**Integrating Convolutional and Generative Adversarial Networks with Real-Time Visualization**

**Executive Summary**

This paper details the integration of Convolutional Neural Networks (CNNs) and Generative Adversarial Networks (GANs) within a cohesive machine learning pipeline tailored for grid-based data processing. The system encompasses comprehensive data handling strategies, including data augmentation and preprocessing, alongside robust model training mechanisms for both CNN and GAN architectures. A pivotal aspect of this integration is the development of a real-time visualization tool through a Graphical User Interface (GUI), which facilitates continuous monitoring and analysis of the training process.

A significant challenge addressed in this work is the channel mismatch between the CNN and GAN components, specifically the discrepancy in the number of input channels expected by the Discriminator compared to the real images. This issue was effectively resolved by implementing one-hot encoding and ensuring consistent data dimensions across all components. The resulting system not only enhances performance and interpretability but also ensures seamless interoperability between the CNN and GAN models.

The GUI component plays a crucial role in providing real-time feedback on training metrics such as loss and accuracy, enabling timely interventions and debugging. Additionally, the integration supports scalable training processes, leveraging advanced optimization techniques and mixed-precision training for efficiency. The comprehensive approach outlined in this paper sets a foundation for developing sophisticated machine learning systems capable of handling complex data structures with enhanced performance and reliability.

**Abstract**

This paper presents the integration of Convolutional Neural Networks (CNNs) and Generative Adversarial Networks (GANs) within a unified machine learning pipeline designed for grid-based data processing. The system incorporates data augmentation, model training, and real-time visualization through a graphical user interface (GUI). A significant focus is placed on addressing and resolving channel mismatch issues between the CNN and GAN components. The integration ensures consistent data preprocessing, robust model training, and effective visualization, thereby enhancing the system's performance and interpretability.

**Introduction**

Machine learning models, particularly CNNs and GANs, have demonstrated remarkable capabilities in various domains, including image processing, data generation, and pattern recognition. Integrating these models within a cohesive pipeline leverages their complementary strengths, leading to enhanced performance and versatility. This paper outlines the development of such an integrated system, emphasizing data handling, model architecture, training processes, and real-time visualization. The primary objectives of this integration are:

1. **Data Processing and Augmentation**: Efficiently handle and augment grid-based data to improve model generalization.
2. **Model Integration**: Seamlessly incorporate CNN and GAN models to perform distinct but complementary tasks.
3. **Real-Time Visualization:** Implement a GUI to monitor training progress, facilitating better understanding and debugging.
4. **Error Handling:** Address and resolve channel mismatch issues to ensure smooth training and model interoperability.

**System Architecture**

The system architecture comprises several interconnected components:

1. **Data Loading and Augmentation**: Processes raw grid data, performs augmentation, and prepares it for model training.
2. **CNN Model**: A ResNet18-based convolutional neural network tailored for grid mapping tasks.
3. **GAN Model**: Consists of a Generator and Discriminator, designed to generate realistic grid-based data.
4. **Training Loop**: Manages the training process for both CNN and GAN models, ensuring synchronized updates.
5. **GUI Visualization**: Provides real-time insights into training metrics and model performance.

**Data Handling and Augmentation**

Efficient data handling is pivotal for training robust machine learning models. The system processes grid-based data, ensuring consistency in dimensions and applying augmentation techniques to enhance dataset diversity.

**Data Structures**

The primary data structure used is the GridPair, which encapsulates input and output grids along with a task identifier.

python

from dataclasses import dataclass

import numpy as np

@dataclass

class GridPair:

task\_id: str

input\_grid: np.ndarray

output\_grid: np.ndarray

**Data Loading**

The load\_arc\_data function reads JSON files containing training and evaluation challenges and solutions.

python

import json

import logging

def load\_json\_file(path):

try:

with open(path, 'r') as f:

data = json.load(f)

logging.info(f"Loaded data from {path}.")

return data

except (FileNotFoundError, json.JSONDecodeError) as e:

logging.error(f"Error loading {path}: {e}")

return {}

def load\_arc\_data():

file\_paths = {

"arc-agi\_training-challenges": "arc-agi\_training\_challenges.json",

"arc-agi\_evaluation-challenges": "arc-agi\_evaluation\_challenges.json",

"arc-agi\_training-solutions": "arc-agi\_training\_solutions.json",

"arc-agi\_evaluation-solutions": "arc-agi\_evaluation\_solutions.json",

}

arc\_data = {key: load\_json\_file(path) for key, path in file\_paths.items()}

return arc\_data

**Data Augmentation**

Augmentation techniques are applied to introduce variability, aiding in preventing overfitting and improving model robustness.

