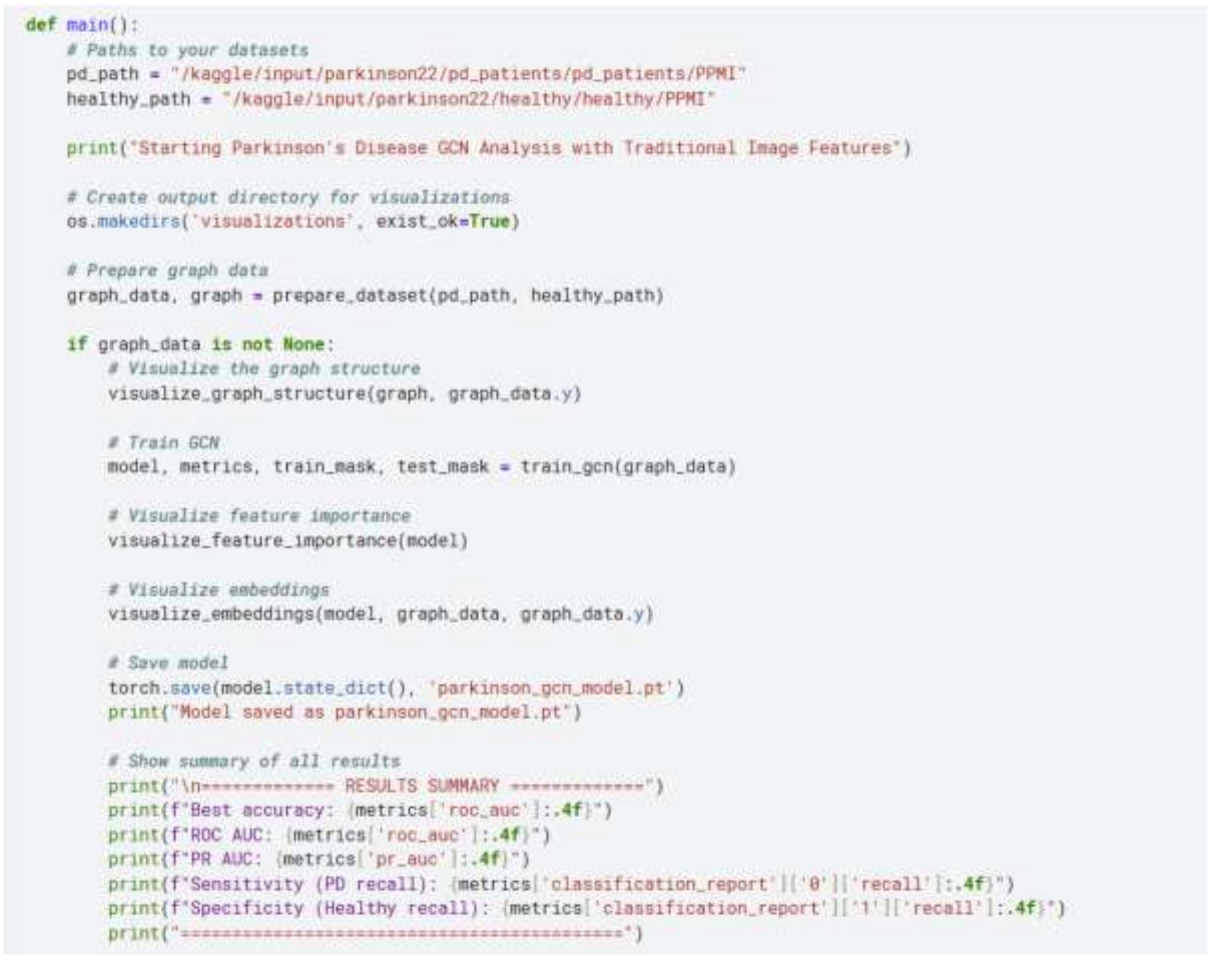
```
        'classification_report': classification_report(
            test_true, test_pred,
            target_names=['Parkinson\'s', 'Healthy'],
            digits=4,
```

```
            output_dict=True
        )
    }

def train_gcn(graph_data, save_history=True):
    """
    Train Graph Convolutional Network with combined features
    Enhanced with history tracking for visualization
    """
    # Split data
    train_mask = torch.rand(graph_data.num_nodes) < 0.7
    test_mask = ~train_mask

    # Updated model initialization with combined feature dimensions
    model = ParkinsonGCNWithImageFeatures(
        metadata_features=7,   # From original metadata features
        image_features=64,     # Traditional image features
        hidden_channels=64,
        num_classes=2
    )

    # Optimizer
    optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
    scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=50, gamma=0.1)

    # Training loop
    best_accuracy = 0
    patience = 20
    patience_counter = 0
    best_model = None
```

```
# History tracking
history = {
    'train_loss': [],
    'train_acc': [],
    'test_acc': [],
    'epochs': []
}

print("\nTraining GCN:")
for epoch in range(200):
    model.train()
    optimizer.zero_grad()
    out = model(graph_data.x, graph_data.edge_index)
    loss = F.nll_loss(out[train_mask], graph_data.y[train_mask])
    loss.backward()
    optimizer.step()
    scheduler.step()

    # Calculate training accuracy
    pred_train = out[train_mask].argmax(dim=1)
    train_correct = (pred_train == graph_data.y[train_mask]).sum()
    train_acc = int(train_correct) / int(train_mask.sum())

    if save_history:
        history['train_loss'].append(loss.item())
        history['train_acc'].append(train_acc)

    # Periodic evaluation
    if epoch % 10 == 0:
        model.eval()
        with torch.no_grad():
            pred = out.argmax(dim=1)
            correct = (pred[test_mask] == graph_data.y[test_mask]).sum()
            acc = int(correct) / int(test_mask.sum())
```

```
            if save_history:
                history['test_acc'].append(acc)
                history['epochs'].append(epoch)

        print(f'Epoch {epoch}: Loss {loss.item():.4f}, Train Acc: {train_acc:.4f}, Test Accuracy: {acc:.4f}')

        # Early stopping and best model tracking
        if acc > best_accuracy:
            best_accuracy = acc
            patience_counter = 0
            best_model = model.state_dict()
        else:
            patience_counter += 1

        if patience_counter >= patience:
            print(f"Early stopping at epoch {epoch}")
            break

# Visualize training history
if save_history and len(history['epochs']) > 1:
    plt.figure(figsize=(12, 5))

    plt.subplot(1, 2, 1)
    plt.plot(range(len(history['train_loss'])), history['train_loss'], label='Training Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title('Training Loss')

    plt.subplot(1, 2, 2)
    plt.plot(history['epochs'], history['test_acc'], 'o-', label='Test Accuracy')
    plt.plot(range(len(history['train_acc'])), history['train_acc'], label='Train Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.title('Model Accuracy')
```

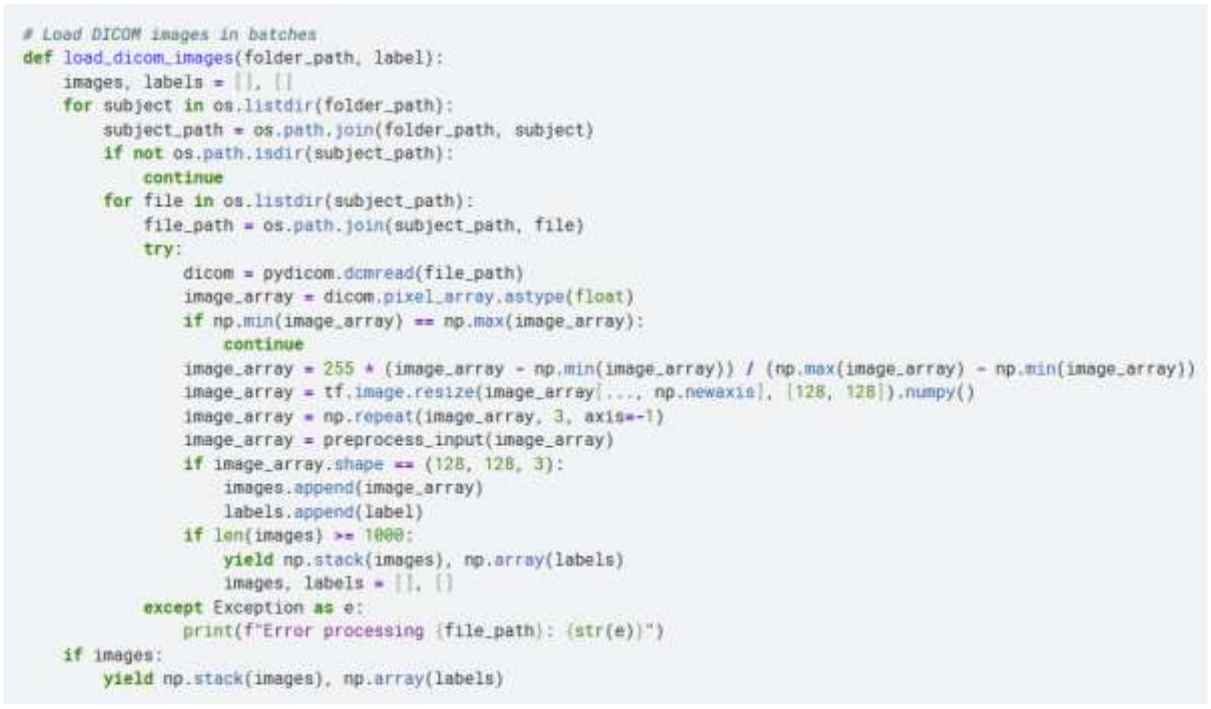
```
        plt.tight_layout()
        plt.savefig('training_history.png')
        plt.close()

        print("Training history visualization saved as training_history.png")

    # Load best model for final evaluation
    model.load_state_dict(best_model)

    # Generate and plot performance metrics
    metrics = plot_performance_metrics(model, graph_data, train_mask, test_mask)

    print(f"\nBest Test Accuracy: {best_accuracy:.4f}")
    return model, metrics, train_mask, test_mask
```



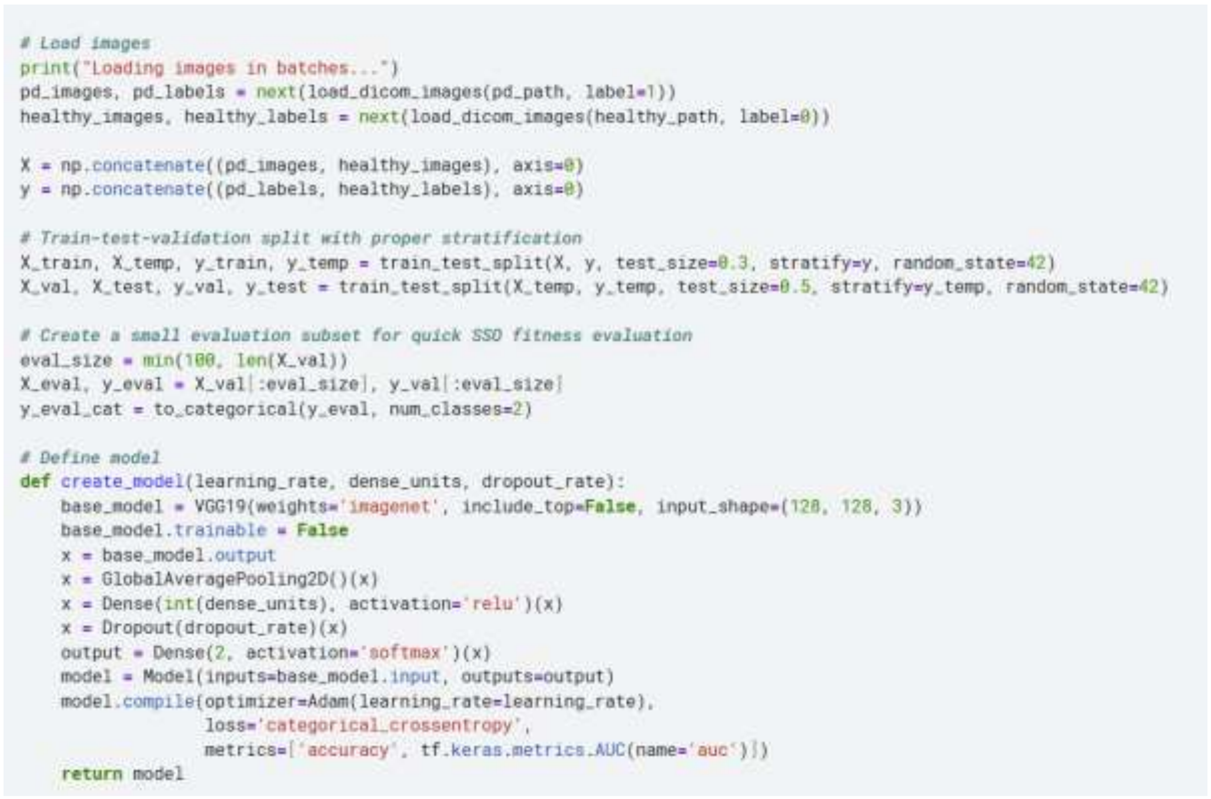
```
def main():
    # Paths to your datasets
    pd_path = "/kaggle/input/parkinson22/pd_patients/pd_patients/PPMI"
    healthy_path = "/kaggle/input/parkinson22/healthy/healthy/PPMI"

    print("Starting Parkinson's Disease GCN Analysis with Traditional Image Features")

    # Create output directory for visualizations
    os.makedirs('visualizations', exist_ok=True)

    # Prepare graph data
    graph_data, graph = prepare_dataset(pd_path, healthy_path)

    if graph_data is not None:
        # Visualize the graph structure
        visualize_graph_structure(graph, graph_data.y)

        # Train GCN
        model, metrics, train_mask, test_mask = train_gcn(graph_data)

        # Visualize feature importance
        visualize_feature_importance(model)

        # Visualize embeddings
        visualize_embeddings(model, graph_data, graph_data.y)

        # Save model
        torch.save(model.state_dict(), 'parkinson_gcn_model.pt')
        print("Model saved as parkinson_gcn_model.pt")

        # Show summary of all results
        print("\n============= RESULTS SUMMARY =============")
        print(f"Best accuracy: {metrics['roc_auc']:.4f}")
        print(f"ROC AUC: {metrics['roc_auc']:.4f}")
        print(f"PR AUC: {metrics['pr_auc']:.4f}")
        print(f"Sensitivity (PD recall): {metrics['classification_report']['0']['recall']:.4f}")
        print(f"Specificity (Healthy recall): {metrics['classification_report']['1']['recall']:.4f}")
        print("==========================================")
```

