**Matt Morrow**

**CIS5898**

**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

As requested via email, please see the below for the projects code (as PDF):

MAIN PROJECT (SEEDID.PY) FILE

# %% Import libraries

import os

import shutil

import tkinter as tk

from tkinter import filedialog, messagebox, ttk

import threading

import torch

import torch.nn as nn

import torch.optim as optim

from torch.utils.data import DataLoader

from torchvision import datasets, transforms, models

from PIL import Image, ImageTk

import numpy as np

from azure.storage.blob import BlobServiceClient

from azure\_upload import upload\_image\_to\_azure

from azure\_retrieve import retrieve\_image\_from\_azure

import logging

from CSVLogHelper import CSVLogHandler   #Import CSVLogger from CSVLogHelper.py

import shutil

#TODO: FIX THE ALERT BANNER

#TODO: FIX HOME BUTTON AFTER AZURE DIRECOTRY IS PRESSED

#TODO: MAKE HOME GO AWAY AFTER PROGRAM TAB IS PRESSED

#CITATION: Benham, A. (2022, September 30). Deep learning tutorial for beginners | AI neural networks explained. YouTube. https://www.youtube.com/watch?v=r7Am-ZGMef8

#Setting up CUDA and GPU usage

#Check for GPU availability

# %% Set up device and logger

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

print(f"Using device: {device}")

#Initialize the logger helper class

logger = logging.getLogger("SeedIdentifierApp")

logger.setLevel(logging.INFO)

logger.addHandler(CSVLogHandler())

#CLASS FIXED CONSTANT VARIABLES; Controls for the program

#Number of seeds in the directory to ID

# %% Define constants

class\_nums = 6

class\_names = ['coconut', 'corn', 'flaxseed', 'pumpkin', 'sunflower', 'wheat']

epoch\_num = 75

LEARN\_RATE = 1e-2 #A smaller learning rate (like 2 or 3) reduces the size of the steps; allowing the model to move more precisely toward the minimum of the loss function.

#Azure Blob Manager for image storage

#CITATION: Microsoft. Connect to and query Azure SQL Database using Python and the pyodbc driver https://learn.microsoft.com/en-us/azure/azure-sql/database/azure-sql-python-quickstart?view=azuresql&tabs=windows%2Csql-inter

# %% Azure Blob Manager Class

class AzureImageManager:

    def \_\_init\_\_(self):

        self.connection\_string = "DefaultEndpointsProtocol=https;AccountName=fitseedid;AccountKey=P+Gn6AEYlmaEvhMZUkpzOnCVoEKP8kAniBg8YL5coK/ACbvJq9hzXFNys1FBVRbSA8NtZRf3tZYz+ASt/44GrA==;EndpointSuffix=core.windows.net"

        self.container\_name = "seedimages"

        self.blob\_service\_client = BlobServiceClient.from\_connection\_string(self.connection\_string)

        self.container\_client = self.blob\_service\_client.get\_container\_client(self.container\_name)

        logger.info("Azure Blob Manager initialized.")

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

        logger.info(f"Uploading image: {blob\_name}")

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

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

        logger.info(f"Retrieving image: {blob\_name}")

        image\_tensor = retrieve\_image\_from\_azure(self.connection\_string, self.container\_name, blob\_name)

        pil\_image = Image.fromarray(image\_tensor)

        return pil\_image

    def delete\_all\_blobs(self):

        logger.info("Deleting all blobs in the container.")

        blob\_list = self.container\_client.list\_blobs()

        for blob in blob\_list:

            self.container\_client.delete\_blob(blob.name)

    # %% Download all images

    def download\_all\_images(self, local\_directory):

        # Delete the directory if it already exists

        if os.path.exists(local\_directory):

            shutil.rmtree(local\_directory)

            print(f"Deleted existing directory: {local\_directory}")

        # Create the main directory

        os.makedirs(local\_directory)

        print(f"Created main directory: {local\_directory}")

