Software Application

for

Artificial Intelligence Powered Seed Identification

Final Report

By

Matthew Morrow

Florida Institute of Technology

CIS 5898 – Projects in Computer Information Systems

for

Dr. Bernard Parenteau

December 6, 2024

**Introduction**

The Seed Identifier project, named **SeedID**, is an advanced AI-driven application designed to revolutionize seed identification using cutting-edge image recognition technology. By leveraging artificial intelligence, SeedID addresses a significant bottleneck in agriculture, conservation, and research: the precise identification of seeds. Traditionally, this task has required highly specialized experts or labor-intensive reference work, which can be time-consuming and prone to human error. SeedID simplifies and accelerates this process, offering users a streamlined solution to upload seed images and receive accurate identification labels alongside weighted confidence scores.

At its core, SeedID combines powerful machine learning algorithms with a user-friendly interface, making it accessible to both experts and non-experts. Users can upload seed images directly from their local devices through an intuitive graphical user interface (GUI), and the application processes these images to deliver results almost instantaneously. Whether identifying seeds for biodiversity monitoring, crop management, or quality assurance in seed distribution systems, SeedID ensures consistent and reliable outcomes.

A key feature of SeedID is its integration with Microsoft Azure, which serves as a central hub for storing images and hosting backups of the machine learning model. This cloud-based infrastructure enables two major benefits:

1. **Remote Accessibility**: Users can access SeedID from multiple devices without the need for local model storage, making it a flexible tool for distributed teams.
2. **Efficient Resource Management**: Machine learning models can be trained locally for users with powerful hardware or pulled directly from the cloud, alleviating the computational burden for devices with limited processing capacity.

SeedID sits at the intersection of technology and the intricate world of organic biology, bridging the gap between advanced AI techniques and practical applications in agriculture and ecology. By offering rapid, accurate seed identification, it empowers stakeholders across multiple sectors to make better decisions with reduced time and effort. The scalability of this AI-powered solution enhances workflows by reducing human error, improving classification accuracy, and fostering data-driven decision-making.

In summary, **SeedID** transforms seed identification into a rapid, reliable, and accessible process, setting a new standard for how technology can support agriculture, conservation, and biological research. With its robust AI capabilities and cloud integration, SeedID is poised to become an indispensable tool in industries that depend on precise and efficient image classification.

**Algorithms/Project Solution**

Leveraging Python and the PyTorch framework, the following has been implemented using the structured approach found in the SDLC process. The listed functions are built into the framework and sourced from the information found in the PyTorch official documentation. They, in addition to supplemental custom helper functions, offer ease of use for preparing and running SeedID.

1. **Image Preprocessing**. This includes the handling of image data and formatting the image to be a in a palatable setting for the recognition engine to process. The functions normalize\_image(image) and resize\_image(image, sizing) were planned to handle tasks such as pixel normalization and resizing before converting images to tensors. However, a function called predict\_image() to handle the preprocessing steps using PyTorch’s transforms pipeline:

**Functions**:

load\_image(path: string): Load image from a file path.

**Data Input Values:**

Image files – .jpg, jpeg, and/or .png.

**Tensoring**: After resizing, the image needs to be converted to a tensor format. As seen above, PyTorch provides utilities like transforms.ToTensor() that convert an image to a tensor and automatically scales pixel values from [0, 255] to [0, 1] by dividing by 255.

1. **Model Definition Module**. Defining the architecture of the image recognition neural network.

**Functions**:

build\_model(): Sets up the neural network layers using which are inherited from the pre-trained ResNet50 model. The convolutional and pooling layers are frozen initially to preserve their pre-trained weights, which are optimized for feature extraction.

load\_model(): Loads a pre-trained or previously saved model, restoring its architecture and weights for inference or further training. Uses the path temp/trained\_model.pth to store and pull the model.

save\_model(): Saves the model's architecture and trained weights for future use.

**Data Input Values:**

Architecture params - number of layers and activation.

**Output**: PyTorch model object.

1. **Training**. Setting up the training for the model.

**Functions**:

train\_model(train\_loader, val\_loader, epoch\_num, learning\_rate): Trains the model over a specified number of epochs, adjusting weights based on loss.

evaluate(val\_loader): Evaluates the model on the validation dataset during training to monitor performance.

run\_training\_process(): Combines all training related steps, including data loading, model training, and temporary directory management for Azure images.

adjust\_learning\_rate(epoch, initial\_lr, step\_size, gamma): Dynamically adjusts the learning rate during training for better convergence.

load\_data(data\_dir, batch\_size=32): Prepares the training and validation data loaders with appropriate transformations.