6.3: VGG19 with SSO

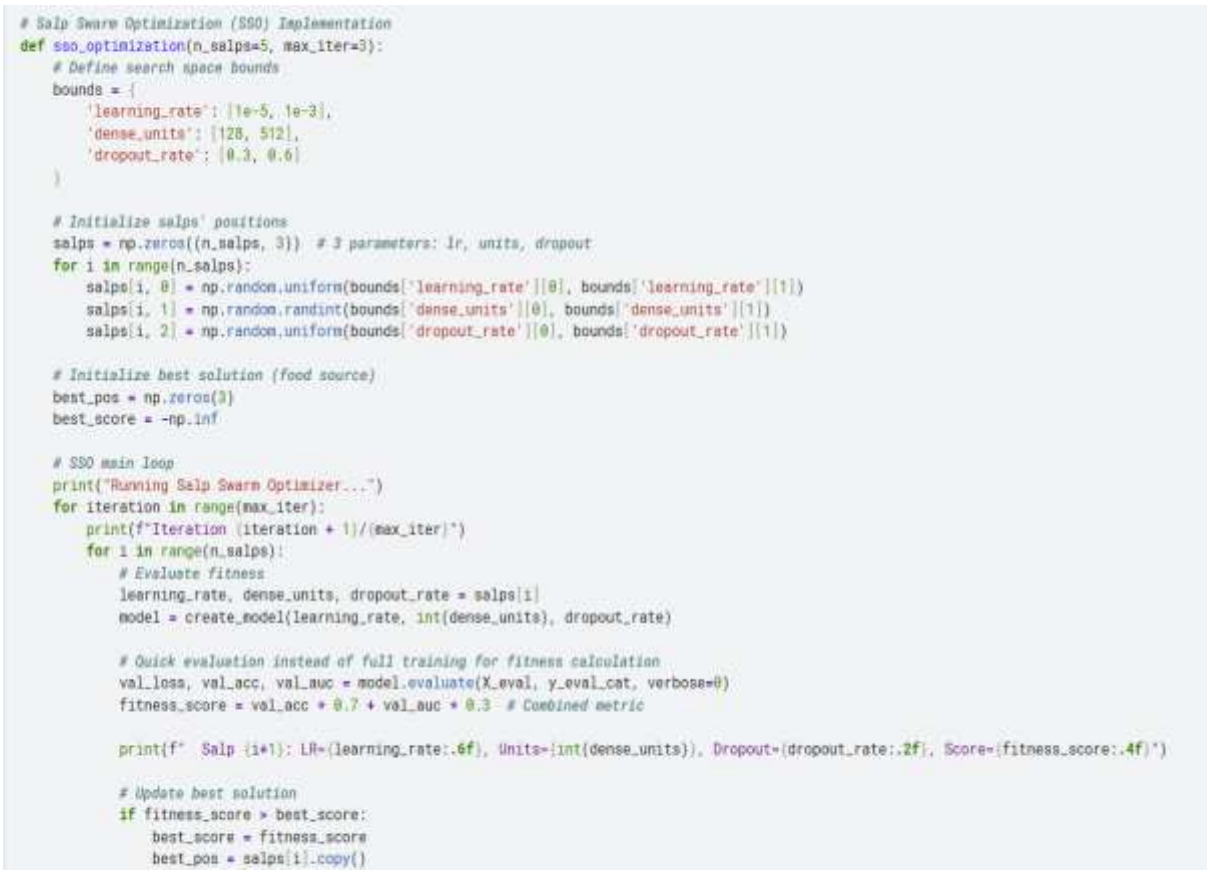
```python
import os
import numpy as np
import pydicom
import tensorflow as tf
from tensorflow.keras.applications.vgg19 import VGG19, preprocess_input
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D, Dropout
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix, roc_curve, auc
import matplotlib.pyplot as plt
import seaborn as sns
import gc

# Set up paths
print("Setting up paths...")
pd_path = '/kaggle/input/parkinson22/pd_patients/pd_patients/PPMI'
healthy_path = '/kaggle/input/parkinson22/healthy/healthy/PPMI'
```



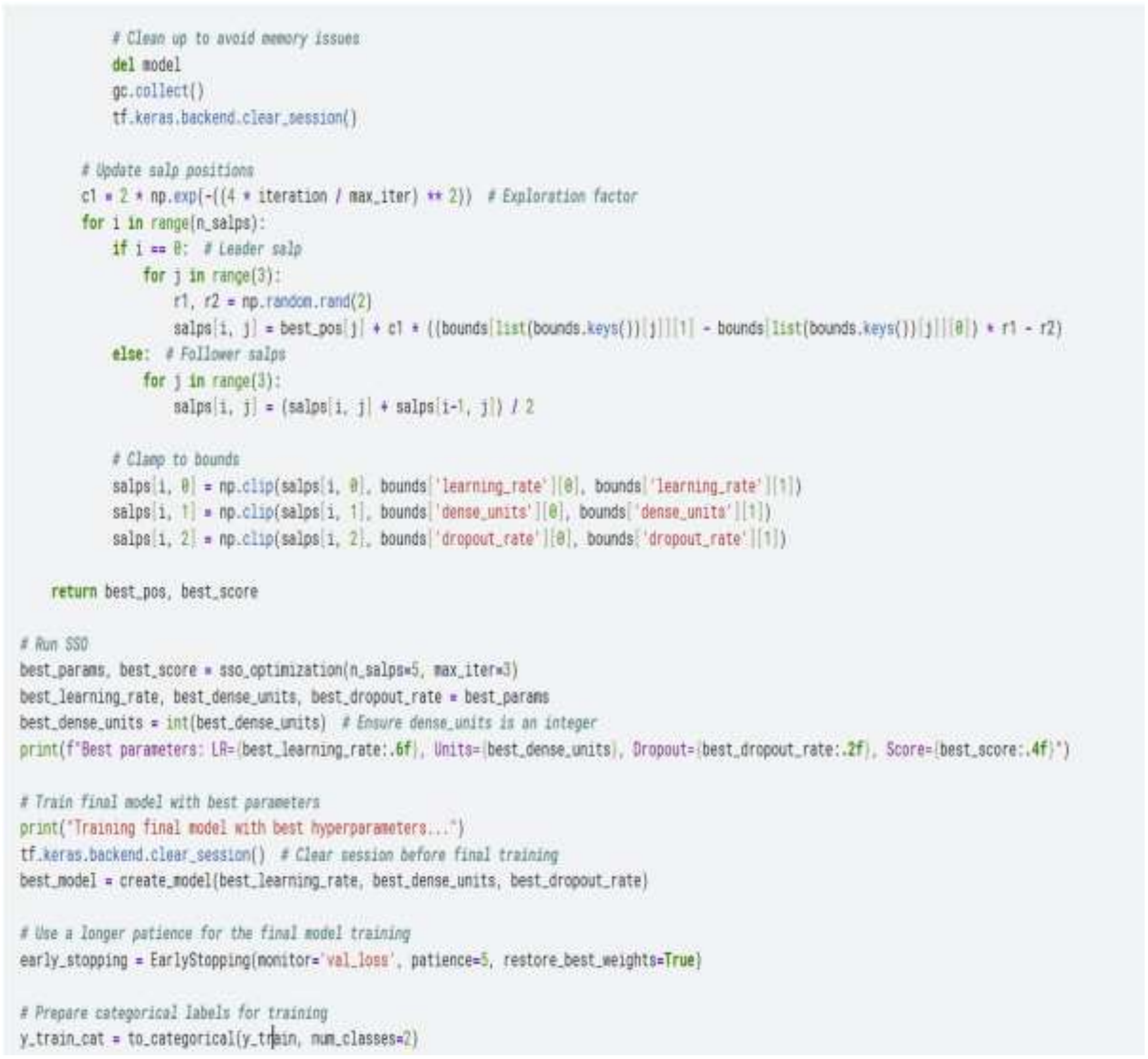
```python
# Load DICOM images in batches
def load_dicom_images(folder_path, label):
    images, labels = [], []
    for subject in os.listdir(folder_path):
        subject_path = os.path.join(folder_path, subject)
        if not os.path.isdir(subject_path):
            continue
        for file in os.listdir(subject_path):
            file_path = os.path.join(subject_path, file)
            try:
                dicom = pydicom.dcmread(file_path)
                image_array = dicom.pixel_array.astype(float)
                if np.min(image_array) == np.max(image_array):
                    continue
                image_array = 255 * (image_array - np.min(image_array)) / (np.max(image_array) - np.min(image_array))
                image_array = tf.image.resize(image_array[..., np.newaxis], [128, 128]).numpy()
                image_array = np.repeat(image_array, 3, axis=-1)
                image_array = preprocess_input(image_array)
                if image_array.shape == (128, 128, 3):
                    images.append(image_array)
                    labels.append(label)
                if len(images) >= 1000:
                    yield np.stack(images), np.array(labels)
                    images, labels = [], []
            except Exception as e:
                print(f"Error processing {file_path}: {str(e)}")
    if images:
        yield np.stack(images), np.array(labels)
```



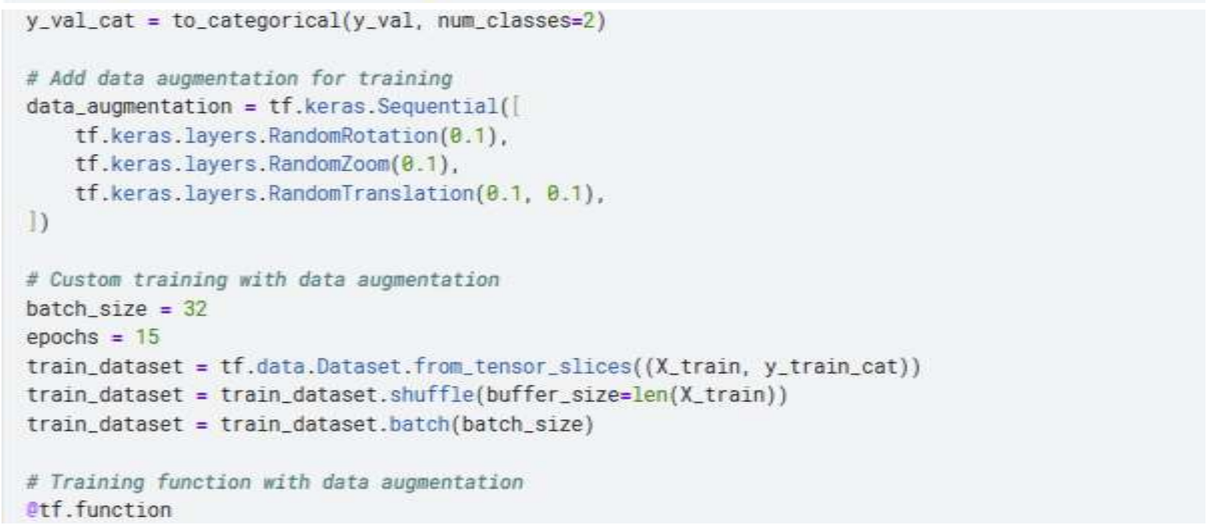
```python
# Load images
print("Loading images in batches...")
pd_images, pd_labels = next(load_dicom_images(pd_path, label=1))
healthy_images, healthy_labels = next(load_dicom_images(healthy_path, label=0))

X = np.concatenate((pd_images, healthy_images), axis=0)
y = np.concatenate((pd_labels, healthy_labels), axis=0)

# Train-test-validation split with proper stratification
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3, stratify=y, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, stratify=y_temp, random_state=42)

# Create a small evaluation subset for quick SSO fitness evaluation
eval_size = min(100, len(X_val))
X_eval, y_eval = X_val[:eval_size], y_val[:eval_size]
y_eval_cat = to_categorical(y_eval, num_classes=2)

# Define model
def create_model(learning_rate, dense_units, dropout_rate):
    base_model = VGG19(weights='imagenet', include_top=False, input_shape=(128, 128, 3))
    base_model.trainable = False
    x = base_model.output
    x = GlobalAveragePooling2D()(x)
    x = Dense(int(dense_units), activation='relu')(x)
    x = Dropout(dropout_rate)(x)
    output = Dense(2, activation='softmax')(x)
    model = Model(inputs=base_model.input, outputs=output)
    model.compile(optimizer=Adam(learning_rate=learning_rate),
                  loss='categorical_crossentropy',
                  metrics=['accuracy', tf.keras.metrics.AUC(name='auc')])
    return model
```