python

import numpy as np

import random

from torchvision.transforms import Resize

from PIL import Image

def augment\_grid(grid, noise\_prob=0.2, dead\_square\_prob=0.1):

augmented\_grid = np.array(grid)

if augmented\_grid.ndim != 2:

logging.error(f"Invalid grid shape: {augmented\_grid.shape}")

return augmented\_grid

noise\_mask = np.random.rand(\*augmented\_grid.shape) < noise\_prob

dead\_mask = np.random.rand(\*augmented\_grid.shape) < dead\_square\_prob

noise\_values = np.random.randint(0, NUM\_CLASSES - 1, size=augmented\_grid.shape)

augmented\_grid = np.where(noise\_mask, noise\_values, augmented\_grid)

augmented\_grid = np.where(dead\_mask, -1, augmented\_grid) # Mark as dead squares

return augmented\_grid

def rotate\_grid(grid):

rotations = random.choice([0, 1, 2, 3])

return np.rot90(grid, rotations)

def flip\_grid(grid):

flip\_choice = random.choice(['none', 'vertical', 'horizontal'])

if flip\_choice == 'vertical':

return np.flipud(grid)

elif flip\_choice == 'horizontal':

return np.fliplr(grid)

else:

return grid

**Dataset Class**

The AugmentedARCDataset class handles data loading, resizing, and augmentation.

python

import torch

from torch.utils.data import Dataset

class AugmentedARCDataset(Dataset):

def \_\_init\_\_(self, grid\_pairs, augment=False):

self.grid\_pairs = [

pair for pair in grid\_pairs

if pair.input\_grid.size != 0 and pair.output\_grid.size != 0

and 0 not in pair.input\_grid.shape and 0 not in pair.output\_grid.shape

]

self.augment = augment

self.resize = Resize((FIXED\_HEIGHT, FIXED\_WIDTH))

logging.info(f"Dataset initialized with {len(self.grid\_pairs)} valid grid pairs.")

def \_\_len\_\_(self):

return len(self.grid\_pairs)

def \_\_getitem\_\_(self, idx):

pair = self.grid\_pairs[idx]

input\_grid = pair.input\_grid

output\_grid = pair.output\_grid

if self.augment:

input\_grid = augment\_grid(input\_grid)

# Convert to PIL Images for resizing

input\_image = Image.fromarray(input\_grid.astype(np.uint8))

input\_image = self.resize(input\_image)

input\_grid\_resized = np.array(input\_image)

target\_image = Image.fromarray(output\_grid.astype(np.uint8))

target\_image = self.resize(target\_image)

target\_grid\_resized = np.array(target\_image)

# Convert to tensors

input\_tensor = torch.tensor(input\_grid\_resized, dtype=torch.float32).unsqueeze(0) # [1, H, W]

target\_tensor = torch.tensor(target\_grid\_resized, dtype=torch.long) # [H, W]

if target\_tensor.dim() > 2:

target\_tensor = target\_tensor.squeeze()

return input\_tensor, target\_tensor

**Example**

Consider a grid pair where the input grid is a 10x13 matrix with integer values representing different classes. The augmentation process might add random noise and introduce dead squares, altering the grid to enhance variability.

python

example\_input = np.array([

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1, 2],

# ... (total 10 rows)

])

example\_output = np.array([

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 0, 1, 2],

# ... (total 10 rows)

])

grid\_pair = GridPair(task\_id="task\_001", input\_grid=example\_input, output\_grid=example\_output)

Applying augmentation:

python

augmented\_input = augment\_grid(grid\_pair.input\_grid)

augmented\_input = rotate\_grid(augmented\_input)

augmented\_input = flip\_grid(augmented\_input)

The resulting augmented\_input retains the same dimensions but incorporates noise and potential dead squares, increasing dataset diversity.

**Convolutional Neural Network (CNN) Model**

The CNN model is based on ResNet18, modified to accommodate single-channel input and produce multi-class outputs suitable for grid mapping tasks.

**Model Architecture**

python

import torch.nn as nn

from torchvision.models import resnet18, ResNet18\_Weights

class CNNGridMapper(nn.Module):

def \_\_init\_\_(self, num\_classes=NUM\_CLASSES):

super(CNNGridMapper, self).\_\_init\_\_()

self.num\_classes = num\_classes

# Use ResNet18 backbone

self.cnn = resnet18(weights=ResNet18\_Weights.DEFAULT)

# Modify the first convolutional layer for single-channel input

self.cnn.conv1 = nn.Conv2d(1, 64, kernel\_size=7, stride=2, padding=3, bias=False)

nn.init.kaiming\_normal\_(self.cnn.conv1.weight, mode='fan\_out', nonlinearity='relu')

# Remove the fully connected layer

self.cnn\_layers = nn.Sequential(\*list(self.cnn.children())[:-2])