        #Pre-create class folders in the main directory

        for class\_name in class\_names:

            class\_folder = os.path.join(local\_directory, class\_name)

            os.makedirs(class\_folder, exist\_ok=True)

            print(f"Created folder for class '{class\_name}': {class\_folder}")

        blobs = self.container\_client.list\_blobs()

        for blob in blobs:

            blob\_client = self.container\_client.get\_blob\_client(blob)

            #Check if the blob name contains a class name as a prefix

            assigned\_class = None

            for class\_name in class\_names:

                if class\_name in blob.name:

                    assigned\_class = class\_name

                    break

            if assigned\_class is None:

                logger.error(f"No matching class found for blob name: {blob.name}")

                continue

            #Define the download path within the appropriate class folder

            download\_path = os.path.join(local\_directory, assigned\_class, os.path.basename(blob.name))

            #Download the blob to the specified path

            try:

                with open(download\_path, "wb") as file:

                    data = blob\_client.download\_blob()

                    file.write(data.readall())

                logger.info(f"Downloaded {blob.name} to {download\_path}")

                print(f"Downloaded '{blob.name}' to '{download\_path}'")

            except Exception as e:

                logger.error(f"Failed to download {blob.name}: {e}")

                print(f"Failed to download {blob.name}: {e}")

        # Print the directory structure for verification

        print("\nContents of temp\_azure\_images directory after download:")

        for root, dirs, files in os.walk(local\_directory):

            print(f"Directory: {root}")

            for name in dirs:

                print(f"  Subdirectory: {name}")

            for name in files:

                print(f"  File: {name}")

    def list\_images\_by\_class(self, class\_names):

        class\_dict = {class\_name: [] for class\_name in class\_names}

        # List all blobs in the container

        blobs = self.container\_client.list\_blobs()

        for blob in blobs:

            blob\_name = blob.name

            # Check if the blob name contains a class name as a prefix or directory

            for class\_name in class\_names:

                if class\_name in blob\_name:

                    class\_dict[class\_name].append(blob\_name)

                    break

        return class\_dict

    def clear\_container(self):

        logger.info("Deleting all blobs in the Azure container.")

        blob\_list = self.container\_client.list\_blobs()

        for blob in blob\_list:

            try:

                self.container\_client.delete\_blob(blob.name)

                logger.info(f"Deleted blob: {blob.name}")

                print(f"Deleted blob: {blob.name}")

            except Exception as e:

                logger.error(f"Failed to delete blob {blob.name}: {e}")

                print(f"Failed to delete blob {blob.name}: {e}")

        logger.info("Container cleared successfully.")

        print("Azure container cleared successfully.")

    #Downloads the trained model from Azure to a temporary location.

    #CITATION: Microsoft. Download a blob with Python https://learn.microsoft.com/en-us/azure/storage/blobs/storage-blob-download-python

    # %% Download the model from Azure

    def download\_model\_from\_azure(self, model\_path, blob\_name="trained\_model.pth"):

        #Downloads the trained model from Azure to a temporary location.

        try:

            blob\_client = self.container\_client.get\_blob\_client(blob\_name)

            with open(model\_path, "wb") as file:

                data = blob\_client.download\_blob()

                file.write(data.readall())

            logger.info(f"Model downloaded from Azure as {blob\_name}")

            print(f"Model downloaded from Azure as {blob\_name}")

        except Exception as e:

            logger.error(f"Failed to download model from Azure: {e}")

            print(f"Failed to download model from Azure: {e}")

    #Uploads an image file to Azure with a specified blob name.

    def upload\_image\_with\_path(self, file\_path, blob\_name):

            try:

                blob\_client = self.container\_client.get\_blob\_client(blob\_name)

                with open(file\_path, "rb") as data:

                    blob\_client.upload\_blob(data, overwrite=True)

                logger.info(f"Successfully uploaded {blob\_name} to Azure.")

                print(f"Uploaded {blob\_name} to Azure.")

            except Exception as e:

                logger.error(f"Failed to upload {file\_path} to Azure: {e}")

                print(f"Failed to upload {file\_path} to Azure: {e}")