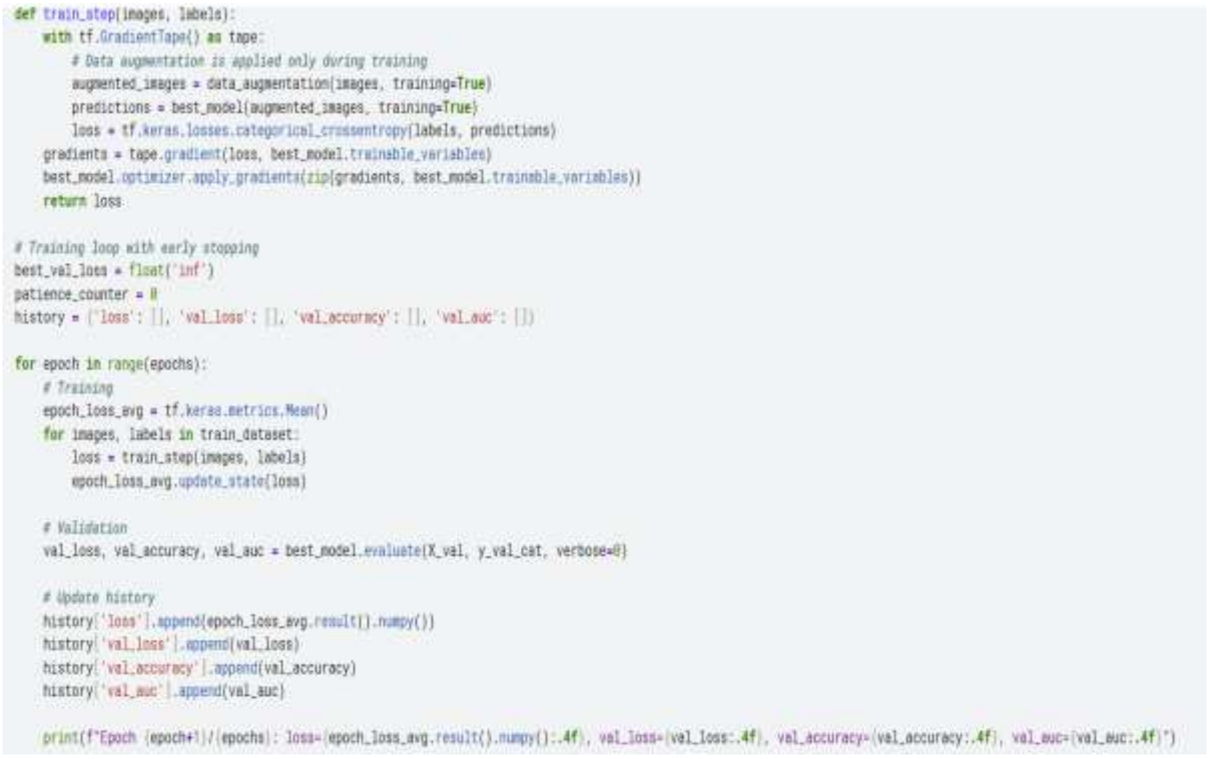
```python
# Salp Swarm Optimization (SSO) Implementation
def sso_optimization(n_salps=5, max_iter=3):
    # Define search space bounds
    bounds = {
        'learning_rate': [1e-5, 1e-3],
        'dense_units': [128, 512],
        'dropout_rate': [0.3, 0.6]
    }

    # Initialize salps' positions
    salps = np.zeros((n_salps, 3))  # 3 parameters: lr, units, dropout
    for i in range(n_salps):
        salps[i, 0] = np.random.uniform(bounds['learning_rate'][0], bounds['learning_rate'][1])
        salps[i, 1] = np.random.randint(bounds['dense_units'][0], bounds['dense_units'][1])
        salps[i, 2] = np.random.uniform(bounds['dropout_rate'][0], bounds['dropout_rate'][1])

    # Initialize best solution (food source)
    best_pos = np.zeros(3)
    best_score = -np.inf

    # SSO main loop
    print("Running Salp Swarm Optimizer...")
    for iteration in range(max_iter):
        print(f"Iteration {iteration + 1}/{max_iter}")
        for i in range(n_salps):
            # Evaluate fitness
            learning_rate, dense_units, dropout_rate = salps[i]
            model = create_model(learning_rate, int(dense_units), dropout_rate)

            # Quick evaluation instead of full training for fitness calculation
            val_loss, val_acc, val_auc = model.evaluate(X_eval, y_eval_cat, verbose=0)
            fitness_score = val_acc * 0.7 + val_auc * 0.3  # Combined metric

            print(f"  Salp {i+1}: LR={learning_rate:.6f}, Units={int(dense_units)}, Dropout={dropout_rate:.2f}, Score={fitness_score:.4f}")

            # Update best solution
            if fitness_score > best_score:
                best_score = fitness_score
                best_pos = salps[i].copy()
```



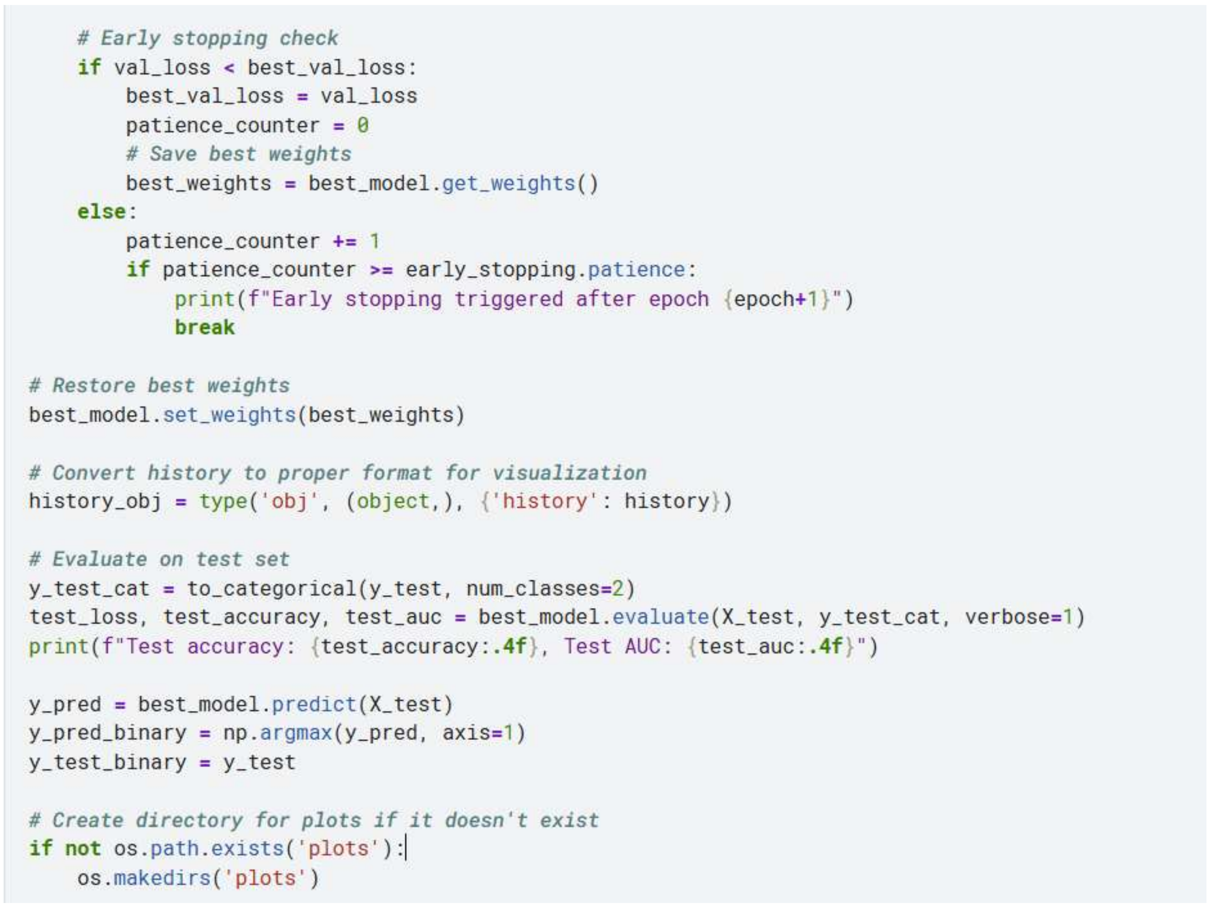
```
            # Clean up to avoid memory issues
            del model
            gc.collect()
            tf.keras.backend.clear_session()

        # Update salp positions
        c1 = 2 * np.exp(-((4 * iteration / max_iter) ** 2))  # Exploration factor
        for i in range(n_salps):
            if i == 0:  # Leader salp
                for j in range(3):
                    r1, r2 = np.random.rand(2)
                    salps[i, j] = best_pos[j] + c1 * ((bounds[list(bounds.keys())[j]][1] - bounds[list(bounds.keys())[j]][0]) * r1 - r2)
            else:  # Follower salps
                for j in range(3):
                    salps[i, j] = (salps[i, j] + salps[i-1, j]) / 2

            # Clamp to bounds
            salps[i, 0] = np.clip(salps[i, 0], bounds['learning_rate'][0], bounds['learning_rate'][1])
            salps[i, 1] = np.clip(salps[i, 1], bounds['dense_units'][0], bounds['dense_units'][1])
            salps[i, 2] = np.clip(salps[i, 2], bounds['dropout_rate'][0], bounds['dropout_rate'][1])

    return best_pos, best_score

# Run SSO
best_params, best_score = sso_optimization(n_salps=5, max_iter=3)
best_learning_rate, best_dense_units, best_dropout_rate = best_params
best_dense_units = int(best_dense_units)  # Ensure dense_units is an integer
print(f"Best parameters: LR={best_learning_rate:.6f}, Units={best_dense_units}, Dropout={best_dropout_rate:.2f}, Score={best_score:.4f}")

# Train final model with best parameters
print("Training final model with best hyperparameters...")
tf.keras.backend.clear_session()  # Clear session before final training
best_model = create_model(best_learning_rate, best_dense_units, best_dropout_rate)

# Use a longer patience for the final model training
early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)

# Prepare categorical labels for training
y_train_cat = to_categorical(y_train, num_classes=2)
```



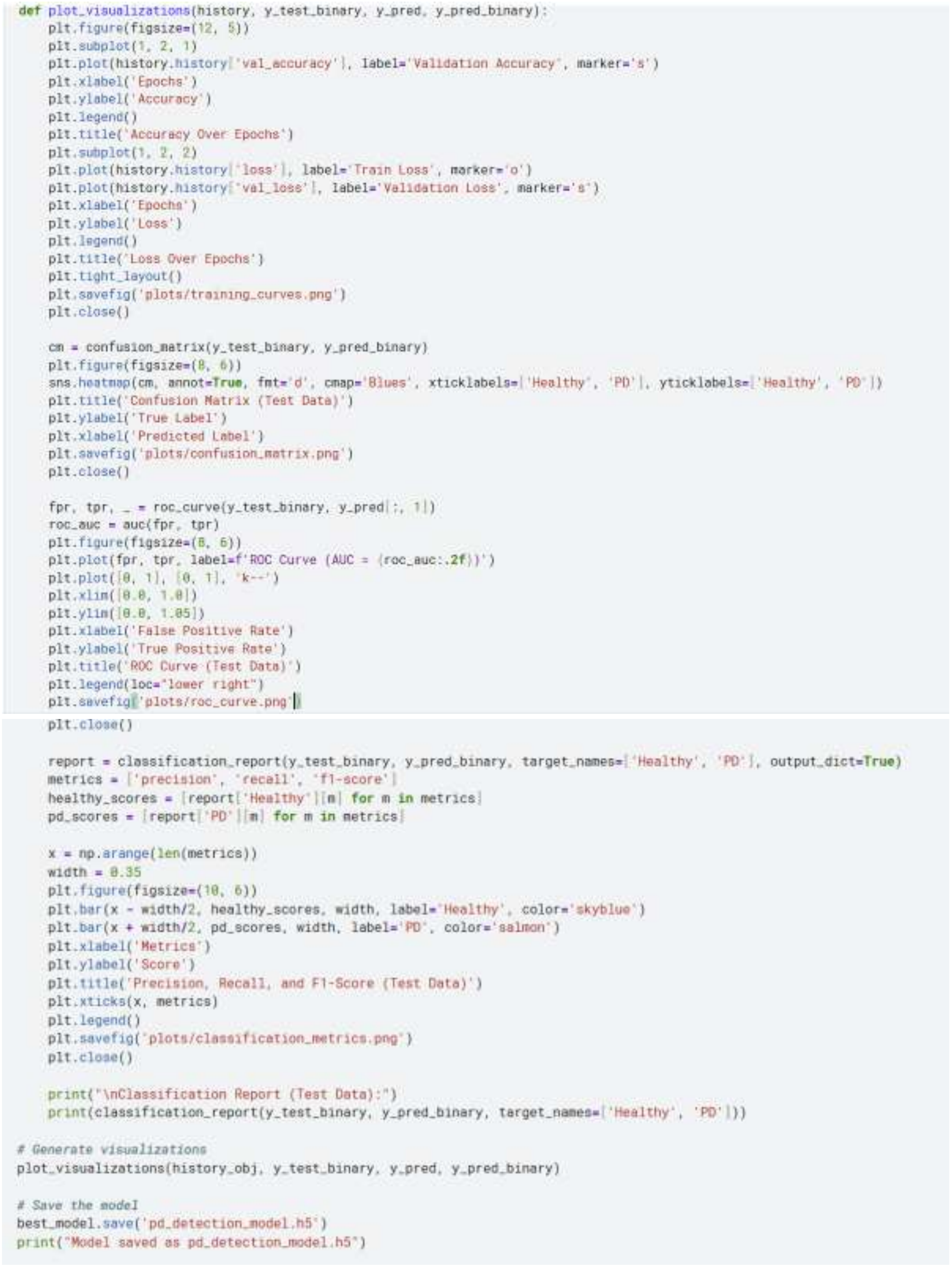
```
y_val_cat = to_categorical(y_val, num_classes=2)

# Add data augmentation for training
data_augmentation = tf.keras.Sequential([
    tf.keras.layers.RandomRotation(0.1),
    tf.keras.layers.RandomZoom(0.1),
    tf.keras.layers.RandomTranslation(0.1, 0.1),
])

# Custom training with data augmentation
batch_size = 32
epochs = 15
train_dataset = tf.data.Dataset.from_tensor_slices((X_train, y_train_cat))
train_dataset = train_dataset.shuffle(buffer_size=len(X_train))
train_dataset = train_dataset.batch(batch_size)

# Training function with data augmentation
@tf.function
```



```python
def train_step(images, labels):
    with tf.GradientTape() as tape:
        # Data augmentation is applied only during training
        augmented_images = data_augmentation(images, training=True)
        predictions = best_model(augmented_images, training=True)
        loss = tf.keras.losses.categorical_crossentropy(labels, predictions)
    gradients = tape.gradient(loss, best_model.trainable_variables)
    best_model.optimizer.apply_gradients(zip(gradients, best_model.trainable_variables))
    return loss

# Training loop with early stopping
best_val_loss = float('inf')
patience_counter = 0
history = {'loss': [], 'val_loss': [], 'val_accuracy': [], 'val_auc': []}

for epoch in range(epochs):
    # Training
    epoch_loss_avg = tf.keras.metrics.Mean()
    for images, labels in train_dataset:
        loss = train_step(images, labels)
        epoch_loss_avg.update_state(loss)

    # Validation
    val_loss, val_accuracy, val_auc = best_model.evaluate(X_val, y_val_cat, verbose=0)

    # Update history
    history['loss'].append(epoch_loss_avg.result().numpy())
    history['val_loss'].append(val_loss)
    history['val_accuracy'].append(val_accuracy)
    history['val_auc'].append(val_auc)

    print(f"Epoch {epoch+1}/{epochs}: loss={epoch_loss_avg.result().numpy():.4f}, val_loss={val_loss:.4f}, val_accuracy={val_accuracy:.4f}, val_auc={val_auc:.4f}")
```