**Data Input Values:**

Training the model to receive labels

**Output**:

Having a trained model to use in the core implementation.

1. **Prediction and Inteference**. The application will have a method of displaying prediction confidence. This should be displayed in addition to the user selected image. Providing the user with the information needed to determine if the accuracy of the application is sufficient.

**Functions**:

display\_results(pred\_class, confidence): Displays the prediction results, including the predicted class name and confidence score, in the GUI.

upload\_image(): Allows the user to select an image file and initiates the prediction process.

**Data Input Values:**

Provided user images.

**Output**:

Prediction labels and confidence scores.

1. **Model Saving and Loading**. After a model is trained there needs to be functionality to save and load it into the application for running the comparisons. This will require the following two functions:

**Functions**:

save\_model(path): Saves the trained model's state dictionary to a file.

load\_model(path): Loads the previously saved model for inference or further training.

save\_model\_to\_azure(model\_path, blob\_name="trained\_model.pth"): Saves the trained model to Azure Blob Storage for remote access and redundancy.

download\_model\_from\_azure(model\_path, blob\_name="trained\_model.pth"): Downloads a previously saved model from Azure Blob Storage if it is not available locally.

**Data Input Values:**

Model object and the file path.

**Output**:

The model file and the loaded/trained model.

1. **Graphical User Interface (GUI)**. An interface is required for the user to be able to interact with the application. Such actions include training the model, uploading images from local directories to for identification, and reviewing prediction scores and results.

**Functions**:

upload\_image(): Allows the user to upload an image for recognition.

train\_model\_gui(): Provides a GUI option to train the model and display progress.

show\_azure\_directory(): Displays the contents of the Azure Blob Storage in the GUI.

clear\_azure\_database\_threaded(): Provides a GUI option to clear all images from Azure Blob Storage.

show\_loading(message)/hide\_loading(message): Displays or hides a loading message in the GUI during tasks like training or syncing with Azure.

**Data Input Values:**

User input images sourced from local repositories.

**Output**: Prediction results and confidence scores.

1. **Error Handling and Logging.** When errors or user actions result in a log worthy event the messages should be stored in a local file for triage and in some instances displayed to the user via a prompt on the UI.

**Component:**

log\_event(event: str): Logs key events such as model training start/stop, errors, or other actions.

**Functions**:

handle\_errors(): Catches and handles any errors that occur during runtime, preventing the system from crashing.

show\_loading(message)/hide\_loading(message): Displays or hides a loading message in the GUI during tasks like training or syncing with Azure.

**Data Input Values:**

Events and user actions

**Output**:

File logging and user visible error messages.

**Database Description**

To be able to host the application online in the future and to enable users from other devices to access the data and trained models, a cloud-based database is required. A workflow with Azure ensures that the image recognition system has a scalable and organized method for managing both image data and associated metadata. Microsoft’s Azure offers solutions that store the image as a long string known as a Blob. The Azure SQL Database with Blob Storage begins by uploading the image files to Azure Blob Storage. Each image is assigned a unique URL or path, which will be used during the model's training and inference stages.

Next, a table is created in Azure SQL Database to store relevant metadata about each image. This table includes columns for the image ID, the URL linking to the image in Blob Storage, labels or classifications, predictions, and timestamps for when the images were uploaded. This metadata will provide easy access to both the image location and important details about the image's content and model output. Once the images are stored in Blob Storage and the metadata is organized in SQL Database, a connection between the two will be established. The SQL Database will serve as the central hub for querying image information, and the stored URLs will be used to retrieve and load the images into the AI system using PyTorch for tasks such as training and prediction.

**Implementation**

The programming language selected for handling the application logic is Python 3.12.4, chosen for its extensive libraries, flexibility, and performance in AI and machine learning tasks. Python’s Pytorch was a critical resource for the implementation and model training of SeedID. The Python code is developed using the PyCharm Community Edition IDE, which seamlessly supports Python development. For database management, the cloud-based solutions provided by Microsoft Azure will serve as the primary data storage platform. Widely supported by the previously mentioned tools, Windows 11, is the chosen OS platform to host the project’s development and presentation.