#CITATION: GeeksForGeeks. Residual Networks (ResNet) – Deep Learning https://www.geeksforgeeks.org/residual-networks-resnet-deep-learning/

# %% TrainModel Class

class TrainModel:

    def \_\_init\_\_(self, num\_classes=class\_nums, azure\_manager=None):

        self.num\_classes = num\_classes

        self.model = None

        self.azure\_manager = azure\_manager  # Pass AzureImageManager instance here

    def load\_data(self, data\_dir, img\_size=256, batch\_size=32):

        logger.info(f"Loading data from directory: {data\_dir}")

        transform = transforms.Compose([

            transforms.RandomResizedCrop(256, scale=(0.8, 1.0)),

            transforms.RandomHorizontalFlip(),

            transforms.RandomVerticalFlip(),

            transforms.RandomRotation(20),

            transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),

            transforms.ToTensor(),

            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])  # Standard normalization for ResNet

        ])

        train\_ds = datasets.ImageFolder(root=data\_dir, transform=transform)

        val\_size = int(0.2 \* len(train\_ds))

        train\_size = len(train\_ds) - val\_size

        train\_ds, val\_ds = torch.utils.data.random\_split(train\_ds, [train\_size, val\_size])

        train\_loader = DataLoader(train\_ds, batch\_size=batch\_size, shuffle=True)

        val\_loader = DataLoader(val\_ds, batch\_size=batch\_size, shuffle=False)

        class\_names = train\_ds.dataset.classes

        return train\_loader, val\_loader, class\_names

    def build\_model(self, input\_shape=(256, 256, 3), num\_classes=class\_nums):

        print("Building model...")

        self.model = models.resnet50(pretrained=True)

            # Freeze all layers initially

        for param in self.model.parameters():

            param.requires\_grad = False

        # Unfreeze the last few layers

        for param in list(self.model.parameters())[-10:]:

            param.requires\_grad = True

        # Replace the final layer

        self.model.fc = nn.Sequential(

            nn.Linear(self.model.fc.in\_features, 512),

            nn.ReLU(),

            nn.Dropout(0.4),

            nn.Linear(512, self.num\_classes)

        )

        self.model = self.model.to(device)

        logger.info("Model built successfully.")

    def evaluate(self, val\_loader):

        self.model.eval()

        correct = 0

        total = 0

        with torch.no\_grad():

            for images, labels in val\_loader:

                images, labels = images.to(device), labels.to(device)

                outputs = self.model(images)

                \_, predicted = torch.max(outputs, 1)

                total += labels.size(0)

                correct += (predicted == labels).sum().item()

        accuracy = correct / total

        logger.info(f"Validation Accuracy: {accuracy:.4f}")

        return correct / total

    def load\_model(self, path="temp/trained\_model.pth"):

        #Check if the model exists locally; if not, download it from Azure

        if not os.path.exists(path) and self.azure\_manager:

            print("Downloading model from Azure...")

            self.azure\_manager.download\_model\_from\_azure(path)

        try:

            print(f"Attempting to load model from {path}...")

            self.model = models.resnet50(pretrained=True)

            self.model.fc = nn.Sequential(

                nn.Linear(self.model.fc.in\_features, 512),

                nn.ReLU(),

                nn.Dropout(0.2),

                nn.Linear(512, self.num\_classes)

            )

            self.model.load\_state\_dict(torch.load(path))

            self.model = self.model.to(device)

            self.model.eval()

            logger.info(f"Model loaded successfully from {path}")

            print(f"Model loaded successfully from {path}")

        except Exception as e:

            logger.error(f"Failed to load model from {path}: {e}")

            print(f"Error loading model from {path}: {e}")

    def save\_model(self, path):

        #Ensure the parent directory exists

        directory = os.path.dirname(path)

        if not os.path.exists(directory):

                os.makedirs(directory)

        try:

            #Save only the model's parameters (state dictionary)

            torch.save(self.model.state\_dict(), path)

            logger.info(f"Model saved to {path}")

            print(f"Model saved successfully to {path}")