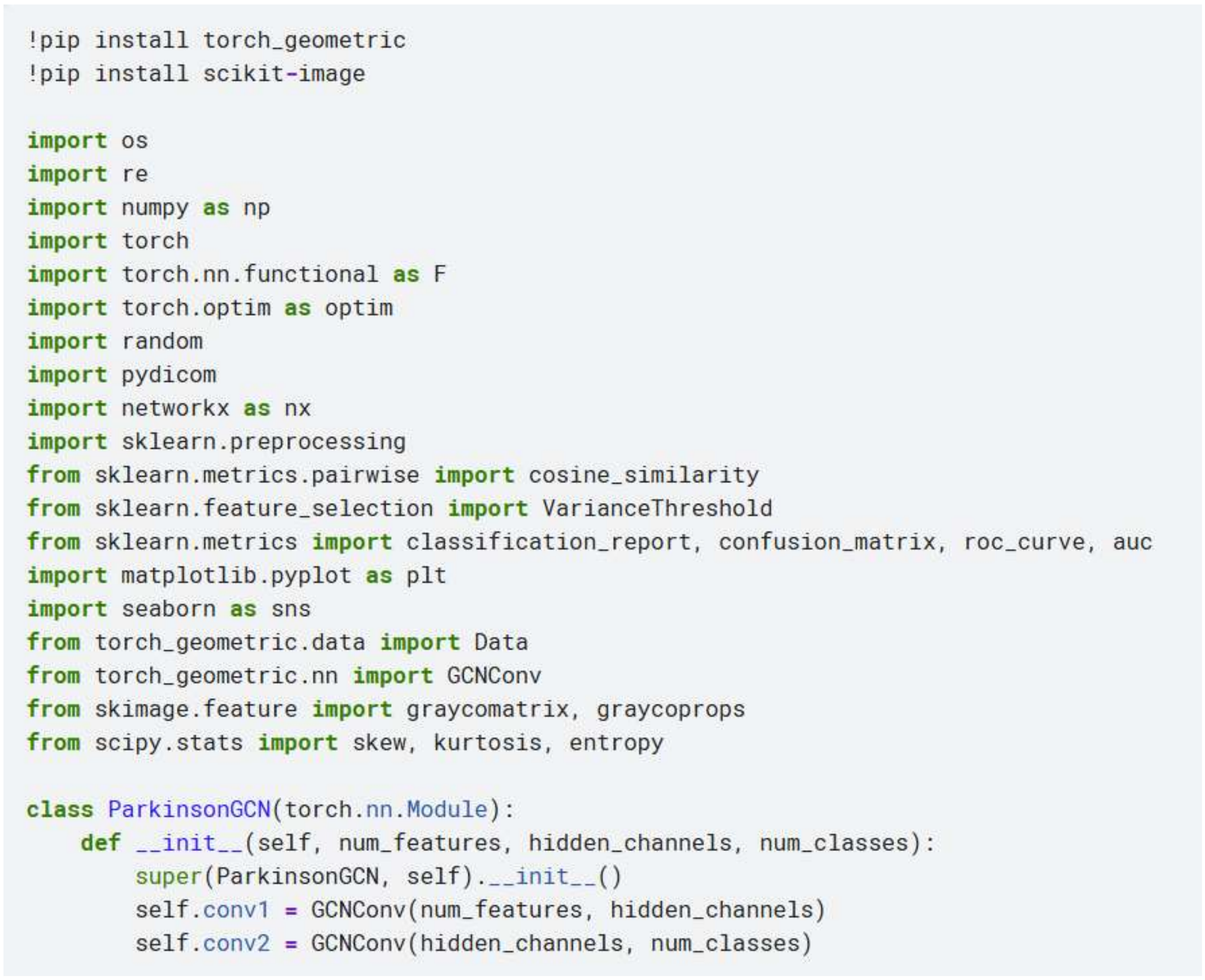


```python
    # Early stopping check
    if val_loss < best_val_loss:
        best_val_loss = val_loss
        patience_counter = 0
        # Save best weights
        best_weights = best_model.get_weights()
    else:
        patience_counter += 1
        if patience_counter >= early_stopping.patience:
            print(f"Early stopping triggered after epoch {epoch+1}")
            break

# Restore best weights
best_model.set_weights(best_weights)

# Convert history to proper format for visualization
history_obj = type('obj', (object,), {'history': history})

# Evaluate on test set
y_test_cat = to_categorical(y_test, num_classes=2)
test_loss, test_accuracy, test_auc = best_model.evaluate(X_test, y_test_cat, verbose=1)
print(f"Test accuracy: {test_accuracy:.4f}, Test AUC: {test_auc:.4f}")

y_pred = best_model.predict(X_test)
y_pred_binary = np.argmax(y_pred, axis=1)
y_test_binary = y_test

# Create directory for plots if it doesn't exist
if not os.path.exists('plots'):
    os.makedirs('plots')
```

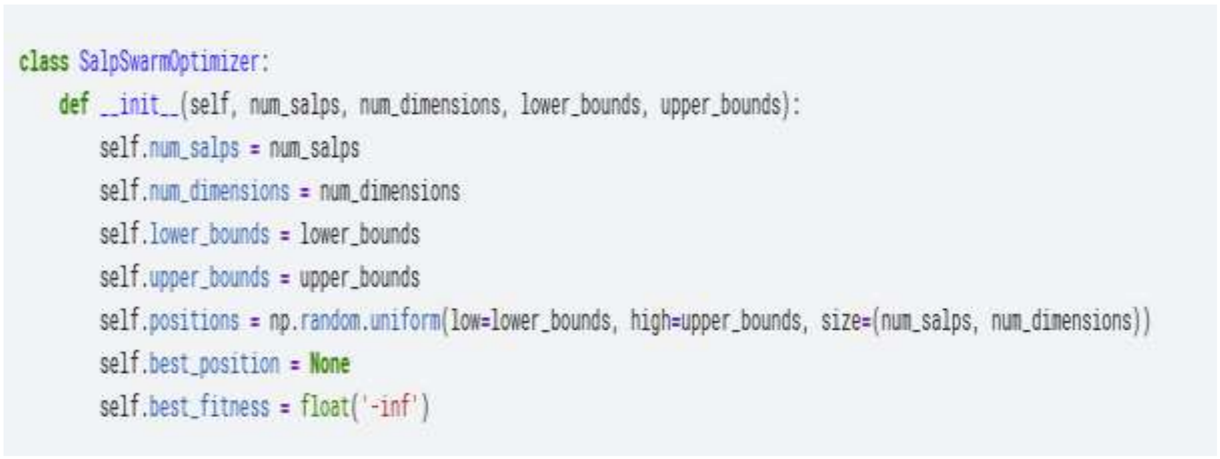


```python
def plot_visualizations(history, y_test_binary, y_pred, y_pred_binary):
    plt.figure(figsize=(12, 5))
    plt.subplot(1, 2, 1)
    plt.plot(history.history['val_accuracy'], label='Validation Accuracy', marker='s')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.title('Accuracy Over Epochs')
    plt.subplot(1, 2, 2)
    plt.plot(history.history['loss'], label='Train Loss', marker='o')
    plt.plot(history.history['val_loss'], label='Validation Loss', marker='s')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.title('Loss Over Epochs')
    plt.tight_layout()
    plt.savefig('plots/training_curves.png')
    plt.close()

    cm = confusion_matrix(y_test_binary, y_pred_binary)
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Healthy', 'PD'], yticklabels=['Healthy', 'PD'])
    plt.title('Confusion Matrix (Test Data)')
    plt.ylabel('True Label')
    plt.xlabel('Predicted Label')
    plt.savefig('plots/confusion_matrix.png')
    plt.close()

    fpr, tpr, _ = roc_curve(y_test_binary, y_pred[:, 1])
    roc_auc = auc(fpr, tpr)
    plt.figure(figsize=(8, 6))
    plt.plot(fpr, tpr, label=f'ROC Curve (AUC = {roc_auc:.2f})')
    plt.plot([0, 1], [0, 1], 'k--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('ROC Curve (Test Data)')
    plt.legend(loc="lower right")
    plt.savefig('plots/roc_curve.png')
    plt.close()

    report = classification_report(y_test_binary, y_pred_binary, target_names=['Healthy', 'PD'], output_dict=True)
    metrics = ['precision', 'recall', 'f1-score']
    healthy_scores = [report['Healthy'][m] for m in metrics]
    pd_scores = [report['PD'][m] for m in metrics]

    x = np.arange(len(metrics))
    width = 0.35
    plt.figure(figsize=(10, 6))
    plt.bar(x - width/2, healthy_scores, width, label='Healthy', color='skyblue')
    plt.bar(x + width/2, pd_scores, width, label='PD', color='salmon')
    plt.xlabel('Metrics')
    plt.ylabel('Score')
    plt.title('Precision, Recall, and F1-Score (Test Data)')
    plt.xticks(x, metrics)
    plt.legend()
    plt.savefig('plots/classification_metrics.png')
    plt.close()

    print("\nClassification Report (Test Data):")
    print(classification_report(y_test_binary, y_pred_binary, target_names=['Healthy', 'PD']))

# Generate visualizations
plot_visualizations(history_obj, y_test_binary, y_pred, y_pred_binary)

# Save the model
best_model.save('pd_detection_model.h5')
print("Model saved as pd_detection_model.h5")
```