# Upsampling layers to recover spatial dimensions

self.upsample = nn.Sequential(

nn.ConvTranspose2d(512, 256, kernel\_size=2, stride=2),

nn.ReLU(inplace=True),

nn.ConvTranspose2d(256, 128, kernel\_size=2, stride=2),

nn.ReLU(inplace=True),

nn.ConvTranspose2d(128, num\_classes, kernel\_size=2, stride=2)

)

def forward(self, x):

x = self.cnn\_layers(x)

x = self.upsample(x)

return x # Output shape: [batch\_size, num\_classes, H', W']

**Example**

Given an input tensor of shape [batch\_size, 1, 32, 32], the CNN processes the data through ResNet18 layers, followed by transposed convolutions to output a tensor of shape [batch\_size, 11, 32, 32]. Each channel in the output corresponds to a specific class, facilitating multi-class grid mapping.

python

import torch

# Instantiate the model

model = CNNGridMapper(num\_classes=11)

# Example input

input\_tensor = torch.randn(4, 1, 32, 32) # [batch\_size=4, channels=1, H=32, W=32]

# Forward pass

output = model(input\_tensor)

print(output.shape) # Expected: torch.Size([4, 11, 32, 32])

**Generative Adversarial Network (GAN) Model**

The GAN comprises a Generator and a Discriminator, designed to generate realistic grid-based data and distinguish between real and fake samples, respectively.

**Generator Architecture**

python

class Generator(nn.Module):

def \_\_init\_\_(self, latent\_dim=100, output\_channels=NUM\_CLASSES, grid\_size=(32, 32)):

super(Generator, self).\_\_init\_\_()

self.latent\_dim = latent\_dim

self.output\_channels = output\_channels

self.grid\_size = grid\_size

self.main = nn.Sequential(

# Input: latent\_dim x 1 x 1

nn.ConvTranspose2d(latent\_dim, 512, 4, 1, 0, bias=False),

nn.BatchNorm2d(512),

nn.ReLU(True),

# State size: 512 x 4 x 4

nn.ConvTranspose2d(512, 256, 4, 2, 1, bias=False),

nn.BatchNorm2d(256),

nn.ReLU(True),

# State size: 256 x 8 x 8

nn.ConvTranspose2d(256, 128, 4, 2, 1, bias=False),

nn.BatchNorm2d(128),

nn.ReLU(True),

# State size: 128 x 16 x 16

nn.ConvTranspose2d(128, output\_channels, 4, 2, 1, bias=False),

nn.Tanh()

)

def forward(self, z):

z = z.view(-1, self.latent\_dim, 1, 1)

output = self.main(z)

print(f"Generator output shape: {output.shape}") # Debugging

return output # Expected Output: [batch\_size, 11, 32, 32]

**5.2 Discriminator Architecture**

python

class Discriminator(nn.Module):

def \_\_init\_\_(self, input\_channels=NUM\_CLASSES, grid\_size=(32, 32)):

super(Discriminator, self).\_\_init\_\_()

self.input\_channels = input\_channels

self.grid\_size = grid\_size

self.main = nn.Sequential(

# Input: (input\_channels) x 32 x 32

nn.Conv2d(input\_channels, 64, 4, 2, 1, bias=False),

nn.LeakyReLU(0.2, inplace=True),

# State size: 64 x 16 x 16

nn.Conv2d(64, 128, 4, 2, 1, bias=False),

nn.BatchNorm2d(128),

nn.LeakyReLU(0.2, inplace=True),

# State size: 128 x 8 x 8

nn.Conv2d(128, 256, 4, 2, 1, bias=False),

nn.BatchNorm2d(256),

nn.LeakyReLU(0.2, inplace=True),

# State size: 256 x 4 x 4

nn.Conv2d(256, 1, 4, 1, 0, bias=False),

nn.Sigmoid()

)

def forward(self, x):

return self.main(x).view(-1, 1).squeeze(1)

**Example**

Generating fake images using the Generator and distinguishing them using the Discriminator:

python

# Instantiate GAN models

generator = Generator(latent\_dim=100, output\_channels=11, grid\_size=(32, 32))

discriminator = Discriminator(input\_channels=11, grid\_size=(32, 32))

# Example noise input

noise = torch.randn(4, 100) # [batch\_size=4, latent\_dim=100]

# Generate fake images

fake\_images = generator(noise) # [4, 11, 32, 32]

# Discriminator evaluation

output = discriminator(fake\_images)

print(output) # Output probabilities [4]

**Training Process and Integration**

The training process involves simultaneous training of the CNN and GAN models, ensuring that data consistency and channel alignment are maintained throughout.