On the hardware end, the image learning processing will be run by the latest in personal computing technology- a 13th Gen Intel Core i9-13980HX CPU and a NVIDIA GeForce RTX 4070 GPU. With 64GB of localized memory, the processing and training is relatively quick and fully capable of ramping up the pixelation, weights, and epochs required for effective and more accurate model training.

The combination of Python’s ease of use, Azure’s cloud solutions, and the cutting-edge hardware allows for efficient processing, training, and deployment of SeedID. This setup ensures the program’s overall scalability, speed, and accuracy in identifying seeds based on images provided by users or external datasets.

The Seed Data

The primary dataset for this project is obtained from publicly available online sources that provide stock footage of simple, easily identifiable seeds such as wheat grains, corn kernels, and almonds. These readily accessible images will serve as the foundation for early-stage model development.

**A yellow corn kernel on a white background

Description automatically generated**Examples of Online Sourced Seed Data

A close up of a nut

Description automatically generatedA close up of a grain

Description automatically generated

*Corn. Image 1*

*Wheat. Image 2*

*Almond. Image 3*

**Graphical User Interface**

A screenshot of a computer

Description automatically generatedThe following outline provides an overview of the original design of the GUI layout for the application. The top bar initial features the tool’s name along with window control options, including the close button. A panel on the left-hand side will contain the user interface buttons, allowing for easy navigation and interaction. The central panel will display the user-selected image, accompanied by the predicted confidence score and the label indicating the identified seed's name.

A screenshot of a computer

Description automatically generatedThe final design matches very similarly to the initial layout, functionality, and color scheme. With one significant alteration, the inclusion of a settings tab.

The settings tab provides functionality for managing application settings and interacting with Azure Blob Storage. It allows users to perform administrative tasks related to the model and data, offering options to sync, view, or clear stored data.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated**Schedule (Gantt Chart)**

The chart breaks down the project into specific, time-bound phases, such as submitting the project proposal, setting up local computation and GPU resources, configuring the Azure database, collecting and processing seed images, developing the core image recognition application, and preparing the final submission. The timeline spans from mid-September to late December, with tasks like the project proposal expected to finish early, while more complex tasks such as developing the core application span a longer period.

Gray bars represent the planned duration of each task, while blue bars indicate active or ongoing work. The chart also includes dependency markers, showing that some tasks rely on the completion of others. For example, setting up the database is a prerequisite for collecting and processing the seed dataset.

**Results**

**References**

Chollet, F. (2018). Deep learning with Python. Manning Publications.

FreePNGImg. (n.d.). Almond PNG image. FreePNGImg. https://freepngimg.com/png/9490-almond-png-image

GeeksForGeeks. (n.d.). Creating Tabbed Widget With Python-Tkinter. Retrieved December 3, 2024, from https://www.geeksforgeeks.org/creating-tabbed-widget-with-python-tkinter/

GeeksForGeeks. (n.d.). Delete an entire directory tree using Python. Retrieved December 3, 2024, from https://www.geeksforgeeks.org/delete-an-entire-directory-tree-using-python-shutil-rmtree-method/

GeeksForGeeks. (n.d.). Loading images in Tkinter using PIL. Retrieved December 3, 2024, from https://www.geeksforgeeks.org/loading-images-in-tkinter-using-pil/

GeeksForGeeks. (n.d.). Residual Networks (ResNet) – Deep Learning. Retrieved December 3, 2024, from https://www.geeksforgeeks.org/residual-networks-resnet-deep-learning/

GeeksForGeeks. (n.d.). Selenium Python tutorial. Retrieved December 3, 2024, from https://www.geeksforgeeks.org/selenium-python-tutorial/

Gantt Chart. (n.d.). TeamGantt. https://www.teamgantt.com

iStock. (n.d.). Corn seeds [Photographs]. iStock. https://www.istockphoto.com/photos/corn-seeds

Microsoft. (2024, September 13). Quickstart: Azure Blob Storage client library for Python. Retrieved December 3, 2024, from https://learn.microsoft.com/en-us/azure/storage/blobs/storage-quickstart-blobs-python

Microsoft. (n.d.). Download a blob with Python. Retrieved December 3, 2024, from https://learn.microsoft.com/en-us/azure/storage/blobs/storage-blob-download-python