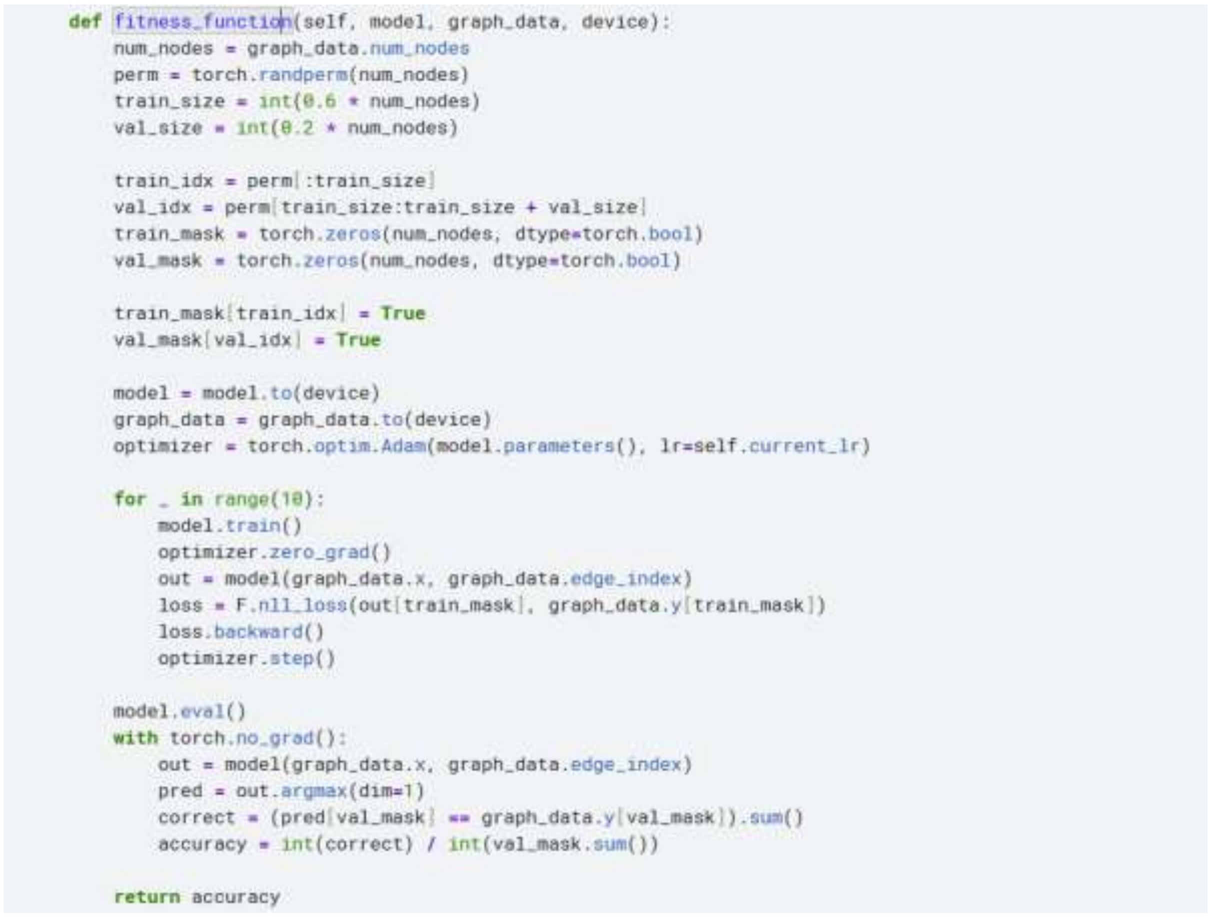
6.4: GCN with SSO



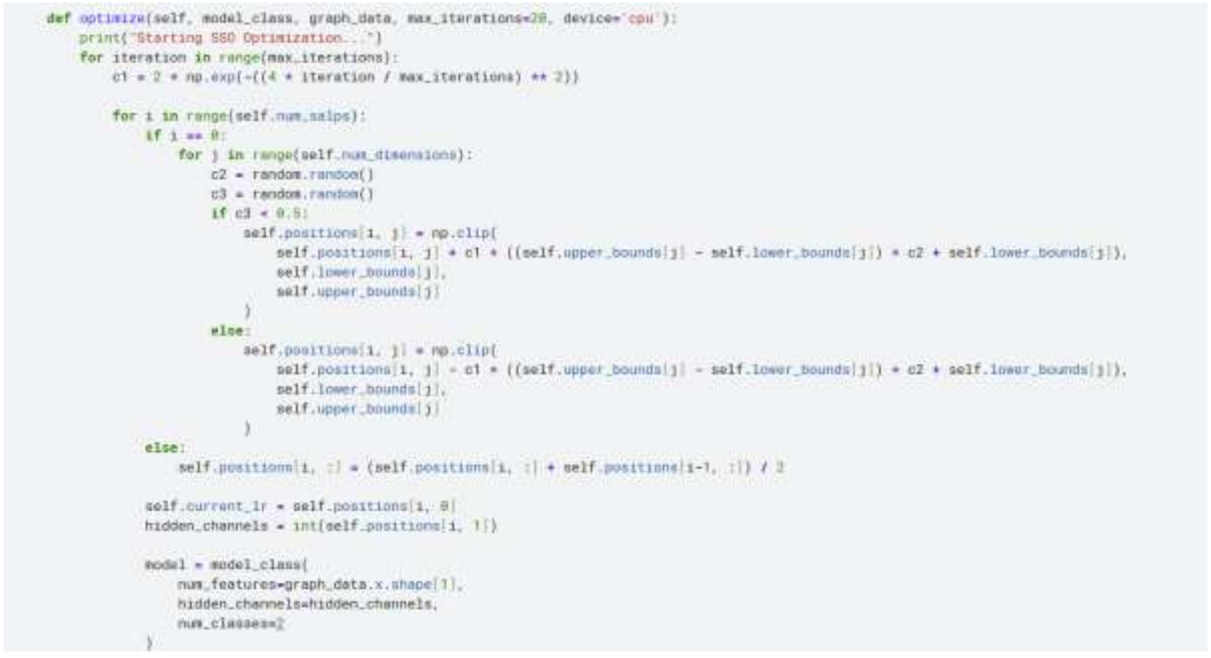
```python
!pip install torch_geometric
!pip install scikit-image

import os
import re
import numpy as np
import torch
import torch.nn.functional as F
import torch.optim as optim
import random
import pydicom
import networkx as nx
import sklearn.preprocessing
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.feature_selection import VarianceThreshold
from sklearn.metrics import classification_report, confusion_matrix, roc_curve, auc
import matplotlib.pyplot as plt
import seaborn as sns
from torch_geometric.data import Data
from torch_geometric.nn import GCNConv
from skimage.feature import graycomatrix, graycoprops
from scipy.stats import skew, kurtosis, entropy

class ParkinsonGCN(torch.nn.Module):
    def __init__(self, num_features, hidden_channels, num_classes):
        super(ParkinsonGCN, self).__init__()
        self.conv1 = GCNConv(num_features, hidden_channels)
        self.conv2 = GCNConv(hidden_channels, num_classes)
```



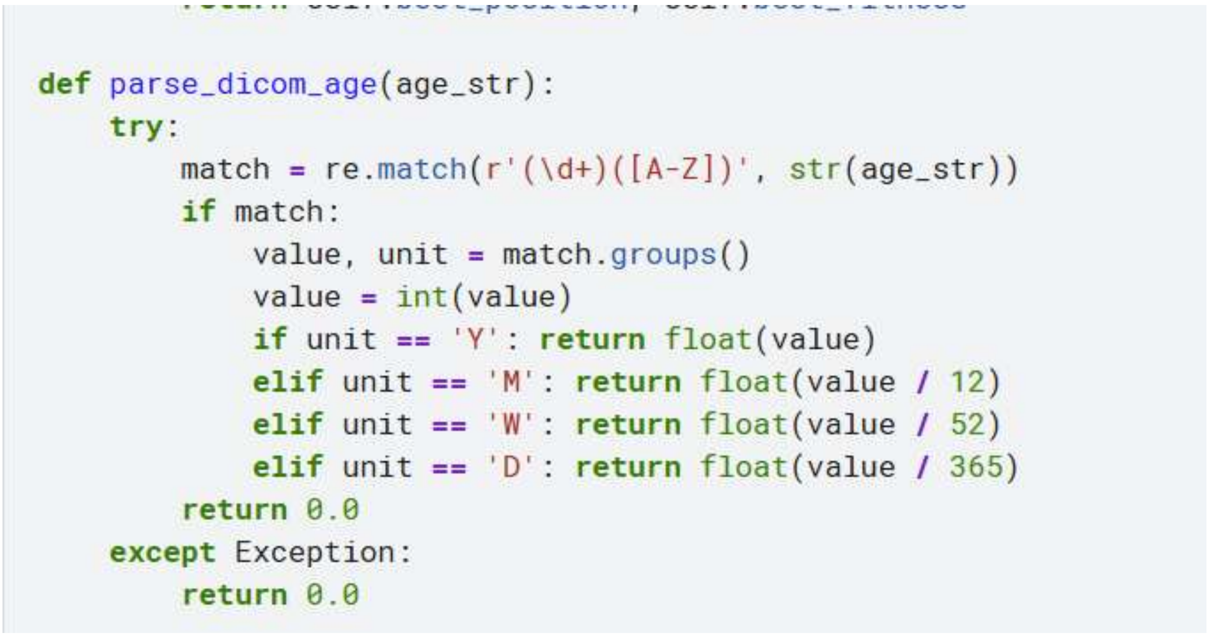
```python
class SalpSwarmOptimizer:
    def __init__(self, num_salps, num_dimensions, lower_bounds, upper_bounds):
        self.num_salps = num_salps
        self.num_dimensions = num_dimensions
        self.lower_bounds = lower_bounds
        self.upper_bounds = upper_bounds
        self.positions = np.random.uniform(low=lower_bounds, high=upper_bounds, size=(num_salps, num_dimensions))
        self.best_position = None
        self.best_fitness = float('-inf')
```



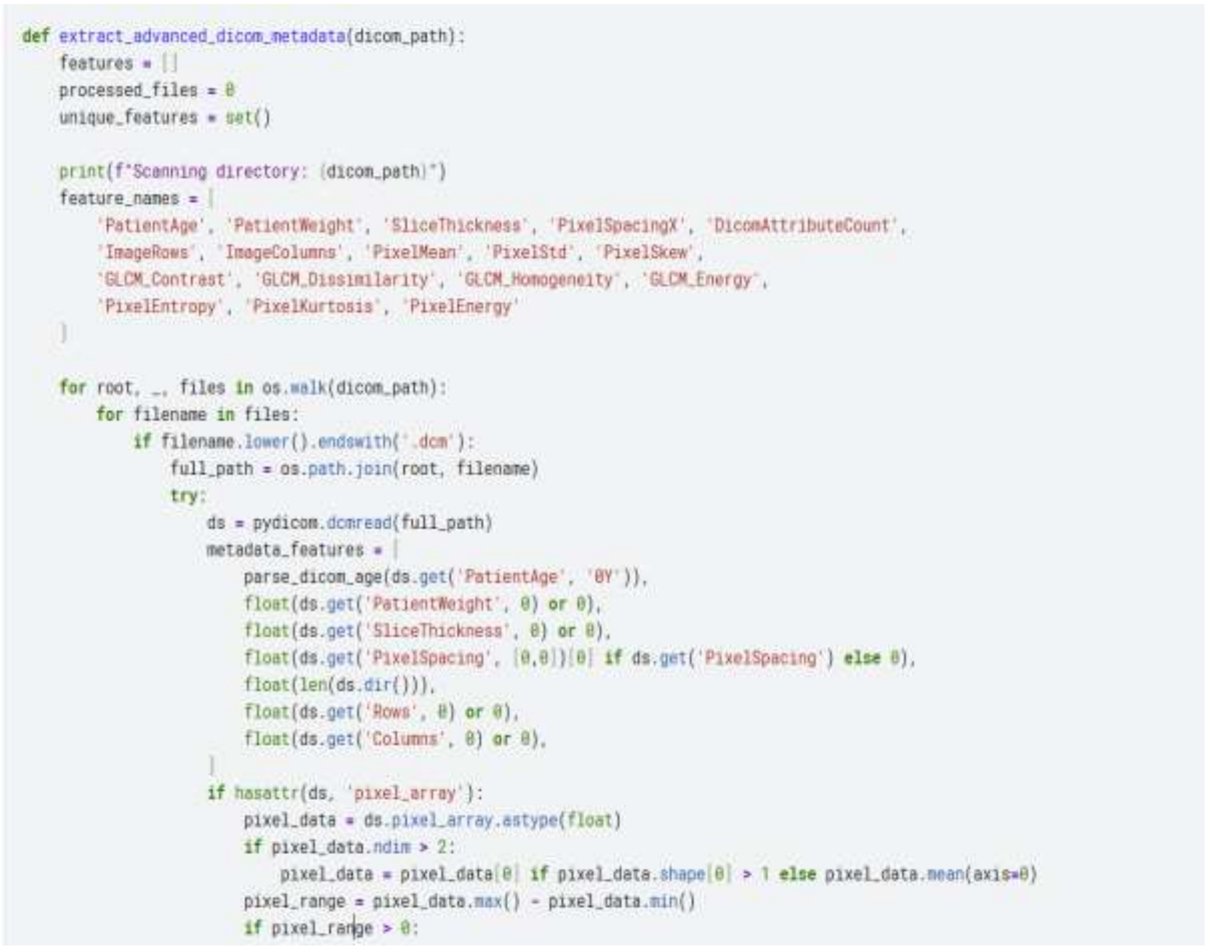
```python
    def fitness_function(self, model, graph_data, device):
        num_nodes = graph_data.num_nodes
        perm = torch.randperm(num_nodes)
        train_size = int(0.6 * num_nodes)
        val_size = int(0.2 * num_nodes)

        train_idx = perm[:train_size]
        val_idx = perm[train_size:train_size + val_size]
        train_mask = torch.zeros(num_nodes, dtype=torch.bool)
        val_mask = torch.zeros(num_nodes, dtype=torch.bool)

        train_mask[train_idx] = True
        val_mask[val_idx] = True

        model = model.to(device)
        graph_data = graph_data.to(device)
        optimizer = torch.optim.Adam(model.parameters(), lr=self.current_lr)

        for _ in range(10):
            model.train()
            optimizer.zero_grad()
            out = model(graph_data.x, graph_data.edge_index)
            loss = F.nll_loss(out[train_mask], graph_data.y[train_mask])
            loss.backward()
            optimizer.step()

        model.eval()
        with torch.no_grad():
            out = model(graph_data.x, graph_data.edge_index)
            pred = out.argmax(dim=1)
            correct = (pred[val_mask] == graph_data.y[val_mask]).sum()
            accuracy = int(correct) / int(val_mask.sum())

        return accuracy
```



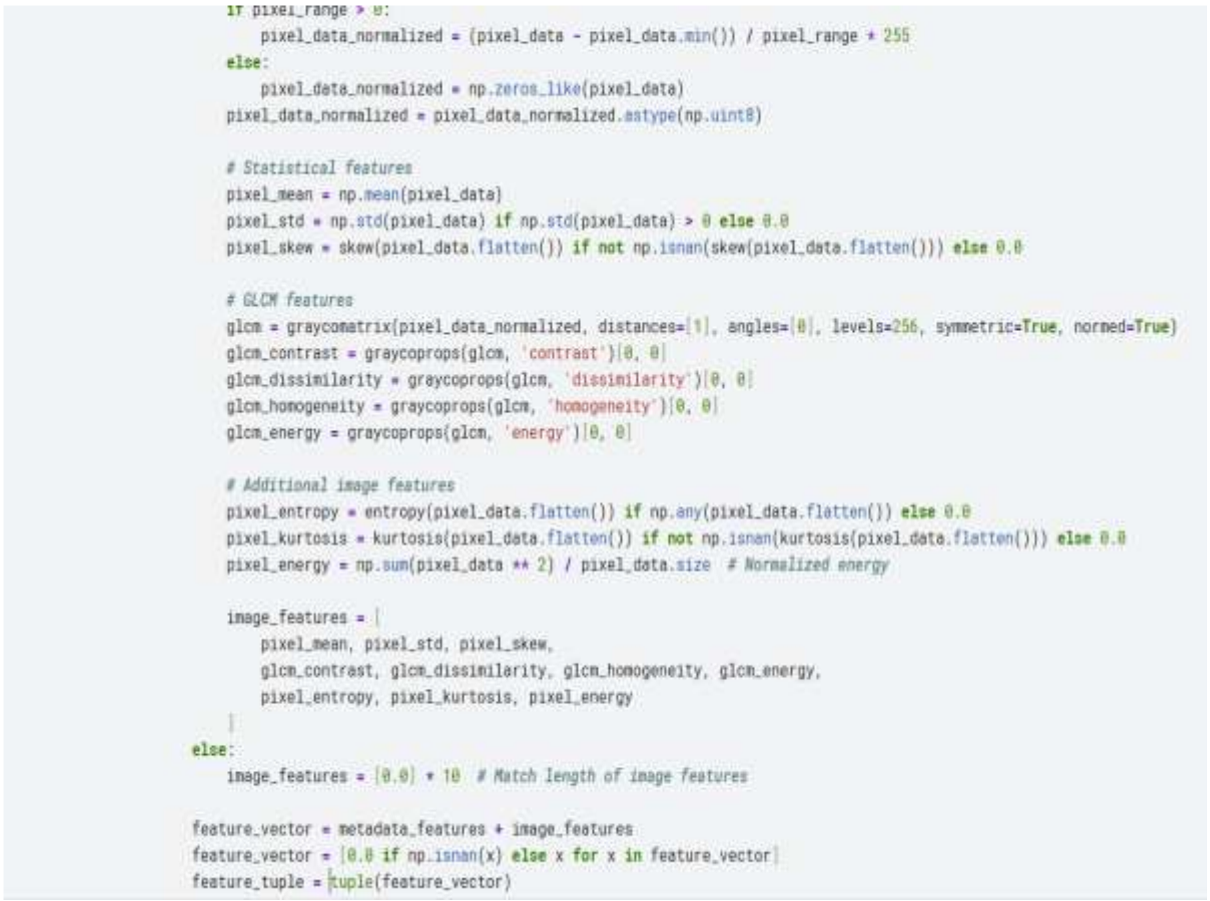
```python
def optimize(self, model_class, graph_data, max_iterations=20, device='cpu'):
    print("Starting SSO Optimization...")
    for iteration in range(max_iterations):
        c1 = 2 * np.exp(-((4 * iteration / max_iterations) ** 2))

        for i in range(self.num_salps):
            if i == 0:
                for j in range(self.num_dimensions):
                    c2 = random.random()
                    c3 = random.random()
                    if c3 < 0.5:
                        self.positions[i, j] = np.clip(
                            self.positions[i, j] + c1 * ((self.upper_bounds[j] - self.lower_bounds[j]) * c2 + self.lower_bounds[j]),
                            self.lower_bounds[j],
                            self.upper_bounds[j]
                        )
                    else:
                        self.positions[i, j] = np.clip(
                            self.positions[i, j] - c1 * ((self.upper_bounds[j] - self.lower_bounds[j]) * c2 + self.lower_bounds[j]),
                            self.lower_bounds[j],
                            self.upper_bounds[j]
                        )
            else:
                self.positions[i, :] = (self.positions[i, :] + self.positions[i-1, :]) / 2

            self.current_lr = self.positions[i, 0]
            hidden_channels = int(self.positions[i, 1])

            model = model_class(
                num_features=graph_data.x.shape[1],
                hidden_channels=hidden_channels,
                num_classes=2
            )
```