**Handling Channel Mismatch**

A critical issue addressed is the channel mismatch between real images (1 channel) and the Discriminator's expectation (11 channels). This is resolved by one-hot encoding the real images to match the required number of channels.

python

# One-hot encode real images

inputs\_one\_hot = torch.nn.functional.one\_hot(real\_images.squeeze(1).long(), num\_classes=NUM\_CLASSES)

inputs\_one\_hot = inputs\_one\_hot.permute(0, 3, 1, 2).float() # [batch\_size, 11, H, W]

# Normalize to [-1, 1]

inputs\_one\_hot = (inputs\_one\_hot - 0.5) \* 2 # Matching Generator's output

**6.2 Training Loop**

The training loop manages both CNN and GAN training within each epoch, ensuring synchronized updates and monitoring.

python

import torch.optim as optim

def train\_thread(self):

# Optimizer and criterion setup

optimizer\_cnn = optim.AdamW(self.model.parameters(), lr=0.01, weight\_decay=1e-4)

scheduler\_cnn = optim.lr\_scheduler.StepLR(optimizer\_cnn, step\_size=10, gamma=0.1)

criterion\_cnn = nn.CrossEntropyLoss()

optimizer\_gen = optim.Adam(self.generator.parameters(), lr=0.0002, betas=(0.5, 0.999))

optimizer\_disc = optim.Adam(self.discriminator.parameters(), lr=0.0002, betas=(0.5, 0.999))

criterion\_gan = nn.BCELoss()

scaler\_cnn = torch.cuda.amp.GradScaler() if self.device.type == 'cuda' else None

scaler\_gan = torch.cuda.amp.GradScaler() if self.device.type == 'cuda' else None

for epoch in range(self.total\_epochs):

if self.stop\_event.is\_set():

break

self.model.train()

self.generator.train()

self.discriminator.train()

running\_loss = 0.0

correct = 0

total = 0

# CNN Training

for batch\_idx, (inputs, targets) in enumerate(self.train\_loader, 1):

if self.stop\_event.is\_set():

break

optimizer\_cnn.zero\_grad()

inputs, targets = inputs.to(self.device), targets.to(self.device)

with torch.cuda.amp.autocast(enabled=True):

outputs = self.model(inputs)

outputs = F.interpolate(outputs, size=targets.shape[1:], mode='bilinear', align\_corners=False)

loss = criterion\_cnn(outputs, targets)

scaler\_cnn.scale(loss).backward()

scaler\_cnn.step(optimizer\_cnn)

scaler\_cnn.update()

running\_loss += loss.item() \* inputs.size(0)

\_, predicted = outputs.max(1)

correct += predicted.eq(targets).sum().item()

total += targets.numel()

if batch\_idx % 10 == 0:

gui\_batch\_loss = running\_loss / total

gui\_batch\_accuracy = 100.0 \* correct / total

self.queue.put({'batch': batch\_idx, 'loss': gui\_batch\_loss, 'accuracy': gui\_batch\_accuracy})

# GAN Training

for batch\_idx, (inputs\_gan, \_) in enumerate(self.train\_loader, 1):

if self.stop\_event.is\_set():

break

real\_images = inputs\_gan.to(self.device)

batch\_size = real\_images.size(0)

label\_real = torch.ones(batch\_size, device=self.device)

label\_fake = torch.zeros(batch\_size, device=self.device)

# One-hot encode and normalize

inputs\_one\_hot = torch.nn.functional.one\_hot(real\_images.squeeze(1).long(), num\_classes=NUM\_CLASSES)

inputs\_one\_hot = inputs\_one\_hot.permute(0, 3, 1, 2).float()

inputs\_one\_hot = (inputs\_one\_hot - 0.5) \* 2

# Train Discriminator

optimizer\_disc.zero\_grad()

with torch.cuda.amp.autocast(enabled=True):

output\_real = self.discriminator(inputs\_one\_hot)

d\_loss\_real = criterion\_gan(output\_real, label\_real)

noise = torch.randn(batch\_size, 100, device=self.device)

fake\_images = self.generator(noise)

output\_fake = self.discriminator(fake\_images.detach())

d\_loss\_fake = criterion\_gan(output\_fake, label\_fake)

d\_loss = d\_loss\_real + d\_loss\_fake

scaler\_gan.scale(d\_loss).backward()

scaler\_gan.step(optimizer\_disc)

scaler\_gan.update()

# Train Generator

optimizer\_gen.zero\_grad()

with torch.cuda.amp.autocast(enabled=True):

output\_fake = self.discriminator(fake\_images)

g\_loss = criterion\_gan(output\_fake, label\_real)

scaler\_gan.scale(g\_loss).backward()

scaler\_gan.step(optimizer\_gen)

scaler\_gan.update()

if batch\_idx % 50 == 0:

logging.info(f"[Epoch {epoch+1}/{self.total\_epochs}] [GAN Batch {batch\_idx}/{len(self.train\_loader)}] [D loss: {d\_loss.item():.4f}] [G loss: {g\_loss.item():.4f}]")