            #Upload to Azure if the AzureImageManager is available

            if self.azure\_manager:

                self.azure\_manager.save\_model\_to\_azure(path)

        except Exception as e:

            logger.error(f"Failed to save model: {e}")

            print(f"Error saving model: {e}")

    # %% Train the model

    def train\_model(self, train\_loader, val\_loader, epochs=epoch\_num, learning\_rate=LEARN\_RATE):

        self.build\_model()

        criterion = nn.CrossEntropyLoss(label\_smoothing=0.1)  #Apply slight smoothing (adjusts probabilities slightly to introduce uncertainty)

        optimizer = optim.SGD(self.model.fc.parameters(), lr=learning\_rate, momentum=0.9)

        for epoch in range(epochs):

            self.model.train()

            running\_loss = 0.0

            for images, labels in train\_loader:

                images, labels = images.to(device), labels.to(device)

                optimizer.zero\_grad()

                outputs = self.model(images)

                loss = criterion(outputs, labels)

                loss.backward()

                optimizer.step()

                running\_loss += loss.item()

            #Validation accuracy

            val\_accuracy = self.evaluate(val\_loader)

            print(f"Epoch {epoch+1}/{epochs}, Loss: {running\_loss/len(train\_loader):.4f}, Val Accuracy: {val\_accuracy:.4f}")

            logger.info(f"Epoch {epoch+1}/{epochs}, Loss: {running\_loss/len(train\_loader):.4f}, Val Accuracy: {val\_accuracy:.4f}")

    # %% Save the model to azure

    def save\_model\_to\_azure(self, model\_path, blob\_name="trained\_model.pth"):

        """Uploads the trained model to Azure."""

        try:

            blob\_client = self.container\_client.get\_blob\_client(blob\_name)

            with open(model\_path, "rb") as data:

                blob\_client.upload\_blob(data, overwrite=True)

            logger.info(f"Model saved to Azure as {blob\_name}")

            print(f"Model saved to Azure as {blob\_name}")

        except Exception as e:

            logger.error(f"Failed to save model to Azure: {e}")

            print(f"Failed to save model to Azure: {e}")

#Main application class

#CITATION: Graphical User Interface (GUI) with Tkinter https://docs.python.org/3/library/tkinter.html

# %% SeedIdentifierApp Class

class SeedIdentifierApp:

    def \_\_init\_\_(self, root):

        self.root = root

        self.root.title("SeedID")

        self.root.geometry("800x600")

        self.azure\_manager = AzureImageManager()

        # Initialize the model

        self.tm = TrainModel(azure\_manager=self.azure\_manager)

        # Alert bar at the top for displaying error or success messages

        self.alert\_bar = tk.Label(

            self.root,

            text="",

            bg="yellow",

            fg="black",

            font=("Helvetica", 12, "bold"),

            anchor="center"

        )

        self.alert\_bar.pack(side=tk.TOP, fill=tk.X, padx=0, pady=0)  # Attach at the top and stretch horizontally

        self.alert\_bar.pack\_forget()  # Initially hide the alert bar

        #Style for the tabs

        style = ttk.Style()

        style.configure("TNotebook.Tab", font=("Helvetica", 13, "bold"), padding=[10, 5])  # Set font size and padding

        # Create sidebar frame for tabs

        self.sidebar = tk.Frame(self.root, width=200, bg="#ADD8E6")

        self.sidebar.pack(side=tk.LEFT, fill=tk.Y)

        # Create a Notebook (tabs) for the sidebar

        self.notebook = ttk.Notebook(self.sidebar, style="TNotebook")

        self.notebook.pack(fill=tk.BOTH, expand=True)

        # Create frames for each tab

        #CITATION: GeeksForGeeks. Creating Tabbed Widget With Python-Tkinter https://www.geeksforgeeks.org/creating-tabbed-widget-with-python-tkinter/

        self.main\_tab = tk.Frame(self.notebook, bg="#ADD8E6")

        self.settings\_tab = tk.Frame(self.notebook, bg="#ADD8E6")