Microsoft. (n.d.). Upload a block blob with Python. Retrieved December 3, 2024, from https://learn.microsoft.com/en-us/azure/storage/blobs/storage-blob-upload-python

PNG Hunter. (n.d.). Wheat. PNG Hunter. https://pnghunter.com/png/wheat-49/

PyTorch. (n.d.). PyTorch documentation. PyTorch. Retrieved September 11, 2024, from https://pytorch.org/docs/

PyTorch Documentation. (n.d.). torch.nn.CrossEntropyLoss. Retrieved December 3, 2024, from https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html

PyTorch Documentation. (n.d.). torch.save. Retrieved December 3, 2024, from https://pytorch.org/docs/stable/generated/torch.save.html

Real Python. (n.d.). Python's unittest: Writing unit tests for your code. Retrieved December 3, 2024, from https://realpython.com/python-unittest/

Stack Overflow. (n.d.). Thread not working but \_thread does. Retrieved December 3, 2024, from https://stackoverflow.com/questions/67539049/threading-thread-not-working-but-thread-does

Python Software Foundation. (n.d.). logging — Logging facility for Python. Python Documentation. Retrieved December 3, 2024, from https://docs.python.org/3/library/logging.html

Python Software Foundation. (n.d.). os — Miscellaneous operating system interfaces. Python Documentation. Retrieved December 3, 2024, from https://docs.python.org/3/library/os.html

Python Software Foundation. (n.d.). shutil — High-level file operations. Python Documentation. Retrieved December 3, 2024, from https://docs.python.org/3/library/shutil.html

Python Software Foundation. (n.d.). threading — Thread-based parallelism. Python Documentation. Retrieved December 3, 2024, from https://docs.python.org/3/library/threading.html

Python Software Foundation. (n.d.). tkinter — Python interface to Tcl/Tk. Python Documentation. Retrieved December 3, 2024, from https://docs.python.org/3/library/tkinter.html

PythonTutorial. (n.d.). Tkinter Open File Dialog. Retrieved December 3, 2024, from https://www.pythontutorial.net/tkinter/tkinter-open-file-dialog/

PythonTutorial. (n.d.). Tkinter Scrollbar. Retrieved December 3, 2024, from https://www.pythontutorial.net/tkinter/tkinter-scrollbar/

**Results**

SeedID aimed to create a machine learning application for seed identification, integrating the deep learning framework found in Tensorflow and pairing it with Azure’s robust data management. The implementation required overcoming technical challenges in data collection, model training, and cloud integration. The original plan to use PyTorch and FastAI encountered compatibility issues with Python 3.12.4 and Windows 11. Transitioning to TensorFlow provided a temporary solution, but GPU limitations required reverting to PyTorch with CUDA for accelerated training.

Data collection posed another challenge, as publicly available seed datasets were insufficient or difficult to source. To address this, a custom web crawler was deployed which utilized Selenium and BeautifulSoup to retrieve seed images. This automated solution parsed and downloaded images, which were then uploaded to Azure for model training.

Initial training results showed low accuracy. Enhancements included implementing data augmentation techniques, such as rotations and flips, to improve generalization. Transitioning to a ResNet50 architecture allowed for better feature extraction. Additionally, increasing training epochs and optimizing learning rates further improved performance. Azure Blob Storage integration required partitioning into containers and implementing robust APIs for seamless uploads and retrievals. Model accuracy improved from an initial 30% to 95%+ after optimizing training with ResNet50 and data augmentation. GPU acceleration reduced training time by 65 percent compared to CPU-only processing. A Tkinter-based GUI was developed to support image uploads, training, and log monitoring. Real-time logging functionality was implemented with the CSVLogHandler class, improving error tracking and debugging

The SeedID project has successfully developed a functional application for seed identification. Key milestones included improved model accuracy, seamless cloud integration, and an enhanced user interface. Future steps involve expanding the dataset, hosting the program online, and further enhancements to the user experience.

**Matt Morrow Sat Sep 21, 2024**

**CIS5898**

**First Progress Report**

**09/14 –** The project proposal has been submitted.

**09/15** - The initial steps have been taken in the development of a SeedID. The program now connects to an Azure Blob Storage, uploads images, and processes them using PyTorch; with CUDA support for GPU acceleration.

CUDA 12.4: This version of CUDA leverages NVIDIA GPUs for high-performance computing. This will speed up deep learning tasks that involve processing large datasets or performing neural network computations.