# Scheduler step

scheduler\_cnn.step()

# Epoch metrics

epoch\_loss = running\_loss / total

epoch\_accuracy = 100.0 \* correct / total

self.queue.put({'epoch': epoch + 1, 'loss': epoch\_loss, 'accuracy': epoch\_accuracy})

**Example**

During training, after processing a batch, the system updates the GUI with the current loss and accuracy:

python

if batch\_idx % 10 == 0:

gui\_batch\_loss = running\_loss / total

gui\_batch\_accuracy = 100.0 \* correct / total

self.queue.put({'batch': batch\_idx, 'loss': gui\_batch\_loss, 'accuracy': gui\_batch\_accuracy})

This ensures that users receive real-time feedback on the training progress, facilitating timely interventions if necessary.

**Real-Time Visualization with GUI**

A Tkinter-based GUI provides an interactive platform to monitor training metrics, visualize model performance, and navigate the data hierarchy.

**GUI Components**

1. **Labels**: Display current epoch, batch number, loss, and accuracy.
2. **Data Tree Visualization**: Represents the hierarchical structure of tasks using AnyTree and Graphviz.
3. **Plots:**
   * **2D Plot:** Shows training and validation loss over batches.
   * **3D Plot:** Illustrates the relationship between epochs, accuracy, and prediction error distance.
4. **Control Buttons:** Allow users to start training, stop training, and evaluate the model.

**Example**

The GUI updates in real-time as training progresses. For instance, after processing the 10th batch, the GUI might display:

* **Epoch:** 1/10
* **Batch:** 10/100
* **Loss:** 0.5234
* **Accuracy:** 75.00%

Additionally, the 2D plot will reflect the decrease in loss over batches, while the 3D plot will show the trend of accuracy and prediction error over epochs.

python

import tkinter as tk

from tkinter import ttk, messagebox

from PIL import Image, ImageTk

import matplotlib.pyplot as plt

from matplotlib.backends.backend\_tkagg import FigureCanvasTkAgg

class TrainingGUI:

"""

A Tkinter-based GUI for real-time training progress visualization with 3D metrics plotting and data tree integration.

"""

def \_\_init\_\_(self, root, total\_epochs, total\_batches, model, train\_loader,

val\_loader, eval\_loader, device, data\_tree, task\_dict, generator, discriminator):

"""Initialize the Training GUI."""

self.root = root

self.total\_epochs = total\_epochs

self.total\_batches = total\_batches

self.model = model

self.train\_loader = train\_loader

self.val\_loader = val\_loader

self.eval\_loader = eval\_loader

self.device = device

self.data\_tree = data\_tree # Data tree integration

self.task\_dict = task\_dict # Store task dictionary for training logic

# GAN models

self.generator = generator

self.discriminator = discriminator

# Initialize other required attributes

self.queue = queue.Queue()

self.stop\_event = threading.Event()

# Initialize data storage for plots

self.loss\_data = []

self.val\_loss\_data = []

self.acc\_data = []

self.prediction\_distances = []

# Set up the GUI

self.setup\_gui()

self.root.after(100, self.process\_queue)

def setup\_gui(self):

"""Set up the GUI components."""

self.frame = tk.Frame(self.root)

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

# Top Section for Labels

self.label\_frame = tk.Frame(self.frame)

self.label\_frame.pack(pady=10)

self.epoch\_label = tk.Label(self.label\_frame, text=f"Epoch: 0/{self.total\_epochs}", font=("Helvetica", 14))

self.epoch\_label.grid(row=0, column=0, padx=10)

self.batch\_label = tk.Label(self.label\_frame, text=f"Batch: 0/{self.total\_batches}", font=("Helvetica", 12))

self.batch\_label.grid(row=0, column=1, padx=10)

self.loss\_label = tk.Label(self.label\_frame, text="Loss: 0.0000", font=("Helvetica", 12))

self.loss\_label.grid(row=0, column=2, padx=10)

self.accuracy\_label = tk.Label(self.label\_frame, text="Accuracy: 0.0000", font=("Helvetica", 12))

self.accuracy\_label.grid(row=0, column=3, padx=10)

# Data Tree Visualization Section

self.tree\_frame = tk.Frame(self.frame, width=300, height=400)

self.tree\_frame.pack(side=tk.LEFT, padx=10, pady=10, fill=tk.Y)

self.tree\_label = tk.Label(self.tree\_frame, text="Data Tree", font=("Helvetica", 14))

self.tree\_label.pack()

self.tree\_canvas = tk.Canvas(self.tree\_frame, width=300, height=400, bg='white')

self.tree\_canvas.pack()