        #Add tabs to the Notebook

        self.notebook.add(self.main\_tab, text="Program")

        self.notebook.add(self.settings\_tab, text="Settings")

        #Bind tab change event to handle result label visibility

        self.notebook.bind("<<NotebookTabChanged>>", self.on\_tab\_change)

        #Main content area frame

        self.main\_area = tk.Frame(self.root)

        self.main\_area.pack(side=tk.RIGHT, expand=True, fill=tk.BOTH)

        #Loading label for showing training in progress when model is being trained

        self.loading\_label = tk.Label(self.main\_area, text="", font=("Helvetica", 12))

        self.loading\_label.pack(pady=10)

        self.home\_info\_label\_main = tk.Label(

            self.main\_area,

            text="",  # Initially empty

            font=("Helvetica", 12),

            justify="left",

            wraplength=600,

        )

        self.home\_info\_label\_main.pack(pady=10, padx=10, fill=tk.BOTH)

        #MAIN PROGRAM TAB

        #Sidebar buttons

        self.upload\_button = tk.Button(self.main\_tab, text="Upload Image", font=("Helvetica", 13), command=self.upload\_image)

        self.upload\_button.pack(pady=10, padx=10, fill=tk.X)

        self.train\_button = tk.Button(self.main\_tab, text="Train Model", font=("Helvetica", 13), command=self.train\_model\_gui)

        self.train\_button.pack(pady=10, padx=10, fill=tk.X)

        #SETTINGS TAB

        #Add the new "Home" button to the settings tab

        self.home\_button = tk.Button(self.settings\_tab, text="Home", font=("Helvetica", 13), command=self.display\_home\_info)

        self.home\_button.pack(pady=10, padx=10, fill=tk.X)

        self.sync\_button = tk.Button(self.settings\_tab, text="Sync to Azure", font=("Helvetica", 13), command=self.sync\_to\_azure\_threaded)

        self.sync\_button.pack(pady=10, padx=10, fill=tk.X)

        self.show\_azure\_button = tk.Button(self.settings\_tab, text="Show Azure Directory", font=("Helvetica", 13), command=self.show\_azure\_directory)

        self.show\_azure\_button.pack(pady=10, padx=10, fill=tk.X)

        self.clear\_button = tk.Button(self.settings\_tab, text="Clear Azure Database", font=("Helvetica", 13), command=self.clear\_azure\_database\_threaded)

        self.clear\_button.pack(pady=10, padx=10, fill=tk.X)

        #Image display label

        self.img\_label = tk.Label(self.main\_area)

        self.img\_label.pack(pady=10)

        #Result display frame at the bottom of the main area

        self.result\_frame = tk.Frame(self.main\_area)

        self.result\_frame.pack(side=tk.BOTTOM, pady=100)  # Anchor it to the bottom

        #Result label inside the result frame

        self.result\_label = tk.Label(self.result\_frame, text="Prediction results will appear here.", font=("Helvetica", 12, "bold"))

        self.result\_label.pack()

        #Azure directory display frame within the main app

        #CITATION: PythonTutorial. Tkinter Scrollbar https://www.pythontutorial.net/tkinter/tkinter-scrollbar/

        self.azure\_dir\_frame = tk.Frame(self.main\_area)

        self.azure\_dir\_frame.pack(fill=tk.BOTH, expand=True)

        self.azure\_dir\_canvas = tk.Canvas(self.azure\_dir\_frame)

        self.scrollbar = tk.Scrollbar(self.azure\_dir\_frame, orient="vertical", command=self.azure\_dir\_canvas.yview)

        self.scrollable\_frame = tk.Frame(self.azure\_dir\_canvas)

        self.scrollable\_frame.bind(

            "<Configure>",

            lambda e: self.azure\_dir\_canvas.configure(scrollregion=self.azure\_dir\_canvas.bbox("all"))

        )

        self.azure\_dir\_canvas.create\_window((0, 0), window=self.scrollable\_frame, anchor="nw")

        self.azure\_dir\_canvas.configure(yscrollcommand=self.scrollbar.set)

        self.azure\_dir\_canvas.pack(side="left", fill="both", expand=True)

        self.scrollbar.pack(side="right", fill="y")