PyTorch: Installed as the primary deep learning library. PyTorch provides seamless integration with CUDA, enabling the use of GPUs for faster image processing and model training.

A black screen with white text

Description automatically generated

**09/16 -** Azure blob storage has been partitioned. The container "seedimages" was created within the Azure Blob Storage account "fitseedid" to store the images. The Azure Blob service client was set up using the connection string to communicate with the container.

A screenshot of a computer

Description automatically generated

**09/18 -** The Program can now communicate with the Azure database and add an image from a local file path.

A screenshot of a computer program

Description automatically generated

The core function at this point in development is:

*def upload\_image(self, local\_file\_path):*

*upload\_image\_to\_azure(self.connection\_string, self.container\_name, local\_file\_path)*

Which passes the connection string, container name, and file path to a separate class that uses a built in function, *blob\_service\_client.get\_container\_client()*, to send the blob to the container.

**Matt Morrow Sat Oct 5, 2024**

**CIS5898**

**Second Progress Report**

This period of the project’s development focused on acquiring initial data and ensuring it can be uploaded/accessed from the Azure Database. A specialized program was needed and thus developed to perform web crawling for the purpose of collecting seed data. This custom crawler successfully retrieved images from vecteezy.com and stored them in a designated local folder for further processing and analysis.

**Details of how the program works:**

This code in the program automates the process of web scraping and downloading seed images using Selenium and BeautifulSoup. It starts by initializing a Selenium WebDriver with Chrome, configuring it to ignore SSL and certificate errors. The script then constructs a URL based on a specified seed type and navigates to the vecteezy.com search results page for relevant seed images.

A collage of corn seeds

Description automatically generatedOnce the page is loaded, BeautifulSoup is used to parse the HTML content and extract all image elements (img tags). The code identifies and collects URLs of the images by checking attributes like src, data-src, and srcset, filtering for valid image formats such as .jpg and .png. After gathering the image URLs, the script creates a local folder named after the seed type and proceeds to download each image, saving them in the folder with a sequential naming convention. If an image fails to download, an error message is printed, and the script continues downloading the remaining images. Upon completion, the script outputs the total number of successfully downloaded images.

Those downloaded images show up on the local folder as such:

Then from the local folder the files are imported to the Azure database using the upload\_image function.

    def upload\_image(self, blob\_name):

        upload\_image\_to\_azure(self.connection\_string, self.container\_name, blob\_name)

A screenshot of a computer screen

Description automatically generatedThe following screenshot illustrates that the image uploading functionality is performing as expected in the primary SeedID program, confirming that the process is operational and successfully handling uploads:

A white background with black text

Description automatically generatedSo far 119 images have been uploaded for 4 different seeds that have been selected to be the initial run. Those seeds are sunflower, corn, wheat, and coconut.

**Changes to the initial proposal/plan:**

I opted to use Visual Studio Code over PyCharm due to its superior integration with Azure services. This choice enhances workflow efficiency, particularly in managing cloud-based resources and deployments, while also providing more seamless connectivity for development and debugging in Azure environments.

**Matt Morrow Sat Oct 19, 2024**

**CIS5898**

**Third Progress Report**

During this progress period, I attempted to implement the training process outlined in the article, *“Is it a Bird? Creating a Model from Your Own Data.”* The program was to connect to the Azure database, retrieve the stored images, load them into a proprietary data loader, and train the model using the **vision\_learner()** function.

A screenshot of a computer screen

Description automatically generated

However, due to what seems to be an unexplainable compatibility issue with PyToch/FastAi and the latest Python version (3.12.4) with Windows 11; I needed to resort to an alternative method: Tensorflow.

A screenshot of a computer

Description automatically generatedAs you can see in the below screenshot the output was generating “nan” and 0.00 scores during the training of the custom model. Days were spent troubleshooting this process but to no avail.

After installing Tensorflow, I introduced a new class defined as, TrainModel, that prepares image data and trains a machine learning model using three functions, load\_data, build\_model and train\_model. The load\_data method loads and preprocesses image data from a directory, splitting it into training and validation sets. The build\_model method constructs a convolutional neural network (CNN) with multiple layers, including convolutional layers, max-pooling layers, and fully connected layers, which is compiled with an optimizer and loss function suitable for multi-class classification. Lastly, in the train\_model method, the model is trained using the provided datasets for a specified number of epochs, and the trained model is saved to a file for future use. These epochs can be altered to enhance the overall prediction accuracy of the model but it does require more processing power as the epoch increases.