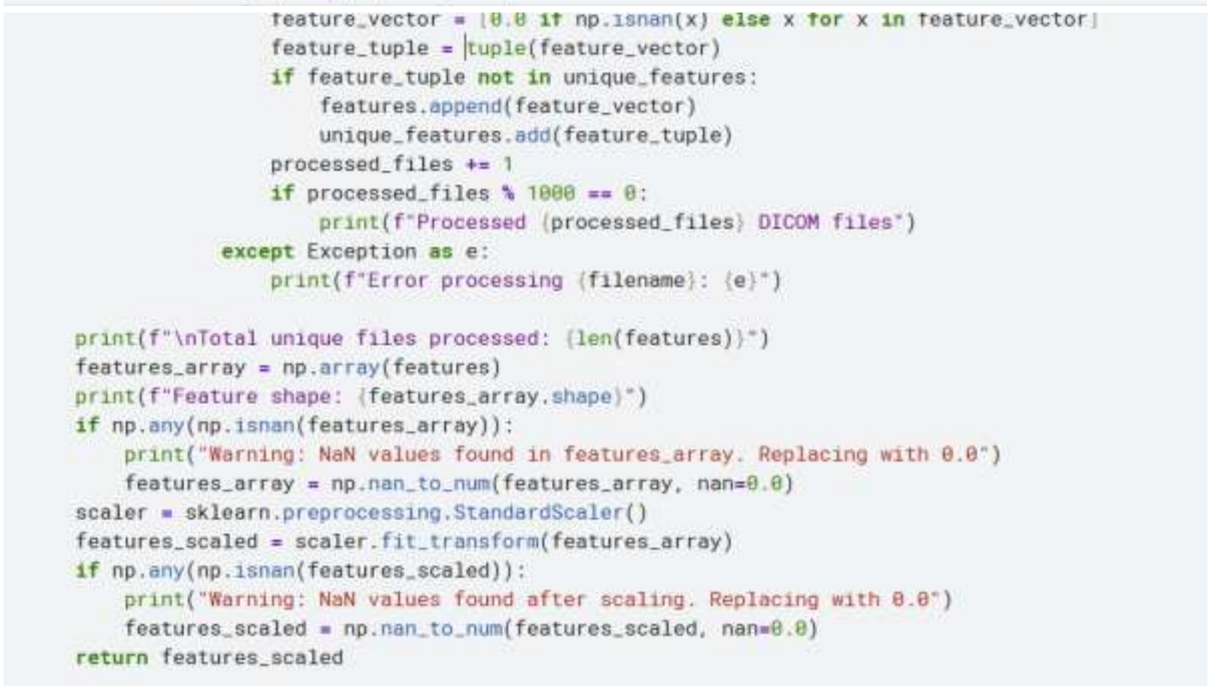
```python
def parse_dicom_age(age_str):
    try:
        match = re.match(r'(\d+)([A-Z])', str(age_str))
        if match:
            value, unit = match.groups()
            value = int(value)
            if unit == 'Y': return float(value)
            elif unit == 'M': return float(value / 12)
            elif unit == 'W': return float(value / 52)
            elif unit == 'D': return float(value / 365)
        return 0.0
    except Exception:
        return 0.0
```



```python
def extract_advanced_dicom_metadata(dicom_path):
    features = []
    processed_files = 0
    unique_features = set()

    print(f"Scanning directory: {dicom_path}")
    feature_names = [
        'PatientAge', 'PatientWeight', 'SliceThickness', 'PixelSpacingX', 'DicomAttributeCount',
        'ImageRows', 'ImageColumns', 'PixelMean', 'PixelStd', 'PixelSkew',
        'GLCM_Contrast', 'GLCM_Dissimilarity', 'GLCM_Homogeneity', 'GLCM_Energy',
        'PixelEntropy', 'PixelKurtosis', 'PixelEnergy'
    ]

    for root, _, files in os.walk(dicom_path):
        for filename in files:
            if filename.lower().endswith('.dcm'):
                full_path = os.path.join(root, filename)
                try:
                    ds = pydicom.dcmread(full_path)
                    metadata_features = [
                        parse_dicom_age(ds.get('PatientAge', '0Y')),
                        float(ds.get('PatientWeight', 0) or 0),
                        float(ds.get('SliceThickness', 0) or 0),
                        float(ds.get('PixelSpacing', [0,0])[0] if ds.get('PixelSpacing') else 0),
                        float(len(ds.dir())),
                        float(ds.get('Rows', 0) or 0),
                        float(ds.get('Columns', 0) or 0),
                    ]
                    if hasattr(ds, 'pixel_array'):
                        pixel_data = ds.pixel_array.astype(float)
                        if pixel_data.ndim > 2:
                            pixel_data = pixel_data[0] if pixel_data.shape[0] > 1 else pixel_data.mean(axis=0)
                        pixel_range = pixel_data.max() - pixel_data.min()
                        if pixel_range > 0:
```



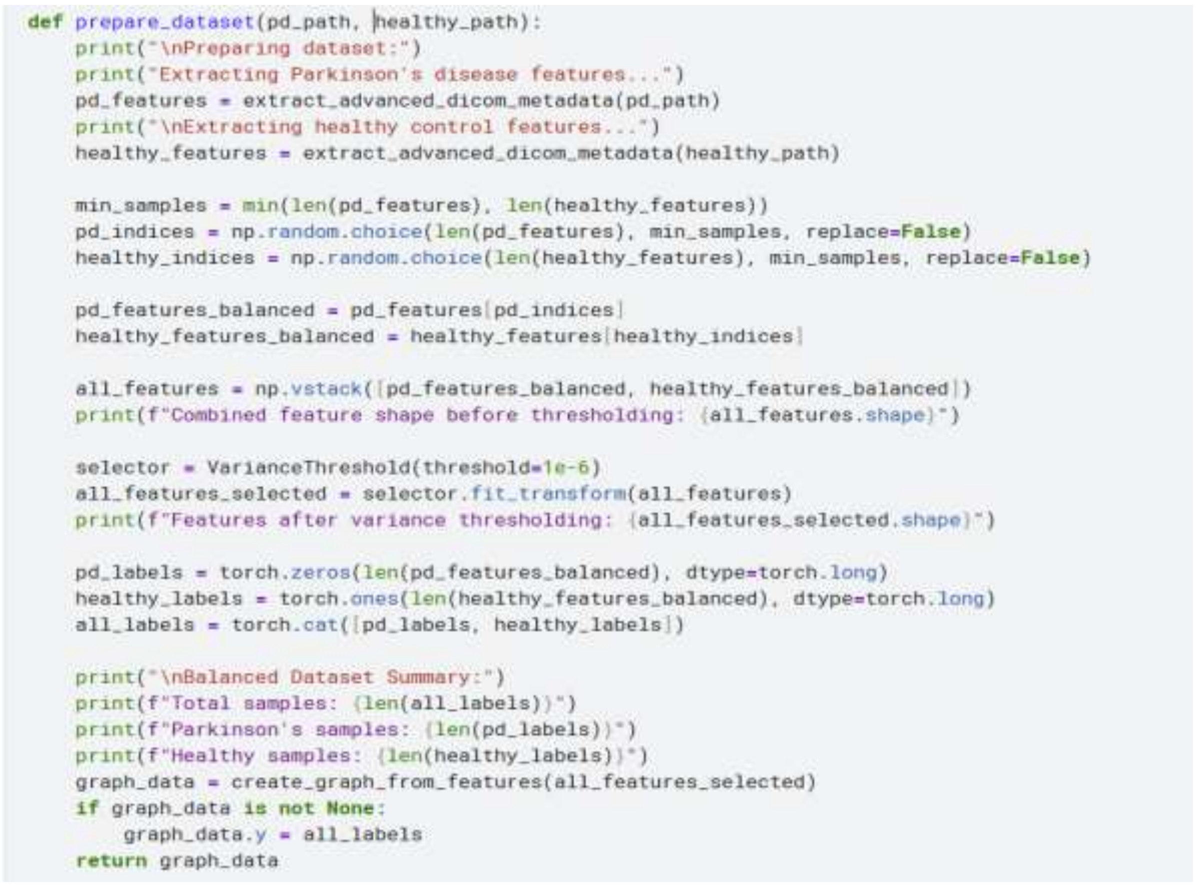
```
                        if pixel_range > 0:
                            pixel_data_normalized = (pixel_data - pixel_data.min()) / pixel_range * 255
                        else:
                            pixel_data_normalized = np.zeros_like(pixel_data)
                        pixel_data_normalized = pixel_data_normalized.astype(np.uint8)

                        # Statistical features
                        pixel_mean = np.mean(pixel_data)
                        pixel_std = np.std(pixel_data) if np.std(pixel_data) > 0 else 0.0
                        pixel_skew = skew(pixel_data.flatten()) if not np.isnan(skew(pixel_data.flatten())) else 0.0

                        # GLCM features
                        glcm = graycomatrix(pixel_data_normalized, distances=[1], angles=[0], levels=256, symmetric=True, normed=True)
                        glcm_contrast = graycoprops(glcm, 'contrast')[0, 0]
                        glcm_dissimilarity = graycoprops(glcm, 'dissimilarity')[0, 0]
                        glcm_homogeneity = graycoprops(glcm, 'homogeneity')[0, 0]
                        glcm_energy = graycoprops(glcm, 'energy')[0, 0]

                        # Additional image features
                        pixel_entropy = entropy(pixel_data.flatten()) if np.any(pixel_data.flatten()) else 0.0
                        pixel_kurtosis = kurtosis(pixel_data.flatten()) if not np.isnan(kurtosis(pixel_data.flatten())) else 0.0
                        pixel_energy = np.sum(pixel_data ** 2) / pixel_data.size  # Normalized energy

                        image_features = [
                            pixel_mean, pixel_std, pixel_skew,
                            glcm_contrast, glcm_dissimilarity, glcm_homogeneity, glcm_energy,
                            pixel_entropy, pixel_kurtosis, pixel_energy
                        ]
                    else:
                        image_features = [0.0] * 10  # Match length of image features

                    feature_vector = metadata_features + image_features
                    feature_vector = [0.0 if np.isnan(x) else x for x in feature_vector]
                    feature_tuple = tuple(feature_vector)
```



```
                    feature_vector = [0.0 if np.isnan(x) else x for x in feature_vector]
                    feature_tuple = tuple(feature_vector)
```

```
                    if feature_tuple not in unique_features:
                        features.append(feature_vector)
                        unique_features.add(feature_tuple)
                    processed_files += 1
                    if processed_files % 1000 == 0:
                        print(f"Processed {processed_files} DICOM files")
                except Exception as e:
                    print(f"Error processing {filename}: {e}")

    print(f"\nTotal unique files processed: {len(features)}")
    features_array = np.array(features)
    print(f"Feature shape: {features_array.shape}")
    if np.any(np.isnan(features_array)):
        print("Warning: NaN values found in features_array. Replacing with 0.0")
        features_array = np.nan_to_num(features_array, nan=0.0)
    scaler = sklearn.preprocessing.StandardScaler()
    features_scaled = scaler.fit_transform(features_array)
    if np.any(np.isnan(features_scaled)):
        print("Warning: NaN values found after scaling. Replacing with 0.0")
        features_scaled = np.nan_to_num(features_scaled, nan=0.0)
    return features_scaled
```