        #Load the trained model if available

        try:

            self.tm.load\_model('model/trained\_model.pth')

            print("Model loaded successfully.")

        except Exception as e:

            print("Model not found or failed to load.")

            logger.warning(f"Model not found or failed to load: {e}")

            self.tm.model = None  # Explicitly set to None if loading fails

    def on\_tab\_change(self, event):

        #Check which tab is selected

        selected\_tab = self.notebook.index(self.notebook.select())

        if selected\_tab == 0:  #Program tab

            #Show the prediction result label text

            self.result\_label.config(text="Prediction results will appear here.")

            #Show the image label if it's hidden

            self.img\_label.pack\_forget()

            #Hide the Azure directory frame

            self.azure\_dir\_frame.pack\_forget()

        elif selected\_tab == 1:  #Settings tab

            #Hide the prediction result label text

            self.result\_label.config(text="")

            #Hide the image label when switching to settings

            self.img\_label.pack\_forget()

            #Show the Azure directory frame

            self.azure\_dir\_frame.pack(fill=tk.BOTH, expand=True)

            self.display\_home\_info()

    #Display descriptions of the settings buttons.

    def display\_home\_info(self):

        self.azure\_dir\_frame.pack\_forget()

        descriptions = (

            "Descriptions of Settings Options:\n\n"

            "1. \*\*Sync to Azure\*\*: \nUpload all local seed images to Azure Blob Storage.\n\n"

            "2. \*\*Show Azure Directory\*\*: \nDisplay the list of images stored in Azure, organized by classes.\n\n"

            "3. \*\*Clear Azure Database\*\*: \nDelete all images from Azure Blob Storage for a fresh start.\n\n"

        )

        self.home\_info\_label\_main.config(text=descriptions)

        self.home\_info\_label\_main.pack()  #Show the label

    def show\_azure\_directory(self):

        #Hide the home description label

        self.home\_info\_label\_main.pack\_forget()

        #Ensure the Azure directory frame is visible

        self.azure\_dir\_frame.pack(fill=tk.BOTH, expand=True)

        #Clear any existing content in the scrollable frame

        for widget in self.scrollable\_frame.winfo\_children():

            widget.destroy()

        #Fetch the list of images organized by class from Azure

        class\_images = self.azure\_manager.list\_images\_by\_class(class\_names)

        #Display class names and their images in the scrollable frame

        for class\_name, images in class\_images.items():

            #Class Label

            tk.Label(self.scrollable\_frame, text=class\_name, font=("Helvetica", 12, "bold")).pack(anchor="w", padx=10, pady=5)

            #List of images in this class

            for image\_name in images:

                tk.Label(self.scrollable\_frame, text=image\_name, font=("Helvetica", 10)).pack(anchor="w", padx=20)

    def show\_alert(self, message):

        self.alert\_bar.config(text=message, anchor="center")

        #Repack to ensure alignment

        self.alert\_bar.pack(side=tk.TOP, fill=tk.X, padx=0, pady=0)

    def hide\_alert(self):

        self.alert\_bar.pack\_forget()

    # %% Upload Image

    def upload\_image(self):

        if not self.tm.model:

            self.show\_alert("No model found. Please train the model first.")

            return

        file\_path = filedialog.askopenfilename(

            title="Select an Image",

            filetypes=(("Image Files", "\*.jpg;\*.jpeg;\*.png"), ("All Files", "\*.\*"))

        )

        if file\_path:

            logger.info(f"Image uploaded: {file\_path}")

            img = Image.open(file\_path)

            img.thumbnail((200, 200))

            img\_tk = ImageTk.PhotoImage(img)

            self.img\_label.pack(pady=10)

            self.img\_label.config(image=img\_tk)

            self.img\_label.image = img\_tk

            self.predict\_image(file\_path)