# Display the data tree

self.display\_data\_tree()

# Plot Section (2D + 3D)

self.fig = plt.figure(figsize=(12, 6))

# 3D Plot on the Left

self.ax\_3d = self.fig.add\_subplot(121, projection='3d')

self.ax\_3d.set\_xlabel('Epoch')

self.ax\_3d.set\_ylabel('Accuracy')

self.ax\_3d.set\_zlabel('Distance from Actual')

# 2D Plot on the Right

self.ax\_2d = self.fig.add\_subplot(122)

self.line\_loss, = self.ax\_2d.plot([], [], label='Training Loss')

self.line\_val\_loss, = self.ax\_2d.plot([], [], label='Validation Loss')

self.ax\_2d.legend()

self.canvas\_plot = FigureCanvasTkAgg(self.fig, master=self.frame)

self.canvas\_plot.draw()

self.canvas\_plot.get\_tk\_widget().pack(side=tk.TOP, fill=tk.BOTH, expand=True)

# Bottom Section for Control Buttons

self.button\_frame = tk.Frame(self.frame)

self.button\_frame.pack(pady=10)

self.start\_button = tk.Button(self.button\_frame, text="Start Training", command=self.start\_training)

self.start\_button.grid(row=0, column=0, padx=10)

self.stop\_button = tk.Button(self.button\_frame, text="Stop Training", command=self.stop\_training)

self.stop\_button.grid(row=0, column=1, padx=10)

self.evaluate\_button = tk.Button(self.button\_frame, text="Evaluate Model", command=self.evaluate\_model\_button)

self.evaluate\_button.grid(row=0, column=2, padx=10)

def display\_data\_tree(self):

"""Generate and display the data tree as an image."""

try:

# Render the tree to a PNG using anytree and Graphviz

dot\_file = "tree.dot"

png\_file = "tree.png"

# Export to .dot file

DotExporter(self.data\_tree).to\_dotfile(dot\_file)

logging.info(f"Tree exported to {dot\_file}")

# Convert .dot to .png using Graphviz

result = os.system(f'dot -Tpng {dot\_file} -o {png\_file}')

if result != 0:

raise RuntimeError("Failed to generate PNG. Ensure Graphviz is installed and in PATH.")

# Load and display the PNG image

img = Image.open(png\_file)

img = img.resize((300, 400), Image.LANCZOS)

img\_tk = ImageTk.PhotoImage(img)

# Display the image in the canvas

self.tree\_canvas.create\_image(0, 0, anchor=tk.NW, image=img\_tk)

self.tree\_canvas.image = img\_tk # Keep reference to avoid garbage collection

logging.info("Tree visualization displayed successfully.")

except Exception as e:

logging.exception("Failed to display the data tree.")

messagebox.showerror("Tree Display Error", f"Error: {e}")

def process\_queue(self):

"""Process the queue for thread-safe GUI updates."""

while not self.queue.empty():

message = self.queue.get()

if isinstance(message, dict):

self.update\_gui(message)

self.root.after(100, self.process\_queue)

def update\_gui(self, data):

"""Update the GUI with real-time training and validation metrics."""

try:

if 'batch' in data:

# Update batch-level metrics in the GUI

self.batch\_label.config(text=f"Batch: {data['batch']}/{self.total\_batches}")

self.loss\_label.config(text=f"Loss: {data['loss']:.4f}")

self.accuracy\_label.config(text=f"Accuracy: {data.get('accuracy', 0.0):.4f}")

# Append new batch data to the 2D plot lists

self.loss\_data.append(data['loss'])

self.acc\_data.append(data.get('accuracy', 0.0))

# Update the 2D plot with each batch completion

batches = list(range(1, len(self.loss\_data) + 1))

self.line\_loss.set\_data(batches, self.loss\_data)

self.line\_val\_loss.set\_data(batches, self.acc\_data)

# Adjust the axes to fit the new data

self.ax\_2d.relim()

self.ax\_2d.autoscale\_view()

# Redraw the 2D plot with new data

self.canvas\_plot.draw()

elif 'epoch' in data:

# Update epoch-level metrics

self.epoch\_label.config(text=f"Epoch: {data['epoch']}/{self.total\_epochs}")

# Calculate prediction error distance

predicted = np.array(data.get('guesses', []))

actual = np.array(data.get('actuals', []))

if predicted.size == 0 or actual.size == 0:

distance = float('nan') # Handle empty arrays gracefully

elif predicted.shape != actual.shape:

distance = float('nan') # Handle shape mismatch

else:

distance = np.abs(predicted - actual).mean()