When launching the train model action, the following output is displayed. Showing semi-successful results. I would like to see higher rates of accuracy, but it is a start in the right direction.

A screenshot of a computer

Description automatically generated

A basic GUI was designed using the Tkinter framework. This UI contain the buttons which allow the user to perform key actions such as uploading an image, training the model, and reloading the model after training. Additionally, it features an image display area that shows the uploaded image and a label that informs the user of the prediction score generated by the model.

A yellow corn seed with a white background

Description automatically generated

When the user selects “Upload Image”, a method defined as, upload\_image, opens a file dialog for the user to select an image, resizes the image for display in the GUI, and updates the image label to show the selected image. Once an image is uploaded, it calls predict\_image method to process and predict the class of the image. If an error occurs during prediction, an exception is caught, and a relevant message is displayed.

**Next Steps**

1). Refine and improve the accuracy of the model.

2). Expand the custom image model database to include more seeds.

3). Improve logging/exception handling.

4). Include a log report that is saved as a file in the project’s directory.

5). Create comprehensive unit tests and run regression testing

**Matt Morrow Sat Nov 2, 2024**

**CIS5898**

**Fourth Progress Report**

Since the last progress report, SeedID has made significant progress in enhancing both the functionality and user experience of the model. The training model has been optimized to achieve better prediction accuracy. This was done by adjusting the model architecture and parameters. I increased the number of epochs to allow more extensive training and fine-tuned the learning rate for more efficient convergence. In addition to utilizing the resnet50 model, I applied data augmentation techniques, such as rotations and flips, to enhance the model’s generalization and robustness. In order to support the process heavy training methods I had to leverage the GPU (as expected). However, I again encountered limitations with GPU usage in TensorFlow, as it only supported up to Python 3.7. To leverage GPU acceleration, I had to fall back to utilizing the PyTorch framework.

Comprehensive try/except blocks were added to handle errors within the critical functions. This helps to prevent program crashes and improves overall reliability. Logging was implemented in detail with a new class, named CSVLogHandler, which writes log entries to a local CSV file. It tracks events at different levels, such as INFO, WARNING, ERROR, and assigns each event its appropriate timestamp. This class includes three key functions:

1). initialize\_log\_file(): which sets up the CSV log file with headers if it doesn’t already exist

2). log\_to\_dataframe(): which appends a new log entry to the CSV file with the timestamp, level, and message.

3). Emit(): to handle a log event and send it to the cvs

The log entries appear as such:

A screen shot of a computer

Description automatically generated

A pile of yellow corn seeds

Description automatically generatedThe user interface was also enhanced by adding an embedded console for real-time logs, improving the application’s layout, and enhancing the color scheme for better readability and visual appeal. The new GUI now displays log messages directly in the interface via an alertbar, making it easier to monitor any issues that arise during training or prediction.

In summary, the recent updates have made the model more accurate, the application more resilient, and the user experience more seamless. With the submission of this report, I am also including the model (as requested). Though it is in a .pth file so I hope it uploads and is readable.

TODO items:

1). Expand seed classes

2). Get tier 2 seed images

3). Research/host the program online

4). Improve UI

5). Ensure everything is backed up on the cloud.

6). Create unit tests

**Matt Morrow**

**CIS5898 Wed Nov 20, 2024**

**Last Interim Progress Report**

As of the last check in, the program has come along nicely with new enhancements to the UI, expanded seed training, and even further improvements to identification accuracy. One of the most significant additions during this leg of the project was the integration of Azure into the UI and the ability to store/receive images from the cloud, train the model locally using the machines GPU, and then storing the trained model back into the cloud. Tabs were added to the sidebar of the UI to separate the main program functionality from the administrative settings for managing Azure, as seen below:

A screenshot of a computer

Description automatically generatedA screenshot of a program

Description automatically generated

TODO items:

1). Expand seed classes

2). Get tier 2 seed images

3). Research/host the program online

4). Improve UI

5). Ensure everything is backed up on the cloud.

6). Create unit tests

B: Software Configuration  
This section provides all steps necessary to install and configure application to run

C: User’s Manual  
This section provides all instructions for user to run the application and perform all  
functionality once the application has been installed