    # %% Predict Image

    #CITATION: GeeksForGeeks. Loading Images in Tkinter using PIL https://www.geeksforgeeks.org/loading-images-in-tkinter-using-pil/

    def predict\_image(self, file\_path):

        self.hide\_alert()  #Hide any previous alerts

        if not self.tm.model:

            self.show\_alert("Model not loaded. Please train the model first.")

            logger.warning("Attempted prediction without loaded model.")

            return

        img = Image.open(file\_path).convert('RGB')

        transform = transforms.Compose([

            transforms.Resize((256, 256)),

            transforms.ToTensor(),

            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])

        ])

        img\_tensor = transform(img).unsqueeze(0)  # Add batch dimension

        #Move the tensor to the same device as the model and ensure the input tensor is on the same device

        img\_tensor = img\_tensor.to(device)

        self.tm.model.to(device)

        try:

            self.tm.model.eval()  #Set the model to evaluation mode

            with torch.no\_grad():  #Disable gradient calculation for inference

                output = self.tm.model(img\_tensor)  #Forward pass

                \_, pred\_class = torch.max(output, 1)  #Get the predicted class

                confidence = (torch.softmax(output, dim=1)[0][pred\_class].item()) \* 100   #Get confidence

                self.display\_results(pred\_class.item(), confidence)

                logger.info(f"Prediction: {pred\_class}, Confidence: {confidence:0f}%")

        except Exception as e:

            self.show\_alert(f"Prediction failed: {e}")

            logger.error(f"Prediction failed: {e}")

    # %% Display the results

    def display\_results(self, pred\_class, confidence):

        if confidence < 85: #If the confidence score is less than 85% show as "no results found"

            result = "No results found"

        else:

            result = f"Prediction: {class\_names[pred\_class]}\nConfidence: {confidence:0f}%"

        self.result\_label.config(text=result)

    #Show a loading message.

    def show\_loading(self, message):

        self.loading\_label.config(text=message)

        self.loading\_label.pack()

        self.root.update\_idletasks()

    #Hide the loading message.

    def hide\_loading(self):

        self.loading\_label.config(text="")

        self.loading\_label.pack\_forget()

        self.root.update\_idletasks()

    def train\_model\_gui(self):

        #BUG PATCH: Prevent duplicate model training

        if hasattr(self, 'training\_thread') and self.training\_thread.is\_alive():

            messagebox.showinfo("Training in Progress", "Model is already training. Please wait.")

            return

        #Hide any previous alerts

        self.hide\_alert()

        self.show\_loading("Training... Please wait.")

        threading.Thread(target=self.run\_training\_process).start()

        #DL all images from azure to train the model.

        temp\_training\_dir = './temp\_azure\_images'

        self.azure\_manager.download\_all\_images(temp\_training\_dir)

        self.hide\_loading()

        messagebox.showinfo("Training", "Model trained and saved successfully.")

        logger.info(f"Model training completed and saved")

        # Clean up the temporary directory after training

        shutil.rmtree(temp\_training\_dir)

    def run\_training\_process(self):

        temp\_training\_dir = './temp\_azure\_images'

        self.azure\_manager.download\_all\_images(temp\_training\_dir)A

        train\_loader, val\_loader, class\_names = self.tm.load\_data(temp\_training\_dir)

        #Run training; ensure training completes before moving forward

        self.tm.train\_model(train\_loader, val\_loader)

        #Save the trained model (if necessary)

        #Only after training completes, hide loading and show the completion message

        self.hide\_loading()

        messagebox.showinfo("Training", "Model trained and saved successfully.")

        logger.info("Model training completed and saved")

        #Clean up the temporary directory after training

        #CITATION: Delete an entire directory tree using Python | shutil.rmtree() method https://www.geeksforgeeks.org/delete-an-entire-directory-tree-using-python-shutil-rmtree-method/

        shutil.rmtree(temp\_training\_dir)

    #Start a thread to sync to Azure

    def sync\_to\_azure\_threaded(self):

        threading.Thread(target=self.sync\_to\_azure).start()

    #CITATION: Microsoft. Upload a block blob with Python https://learn.microsoft.com/en-us/azure/storage/blobs/storage-blob-upload-python

    def sync\_to\_azure(self):

        #Syncs the local images to Azure

        print("Starting sync to Azure...")

        self.show\_loading("Syncing to Azure... please wait.")