# Replace NaN with 0.0 for plotting purposes

distance = 0.0 if np.isnan(distance) else distance

# Store valid distances for 3D plot

self.prediction\_distances.append(distance)

# Update the 3D plot

epochs = list(range(1, len(self.prediction\_distances) + 1))

self.ax\_3d.clear()

self.ax\_3d.set\_xlabel('Epoch')

self.ax\_3d.set\_ylabel('Accuracy')

self.ax\_3d.set\_zlabel('Distance from Actual')

self.ax\_3d.set\_title('3D Prediction Error vs Accuracy')

# Scatter plot with prediction distances

self.ax\_3d.scatter(epochs, self.acc\_data, self.prediction\_distances, label='Error vs Accuracy', color='green')

self.ax\_3d.legend()

# Redraw the 3D plot

self.canvas\_plot.draw()

except Exception as e:

logging.exception("An error occurred while updating the GUI.")

messagebox.showerror("Error", f"An error occurred: {e}")

def start\_training(self):

"""Start training in a new thread."""

self.stop\_event.clear()

threading.Thread(target=self.train\_thread, daemon=True).start()

def stop\_training(self):

"""Stop the training process."""

self.stop\_event.set()

def train\_thread(self):

"""Training logic executed in a separate thread to avoid blocking the GUI."""

logging.info("Training thread started.")

# Optimizer and criterion setup

optimizer\_cnn = optim.AdamW(self.model.parameters(), lr=0.01, weight\_decay=1e-4)

scheduler\_cnn = optim.lr\_scheduler.StepLR(optimizer\_cnn, step\_size=10, gamma=0.1)

criterion\_cnn = nn.CrossEntropyLoss()

optimizer\_gen = optim.Adam(self.generator.parameters(), lr=0.0002, betas=(0.5, 0.999))

optimizer\_disc = optim.Adam(self.discriminator.parameters(), lr=0.0002, betas=(0.5, 0.999))

criterion\_gan = nn.BCELoss()

scaler\_cnn = torch.cuda.amp.GradScaler() if self.device.type == 'cuda' else None

scaler\_gan = torch.cuda.amp.GradScaler() if self.device.type == 'cuda' else None

for epoch in range(self.total\_epochs):

if self.stop\_event.is\_set():

logging.info("Training stopped by user.")

break

logging.info(f"Starting epoch {epoch + 1}/{self.total\_epochs}.")

self.model.train()

self.generator.train()

self.discriminator.train()

running\_loss = 0.0

correct = 0

total = 0

# CNN Training

for batch\_idx, (inputs, targets) in enumerate(self.train\_loader, 1):

if self.stop\_event.is\_set():

logging.info("Training stopped by user.")

break

optimizer\_cnn.zero\_grad()

inputs, targets = inputs.to(self.device), targets.to(self.device)

with torch.cuda.amp.autocast(enabled=True):

outputs = self.model(inputs)

outputs = F.interpolate(outputs, size=targets.shape[1:], mode='bilinear', align\_corners=False)

loss = criterion\_cnn(outputs, targets)

scaler\_cnn.scale(loss).backward()

scaler\_cnn.step(optimizer\_cnn)

scaler\_cnn.update()

running\_loss += loss.item() \* inputs.size(0)

\_, predicted = outputs.max(1)

correct += predicted.eq(targets).sum().item()

total += targets.numel()

if batch\_idx % 10 == 0:

gui\_batch\_loss = running\_loss / total

gui\_batch\_accuracy = 100.0 \* correct / total

self.queue.put({'batch': batch\_idx, 'loss': gui\_batch\_loss, 'accuracy': gui\_batch\_accuracy})

# GAN Training

for batch\_idx, (inputs\_gan, \_) in enumerate(self.train\_loader, 1):

if self.stop\_event.is\_set():

logging.info("Training stopped by user.")

break

real\_images = inputs\_gan.to(self.device)

batch\_size = real\_images.size(0)

label\_real = torch.ones(batch\_size, device=self.device)

label\_fake = torch.zeros(batch\_size, device=self.device)

# One-hot encode and normalize

inputs\_one\_hot = torch.nn.functional.one\_hot(real\_images.squeeze(1).long(), num\_classes=NUM\_CLASSES)

inputs\_one\_hot = inputs\_one\_hot.permute(0, 3, 1, 2).float()

inputs\_one\_hot = (inputs\_one\_hot - 0.5) \* 2

# Train Discriminator

optimizer\_disc.zero\_grad()

with torch.cuda.amp.autocast(enabled=True):

output\_real = self.discriminator(inputs\_one\_hot)

d\_loss\_real = criterion\_gan(output\_real, label\_real)

noise = torch.randn(batch\_size, 100, device=self.device)

fake\_images = self.generator(noise)

output\_fake = self.discriminator(fake\_images.detach())

d\_loss\_fake = criterion\_gan(output\_fake, label\_fake)

d\_loss = d\_loss\_real + d\_loss\_fake

scaler\_gan.scale(d\_loss).backward()

scaler\_gan.step(optimizer\_disc)

scaler\_gan.update()