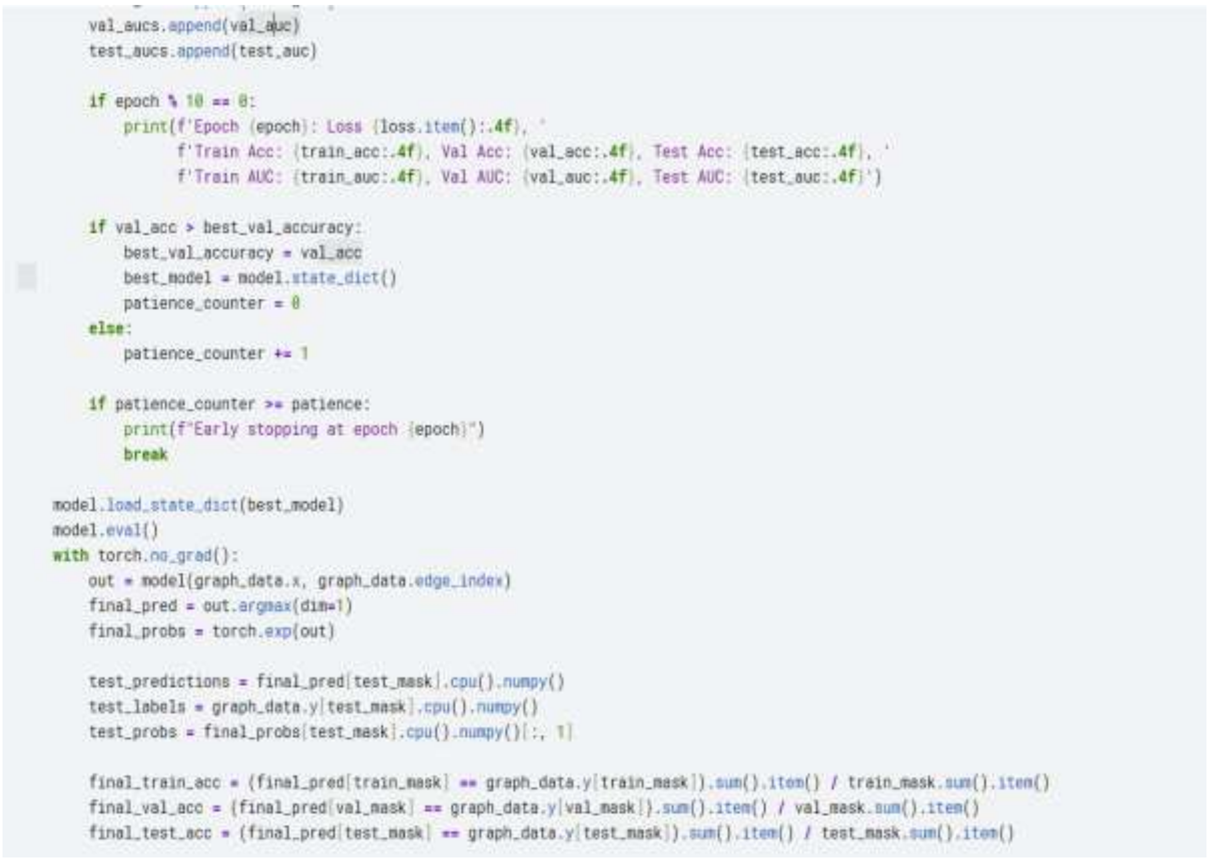
```python
def create_graph_from_features(features):
    if len(features) == 0:
        print("No features to create graph!")
        return None

    if np.any(np.isnan(features)):
        print("Warning: NaN values found in features. Replacing with 0.0")
        features = np.nan_to_num(features, nan=0.0)

    similarity_matrix = cosine_similarity(features)
    threshold = np.percentile(similarity_matrix, 99)
    adjacency_matrix = (similarity_matrix >= threshold).astype(int)

    G = nx.from_numpy_array(adjacency_matrix)
    edge_index = torch.tensor(list(G.edges)).t().contiguous()
    x = torch.tensor(features, dtype=torch.float)

    print(f"Graph - Nodes: {G.number_of_nodes()}")
    print(f"Graph - Edges: {G.number_of_edges()}")

    return Data(x=x, edge_index=edge_index)
```



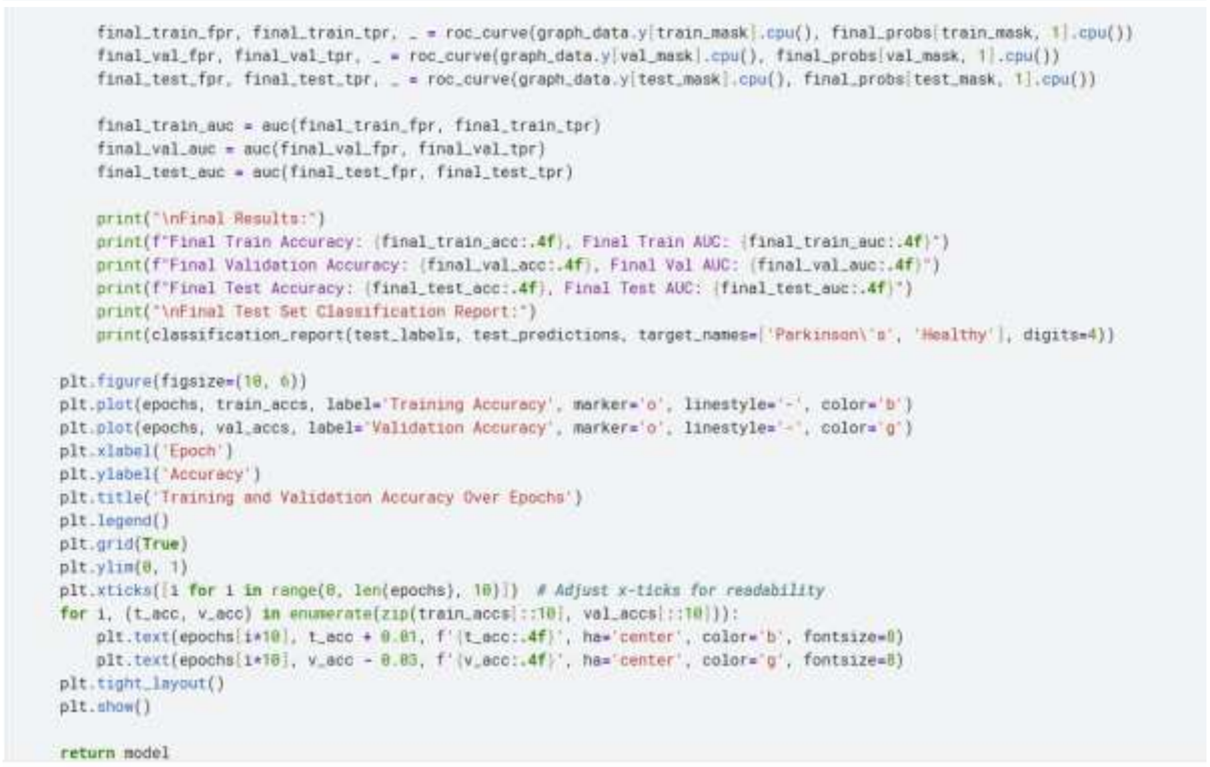
```python
def prepare_dataset(pd_path, healthy_path):
    print("\nPreparing dataset:")
    print("Extracting Parkinson's disease features...")
    pd_features = extract_advanced_dicom_metadata(pd_path)
    print("\nExtracting healthy control features...")
    healthy_features = extract_advanced_dicom_metadata(healthy_path)

    min_samples = min(len(pd_features), len(healthy_features))
    pd_indices = np.random.choice(len(pd_features), min_samples, replace=False)
    healthy_indices = np.random.choice(len(healthy_features), min_samples, replace=False)

    pd_features_balanced = pd_features[pd_indices]
    healthy_features_balanced = healthy_features[healthy_indices]

    all_features = np.vstack([pd_features_balanced, healthy_features_balanced])
    print(f"Combined feature shape before thresholding: {all_features.shape}")

    selector = VarianceThreshold(threshold=1e-6)
    all_features_selected = selector.fit_transform(all_features)
    print(f"Features after variance thresholding: {all_features_selected.shape}")

    pd_labels = torch.zeros(len(pd_features_balanced), dtype=torch.long)
    healthy_labels = torch.ones(len(healthy_features_balanced), dtype=torch.long)
    all_labels = torch.cat([pd_labels, healthy_labels])

    print("\nBalanced Dataset Summary:")
    print(f"Total samples: {len(all_labels)}")
    print(f"Parkinson's samples: {len(pd_labels)}")
    print(f"Healthy samples: {len(healthy_labels)}")
    graph_data = create_graph_from_features(all_features_selected)
    if graph_data is not None:
        graph_data.y = all_labels
    return graph_data
```

```
def optimized_train_gcn(graph_data, best_params, device='cpu'):
    best_lr, best_hidden_channels = best_params
    model = ParkinsonGCN(
        num_features=graph_data.x.shape[1],
        hidden_channels=int(best_hidden_channels),
        num_classes=2
    ).to(device)
    graph_data = graph_data.to(device)
    optimizer = torch.optim.Adam(model.parameters(), lr=best_lr)

    num_nodes = graph_data.num_nodes
    perm = torch.randperm(num_nodes)
    train_size = int(0.6 * num_nodes)
    val_size = int(0.2 * num_nodes)

    train_idx = perm[:train_size]
    val_idx = perm[train_size:train_size + val_size]
    test_idx = perm[train_size + val_size:]

    train_mask = torch.zeros(num_nodes, dtype=torch.bool)
    val_mask = torch.zeros(num_nodes, dtype=torch.bool)
    test_mask = torch.zeros(num_nodes, dtype=torch.bool)

    train_mask[train_idx] = True
    val_mask[val_idx] = True
    test_mask[test_idx] = True

    best_val_accuracy = 0
    patience = 20
    patience_counter = 0
    best_model = None

    train_losses, train_accs, val_accs, test_accs = [], [], [], []
    train_aucs, val_aucs, test_aucs = [], [], []
```