        #Clear all existing blobs in the container first

        print("Clearing Azure container...")

        self.azure\_manager.clear\_container()

        #Define the local folder containing images organized by class

        local\_directory = './temp\_images'

        #Walk through all files and folders in the directory

        for root, dirs, files in os.walk(local\_directory):

            for file\_name in files:

                file\_path = os.path.join(root, file\_name)

                blob\_name = os.path.relpath(file\_path, local\_directory)

                print(f"Uploading {file\_path} as {blob\_name}")

                try:

                    # Upload file with folder structure

                    self.azure\_manager.upload\_image\_with\_path(file\_path, blob\_name)

                    print(f"Uploaded {file\_name} to Azure.")

                except Exception as e:

                    print(f"Failed to upload {file\_path} to Azure: {e}")

        print("Sync to Azure completed.")

        self.show\_alert("Sync to Azure completed successfully.")

        self.hide\_loading()

    #Start a thread to clear the Azure database

    def clear\_azure\_database\_threaded(self):

        threading.Thread(target=self.clear\_azure\_database).start()

    def clear\_azure\_database(self):

        #Clear all blobs in Azure container.

        self.show\_loading("Clearing Azure database... please wait.")

        try:

            #Call the clear\_container method to delete all blobs in Azure

            self.azure\_manager.clear\_container()

            print("All images have been cleared from the Azure container.")

            messagebox.showinfo("Azure Database", "All images have been cleared from the Azure database.")

            for widget in self.scrollable\_frame.winfo\_children():

                widget.destroy()

            tk.Label(self.scrollable\_frame, text="Azure directory is now empty.", font=("Helvetica", 12, "bold")).pack(anchor="w", padx=10, pady=10)

        except Exception as e:

            print(f"Failed to clear Azure database: {e}")

            messagebox.showerror("Error", f"Failed to clear Azure database: {e}")

        finally:

            #Hide the loading message

            self.hide\_loading()

    def clear\_azure\_database(self):

        #Call the clear\_container method to delete all blobs in Azure

        self.azure\_manager.clear\_container()

        print("All images have been cleared from the Azure container.")

        messagebox.showinfo("Azure Database", "All images have been cleared from the Azure database.")

        for widget in self.scrollable\_frame.winfo\_children():

            widget.destroy()

        tk.Label(self.scrollable\_frame, text="Azure directory is now empty.", font=("Helvetica", 12, "bold")).pack(anchor="w", padx=10, pady=10)

# %% Run the application

if \_\_name\_\_ == "\_\_main\_\_":

    root = tk.Tk()

    app = SeedIdentifierApp(root)

    root.mainloop()

LOGGER CLASS FILE

import pandas as pd

import logging

from datetime import datetime

import os

#CITATION: GeeksforGeeks. Log File to Pandas Dataframe. https://www.geeksforgeeks.org/log-file-to-pandas-dataframe/

class CSVLogHandler(logging.Handler):

    def \_\_init\_\_(self, log\_file="application\_logs.csv"):

        super().\_\_init\_\_()

        self.log\_file = log\_file

        self.\_initialize\_log\_file()

    def \_initialize\_log\_file(self):

        if not os.path.exists(self.log\_file):

            pd.DataFrame(columns=["Timestamp", "Level", "Message"]).to\_csv(self.log\_file, index=False)

    def log\_to\_dataframe(self, level, message):

        log\_entry = {

            "Timestamp": datetime.now().strftime("%Y-%m-%d %H:%M:%S"),

            "Level": level,

            "Message": message

        }

        df = pd.DataFrame([log\_entry])

        df.to\_csv(self.log\_file, mode='a', index=False, header=False)

    def emit(self, record):

        self.log\_to\_dataframe(record.levelname, record.getMessage())