# Train Generator

optimizer\_gen.zero\_grad()

with torch.cuda.amp.autocast(enabled=True):

output\_fake = self.discriminator(fake\_images)

g\_loss = criterion\_gan(output\_fake, label\_real)

scaler\_gan.scale(g\_loss).backward()

scaler\_gan.step(optimizer\_gen)

scaler\_gan.update()

if batch\_idx % 50 == 0:

logging.info(f"[Epoch {epoch+1}/{self.total\_epochs}] [GAN Batch {batch\_idx}/{len(self.train\_loader)}] [D loss: {d\_loss.item():.4f}] [G loss: {g\_loss.item():.4f}]")

# Scheduler step

scheduler\_cnn.step()

# Epoch metrics

epoch\_loss = running\_loss / total

epoch\_accuracy = 100.0 \* correct / total

self.queue.put({'epoch': epoch + 1, 'loss': epoch\_loss, 'accuracy': epoch\_accuracy})

logging.info("Training completed.")

self.queue.put({'status': 'Training Completed'})

def evaluate\_model\_button(self):

"""Evaluate the model in a new thread."""

threading.Thread(target=self.evaluate\_model, daemon=True).start()

def evaluate\_model(self):

"""Evaluate the model."""

avg\_loss, accuracy = evaluate\_model(self.model, self.val\_loader, self.device)

messagebox.showinfo("Evaluation", f"Validation Loss: {avg\_loss:.4f}, Accuracy: {accuracy:.4f}")

**Example**

During training, after processing a batch, the system updates the GUI with the current loss and accuracy:

python

if batch\_idx % 10 == 0:

gui\_batch\_loss = running\_loss / total

gui\_batch\_accuracy = 100.0 \* correct / total

self.queue.put({'batch': batch\_idx, 'loss': gui\_batch\_loss, 'accuracy': gui\_batch\_accuracy})

This ensures that users receive real-time feedback on the training progress, facilitating timely interventions if necessary.

**Addressing Channel Mismatch Issue**

A critical challenge encountered was the channel mismatch between real images (1 channel) and the Discriminator's expectation (11 channels). This issue was systematically resolved through the following steps:

**Problem Identification**

The error message:

vbnet

RuntimeError: Given groups=1, weight of size [64, 11, 4, 4], expected input[4, 1, 10, 13] to have 11 channels, but got 1 channels instead

indicated a mismatch in the number of input channels between the real images and the Discriminator's first convolutional layer.

**Solution Implementation**

1. **One-Hot Encoding Real Images**:
   * Real images were converted from single-channel to multi-channel (11 channels) using one-hot encoding.

python

# One-hot encode real images

inputs\_one\_hot = torch.nn.functional.one\_hot(real\_images.squeeze(1).long(), num\_classes=NUM\_CLASSES)

inputs\_one\_hot = inputs\_one\_hot.permute(0, 3, 1, 2).float() # [batch\_size, 11, H, W]

1. **Normalization**:
   * The one-hot encoded images were normalized to match the Generator's output range.

python

inputs\_one\_hot = (inputs\_one\_hot - 0.5) \* 2 # Normalize to [-1, 1]

1. **Generator Configuration**:
   * Ensured that the Generator produces images with 11 channels and spatial dimensions of 32x32.

python

class Generator(nn.Module):

# Initialization and forward method as previously defined

1. **Consistent Data Dimensions**:
   * All images, both real and fake, were resized to 32x32 pixels to maintain uniform spatial dimensions.

python

self.resize = Resize((FIXED\_HEIGHT, FIXED\_WIDTH))

**Outcome**

By implementing one-hot encoding and ensuring consistent spatial dimensions, the channel mismatch issue was effectively resolved. The Discriminator now receives inputs with the expected number of channels, enabling seamless training without runtime errors.

**Conclusion**

This integrated system successfully combines CNN and GAN models within a unified pipeline, addressing key challenges such as data augmentation, channel mismatch, and real-time visualization. The incorporation of a GUI facilitates enhanced monitoring and interpretability of the training process, while systematic error handling ensures robustness. Future enhancements could explore more advanced augmentation techniques, model architectures, and visualization features to further elevate the system's capabilities.

**References**

1. He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep Residual Learning for Image Recognition. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.
2. Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., ... & Bengio, Y. (2014). Generative Adversarial Nets. *Advances in Neural Information Processing Systems (NeurIPS)*.
3. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., ... & Chintala, S. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems (NeurIPS)*.