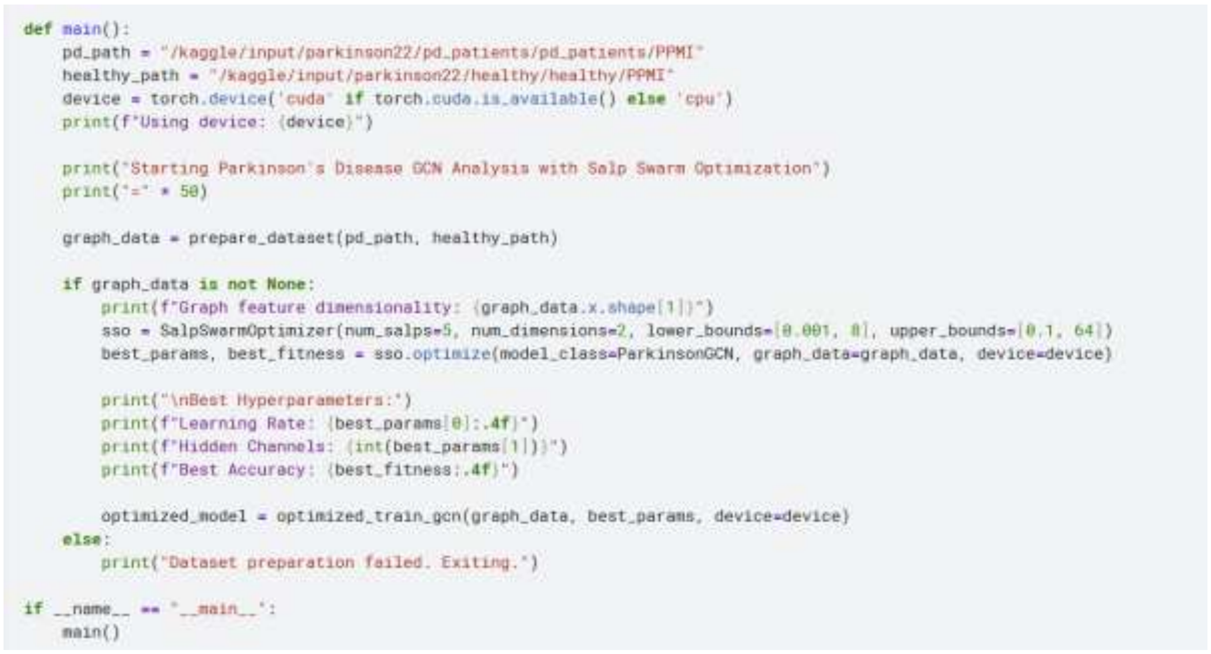
```
    print("\nTraining Optimized GCN:")
    for epoch in range(100):
        model.train()
        optimizer.zero_grad()
        out = model(graph_data.x, graph_data.edge_index)
        loss = F.nll_loss(out[train_mask], graph_data.y[train_mask])
        loss.backward()
        optimizer.step()

        model.eval()
        with torch.no_grad():
            pred = out.argmax(dim=1)
            probs = torch.exp(out)[:, 1]

            train_acc = (pred[train_mask] == graph_data.y[train_mask]).sum().item() / train_mask.sum().item()
            val_acc = (pred[val_mask] == graph_data.y[val_mask]).sum().item() / val_mask.sum().item()
            test_acc = (pred[test_mask] == graph_data.y[test_mask]).sum().item() / test_mask.sum().item()

            train_fpr, train_tpr, _ = roc_curve(graph_data.y[train_mask].cpu(), probs[train_mask].cpu())
            val_fpr, val_tpr, _ = roc_curve(graph_data.y[val_mask].cpu(), probs[val_mask].cpu())
            test_fpr, test_tpr, _ = roc_curve(graph_data.y[test_mask].cpu(), probs[test_mask].cpu())

            train_auc = auc(train_fpr, train_tpr)
            val_auc = auc(val_fpr, val_tpr)
            test_auc = auc(test_fpr, test_tpr)

        epochs.append(epoch)
        train_losses.append(loss.item())
        train_accs.append(train_acc)
        val_accs.append(val_acc)
        test_accs.append(test_acc)
        train_aucs.append(train_auc)
        val_aucs.append(val_auc)
        test_aucs.append(test_auc)
```



```
    val_aucs.append(val_auc)
    test_aucs.append(test_auc)

    if epoch % 10 == 0:
        print(f'Epoch {epoch}: Loss {loss.item():.4f}, '
              f'Train Acc: {train_acc:.4f}, Val Acc: {val_acc:.4f}, Test Acc: {test_acc:.4f}, '
              f'Train AUC: {train_auc:.4f}, Val AUC: {val_auc:.4f}, Test AUC: {test_auc:.4f}')

    if val_acc > best_val_accuracy:
        best_val_accuracy = val_acc
        best_model = model.state_dict()
        patience_counter = 0
    else:
        patience_counter += 1

    if patience_counter >= patience:
        print(f"Early stopping at epoch {epoch}")
        break

model.load_state_dict(best_model)
model.eval()
with torch.no_grad():
    out = model(graph_data.x, graph_data.edge_index)
    final_pred = out.argmax(dim=1)
    final_probs = torch.exp(out)

    test_predictions = final_pred[test_mask].cpu().numpy()
    test_labels = graph_data.y[test_mask].cpu().numpy()
    test_probs = final_probs[test_mask].cpu().numpy()[:, 1]

    final_train_acc = (final_pred[train_mask] == graph_data.y[train_mask]).sum().item() / train_mask.sum().item()
    final_val_acc = (final_pred[val_mask] == graph_data.y[val_mask]).sum().item() / val_mask.sum().item()
    final_test_acc = (final_pred[test_mask] == graph_data.y[test_mask]).sum().item() / test_mask.sum().item()
```



```
    final_train_fpr, final_train_tpr, _ = roc_curve(graph_data.y[train_mask].cpu(), final_probs[train_mask, 1].cpu())
    final_val_fpr, final_val_tpr, _ = roc_curve(graph_data.y[val_mask].cpu(), final_probs[val_mask, 1].cpu())
    final_test_fpr, final_test_tpr, _ = roc_curve(graph_data.y[test_mask].cpu(), final_probs[test_mask, 1].cpu())

    final_train_auc = auc(final_train_fpr, final_train_tpr)
    final_val_auc = auc(final_val_fpr, final_val_tpr)
    final_test_auc = auc(final_test_fpr, final_test_tpr)

    print("\nFinal Results:")
    print(f"Final Train Accuracy: {final_train_acc:.4f}, Final Train AUC: {final_train_auc:.4f}")
    print(f"Final Validation Accuracy: {final_val_acc:.4f}, Final Val AUC: {final_val_auc:.4f}")
    print(f"Final Test Accuracy: {final_test_acc:.4f}, Final Test AUC: {final_test_auc:.4f}")
    print("\nFinal Test Set Classification Report:")
    print(classification_report(test_labels, test_predictions, target_names=['Parkinson\'s', 'Healthy'], digits=4))

plt.figure(figsize=(10, 6))
plt.plot(epochs, train_accs, label='Training Accuracy', marker='o', linestyle='-', color='b')
plt.plot(epochs, val_accs, label='Validation Accuracy', marker='o', linestyle='-', color='g')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Training and Validation Accuracy Over Epochs')
plt.legend()
plt.grid(True)
plt.ylim(0, 1)
plt.xticks([i for i in range(0, len(epochs), 10)])  # Adjust x-ticks for readability
for i, (t_acc, v_acc) in enumerate(zip(train_accs[::10], val_accs[::10])):
    plt.text(epochs[i*10], t_acc + 0.01, f'{t_acc:.4f}', ha='center', color='b', fontsize=8)
    plt.text(epochs[i*10], v_acc - 0.03, f'{v_acc:.4f}', ha='center', color='g', fontsize=8)
plt.tight_layout()
plt.show()

return model
```



```python
def main():
    pd_path = "/kaggle/input/parkinson22/pd_patients/pd_patients/PPMI"
    healthy_path = "/kaggle/input/parkinson22/healthy/healthy/PPMI"
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    print(f"Using device: {device}")

    print("Starting Parkinson's Disease GCN Analysis with Salp Swarm Optimization")
    print("=" * 50)

    graph_data = prepare_dataset(pd_path, healthy_path)

    if graph_data is not None:
        print(f"Graph feature dimensionality: {graph_data.x.shape[1]}")
        sso = SalpSwarmOptimizer(num_salps=5, num_dimensions=2, lower_bounds=[0.001, 8], upper_bounds=[0.1, 64])
        best_params, best_fitness = sso.optimize(model_class=ParkinsonGCN, graph_data=graph_data, device=device)

        print("\nBest Hyperparameters:")
        print(f"Learning Rate: {best_params[0]:.4f}")
        print(f"Hidden Channels: {int(best_params[1])}")
        print(f"Best Accuracy: {best_fitness:.4f}")

        optimized_model = optimized_train_gcn(graph_data, best_params, device=device)
    else:
        print("Dataset preparation failed. Exiting.")

if __name__ == "__main__":
    main()
```

Bibliography

[1] I. Leroi, H. Pantula, K. McDonald, and V. Harbishettar, “Neuropsychiatric symptoms in parkinsons disease with mild cognitive impairment and dementia,” *Parkinsons Dis*, 2012, doi: 10.1155/2012/308097.

[2] T. Bhattacharya, K. T. Thomas, and L. Mathew, “Parkinsons Disease Progression Prediction using Advanced Machine Learning Techniques,” in *2024 International Conference on Electrical, Electronics and Computing Technologies, ICEECT 2024*, Institute of Electrical and Electronics Engineers Inc., 2024. doi: 10.1109/ICEECT61758.2024.10739044.

[3] W. Zhang, M. Xu, Y. Feng, Z. Mao, and Z. Yan, “The Effect of Procrastination on Physical Exercise among College Students—The Chain Effect of Exercise Commitment and Action Control,” *International Journal of Mental Health Promotion*, vol. 26, no. 8, pp. 611–622, 2024, doi: 10.32604/ijmhp.2024.052730.

[4] V. Majhi, S. Paul, and G. Saha, “Impact of Surgical History on Parkinson’s Disease Progression,” in *2022 IEEE Delhi Section Conference, DELCON 2022*, Institute of Electrical and Electronics Engineers Inc., 2022. doi: 10.1109/DELCON54057.2022.9753658.

[5] S. Das, G. Lavanya, S. J. Prakash, N. S. Dey, J. Panuganti, and R. Poojitha, “Lung Cancer Detection and Classification using Transfer Learning with Pre-trained VGG19 Convolutional Neural Networks,” in *2023 3rd International Conference on Emerging Frontiers in Electrical and Electronic Technologies, ICEFEET 2023*, Institute of Electrical and Electronics Engineers Inc., 2023. doi: 10.1109/ICEFEET59656.2023.10452195.

[6] K. S. Gill, V. Anand, and R. Gupta, “Pneumonia Disease Classification Utilizing Artificial Intelligence on VGG19 Model using Chest Xray Images,” in *2023 Global Conference on Information Technologies and Communications, GCITC 2023*, Institute of Electrical and Electronics Engineers Inc., 2023. doi: 10.1109/GCITC60406.2023.10426395.

[7] S. Vats, J. Anand, V. Kukreja, and R. Sharma, “Deep Learning-based VGG16, VGG19, and ResNet Models for Grapevine Disease Classification,” in *2024 IEEE 9th International Conference for Convergence in Technology, I2CT 2024*, Institute of Electrical and Electronics Engineers Inc., 2024. doi: 10.1109/I2CT61223.2024.10544168.

[8] A. Kaur, “Revolutionizing Leukemia Diagnosis: Fine-Tuned VGG19 CNN for Enhanced Classification Accuracy,” in *2024 Global Conference on Communications and Information Technologies, GCCIT 2024*, Institute of Electrical and Electronics Engineers Inc., 2024. doi: 10.1109/GCCIT63234.2024.10862090.

[9] G. Harish Kumar and K. Jaisharma, “Enhancing the Handwritten Digit Recognition by Employing Novel Progressive VGG19 Model and Compare with SVM Performance,” in *2024 15th International Conference on Computing Communication and Networking Technologies, ICCCNT 2024*, Institute of Electrical and Electronics Engineers Inc., 2024. doi: 10.1109/ICCCNT61001.2024.10724278.

[10] L. Abualigah, M. Shehab, M. Alshinwan, and H. Alabool, “Salp swarm algorithm: a comprehensive survey,” Aug. 01, 2020, *Springer*. doi: 10.1007/s00521-019-04629-4.

[11] A. E. Hegazy, M. A. Makhlouf, and G. S. El-Tawel, “Improved salp swarm algorithm for feature selection,” *Journal of King Saud University - Computer and Information Sciences*, vol. 32, no. 3, pp. 335–344, Mar. 2020, doi: 10.1016/j.jksuci.2018.06.003.

[12] H. Faris, S. Mirjalili, I. Aljarah, M. Mafarja, and A. A. Heidari, “Salp swarm algorithm: Theory, literature review, and application in extreme learning machines,” in *Studies in Computational Intelligence*, vol. 811, Springer Verlag, 2020, pp. 185–199. doi: 10.1007/978-3-030-12127-3_11.

[13] S. Mirjalili, A. H. Gandomi, S. Z. Mirjalili, S. Saremi, H. Faris, and S. M. Mirjalili, “Salp Swarm Algorithm: A bio-inspired optimizer for engineering design problems,” *Advances in Engineering Software*, vol. 114, pp. 163–191, Dec. 2017, doi: 10.1016/j.advengsoft.2017.07.002.

[14] X. Zhang, L. He, K. Chen, Y. Luo, J. Zhou, and F. Wang, “Multi-View Graph Convolutional Network and Its Applications on Neuroimage Analysis for Parkinson’s Disease.” [Online]. Available: https://www.niehs.nih.gov/research/supported/health/neurodegenerative/index.cfm

[15] T. Nguyen, G. T. T. Nguyen, T. Nguyen, and D.-H. Le, “Graph convolutional networks for drug response prediction,” Apr. 09, 2020. doi: 10.1101/2020.04.07.030908.

[16] L. Yao, C. Mao, and Y. Luo, “Graph Convolutional Networks for Text Classification.” [Online]. Available: www.aaai.org

[17] S. Parisot *et al.*, “Disease Prediction using Graph Convolutional Networks: Application to Autism Spectrum Disorder and Alzheimer’s Disease,” Jun. 2018, doi: 10.1016/j.media.2018.06.001.

[18] D. Arya *et al.*, “Fusing Structural and Functional MRIs using Graph Convolutional Networks for Autism Classification GCN for Autism Classification,” 2020.

[19] K. Ding *et al.*, “Graph Convolutional Networks for Multi-modality Medical Imaging: Methods, Architectures, and Clinical Applications,” Feb. 2022, [Online]. Available: http://arxiv.org/abs/2202.08916

[20] Y. Suzuki and K. Ichige, “High Accuracy Video Foreground Segmentation Based on Feature Normalization,” in *Proceedings of ISCIT 2021: 2021 20th International Symposium on Communications and Information Technologies: Quest for Quality of Life and Smart City*, Institute of Electrical and Electronics Engineers Inc., Oct. 2021, pp. 15–17. doi: 10.1109/ISCIT52804.2021.9590583.

[21] Z. Peng, L. Wang, Q. Liang, and C. Chen, “Frequency-Dependent Scaling Factors to Estimate Multiple Time-period Global Mass Changes Observed by GRACE-GRACE-FO,” *IEEE J Sel Top Appl Earth Obs Remote Sens*, 2025, doi: 10.1109/JSTARS.2025.3546524.

[22] E. Jain, K. S. Gill, S. Malhotra, and S. Devliyal, “Improving Ovarian Cancer Diagnosis with Deep Learning by Identification of Subtype Using VGG19 CNN Pre-Trained Model,” in *2024 IEEE International Conference on Smart Power Control and Renewable Energy, ICSPCRE 2024*, Institute of Electrical and Electronics Engineers Inc., 2024. doi: 10.1109/ICSPCRE62303.2024.10674939.

[23] K. Mittal, K. S. Gill, D. Upadhyay, and S. Devliyal, “Trailblazing Strategies for Solar Panel Maintenance: Employing VGG19 for Early Detection of Damage,” in *2024 International Conference on Intelligent Systems for Cybersecurity, ISCS 2024*, Institute of Electrical and Electronics Engineers Inc., 2024. doi: 10.1109/ISCS61804.2024.10581079.

[24] W. Wang, C. Liu, G. Liu, and X. Wang, “CF-GCN: Graph Convolutional Network for Change Detection in Remote Sensing Images,” *IEEE Transactions on Geoscience and Remote Sensing*, vol. 62, pp. 1–13, 2024, doi: 10.1109/TGRS.2024.3357085.

[25] F. Liu, X. Qian, L. Jiao, X. Zhang, L. Li, and Y. Cui, “Contrastive Learning-Based Dual Dynamic GCN for SAR Image Scene Classification,” *IEEE Trans Neural Netw Learn Syst*, vol. 35, no. 1, pp. 390–404, Jan. 2024, doi: 10.1109/TNNLS.2022.3174873.

[26] G. Gu and Y. Zhang, “Design and implementation of cookie-based CD-SSO,” in *Proceedings - 2013 International Conference on Computational and Information Sciences, ICCIS 2013*, 2013, pp. 1785–1788. doi: 10.1109/ICCIS.2013.467.

[27] "Application_of_Hybrid_Salp_Swarm_optimization_Method_for_Solving_OPF_Problem".

[28] D. Luo *et al.*, "Prediction for Dissolved Gas in Power Transformer Oil Based On TCN and GCN," *IEEE Trans Ind Appl*, vol. 58, no. 6, pp. 7818–7826, 2022, doi: 10.1109/TIA.2022.3197565.

[29] T. Vaiyapuri, E. L. Lydia, M. Y. Sikkandar, V. G. Diaz, I. V. Pustokhina, and D. A. Pustokhin, "Internet of Things and Deep Learning Enabled Elderly Fall Detection Model for Smart Homecare," *IEEE Access*, vol. 9, pp. 113879–113888, 2021, doi: 10.1109/ACCESS.2021.3094243.

[30] Parkinson's Progression Markers Initiative, "Parkinson's Progression Markers Initiative (PPMI) Dataset," The Michael J. Fox Foundation for Parkinson's Research, 2025, [Online]. Available: https://www.ppmi-info.org