Hierarchical Autoencoder (HAE) Report

1. Hierarchical autoencoder

1. Introduction

The Hierarchical Autoencoder (HAE) is a sophisticated deep learning model designed for hierarchical feature learning and reconstruction. This report provides a comprehensive description of the HAE's functioning, focusing on both training and inference processes. Particular attention is given to the hierarchical structure, elucidating the functions invoked during the training of the model.

2. HAE Architecture Overview

The HAE is structured as an encoder-decoder model, featuring multiple layers that progressively learn and represent features at different levels of abstraction. Below is an in-depth exploration of the HAE's functions during training and inference, highlighting the ordered sequence of functions invoked.

3. Training Process

Data Loading:

The training process commences with the loading of the training dataset, such as ImageNet100, containing images across various classes.

Encoder Training (1st Level):

The initial training phase focuses on the bottom-level encoder, which takes raw image data as input.

The encoder's role is to extract low-level features, including edges, textures, and fundamental shapes.

Reconstruction (1st Level):

Following the encoder training, the model engages in the reconstruction phase.

The decoder attempts to reconstruct the original input from the encoded features, refining the learned representations.

Higher-Level Encoders (2nd Level and Beyond):

Subsequent training phases introduce higher-level encoders to the model.

Each higher-level encoder takes the output of the previous level, capturing more abstract features.

The hierarchical training process iterates until all desired levels are trained.

Hierarchical Training:

The hierarchical structure is a cornerstone of the training process.

Each level of the HAE learns features that build upon the representations acquired by the preceding levels.

This enables the model to discern complex hierarchical relationships within the data.

4. Inference Process

During inference, the trained HAE model operates in a hierarchical manner:

Encoding:

Input data, typically images, is passed through the trained encoders at each level.

Each encoder extracts hierarchical features, progressively capturing more abstract information.

Decoding:

The encoded representations traverse through the corresponding decoders to reconstruct the input.

This process generates a reconstructed version of the original input based on the learned hierarchical features.

5. Training the HAE Model on the 2nd Level

The train_hae.py script illustrates the training process on the 2nd level. This script encompasses the following steps:

Initialization of the model with the bottom-level encoder trained in previous steps.

A training loop that includes both encoder and decoder training for the 2nd level.

Refinement of the model's ability to capture higher-level features and improve representations learned at the 1st level.

The training of the 2nd level involves systematic optimization of the model's parameters, enhancing its capacity to encode and reconstruct features at a more abstract level.

6. Workflow (Libraries and functions used):

- **Data Loading and Preparation**

load_datasets.py: The load_datasets.py module focuses on loading and preparing datasets, primarily the ImageNet100 dataset.

Key functions include:

_make_train_valid_split: Splits the training dataset into training and validation subsets, considering a specified test size.

_make_data_loaders: Creates PyTorch DataLoader instances for training, validation, and testing. Handles random splitting of the training data for validation.

load_ImageNet100: The main function loads the ImageNet100 dataset and returns data loaders, allowing seamless integration into the training pipeline.

- **HAE Model Training**

train_hae.py: The train_hae.py module orchestrates the training of the HAE model layer by layer.

Significant functions include:

load_hae_from_checkpoints: Loads the HAE model layer by layer from previously saved checkpoints.

get_dataloader: Creates data loaders for either training or validation, depending on the specified mode.

train_full_stack: Trains the HAE model layer by layer, stacking encoders and decoders. Utilizes PyTorch Lightning for efficient training.

train: The main function initiates and trains the HAE model on the ImageNet100 dataset. It handles the overall training pipeline, creating necessary directories for checkpoints if they do not exist.

- **HAE Model Testing**

test_hae.py: The test_hae.py script is dedicated to testing the HAE model.

Key functionalities include:

load_hae_from_checkpoints: Loads the trained HAE model from previously saved checkpoints.

Reconstruction and Evaluation: Evaluates and prints the accuracy of the reconstructed images at each layer of the hierarchical autoencoder.

Adversarial Attack and Testing: Applies an adversarial attack (Fast Gradient Sign Method - FGSM) to the ResNet model. The script then tests the accuracy after the attack for each layer.

- **ResNet Model Implementation**

resnet_I_50.py: The resnet_I_50.py module implements a specific ResNet50-based classifier, resnetI50, tailored for ImageNet datasets.

This class extends the base class PyTorch_ResnetI and defines methods for various model components:

_define_resnet_body: Implements the ResNet50-based body for feature extraction.

_define_linear_layers: Defines linear layers for classification.

_define_optimizer: Specifies the optimizer (Stochastic Gradient Descent - SGD) for model training.

_define_loss_function: Defines the loss function (CrossEntropyLoss) for training.

- **Evaluation and Analysis**

evaluate_recons.py: The evaluate_recons.py module focuses on evaluating the models and conducting in-depth analysis.

Key functionalities include:

evaluate_dataset: Evaluates the accuracy of model predictions, generates confusion matrices, and optionally saves the evaluation results.

sample_by_class: Samples images by class, providing a nuanced understanding of model performance.

7. Conclusion

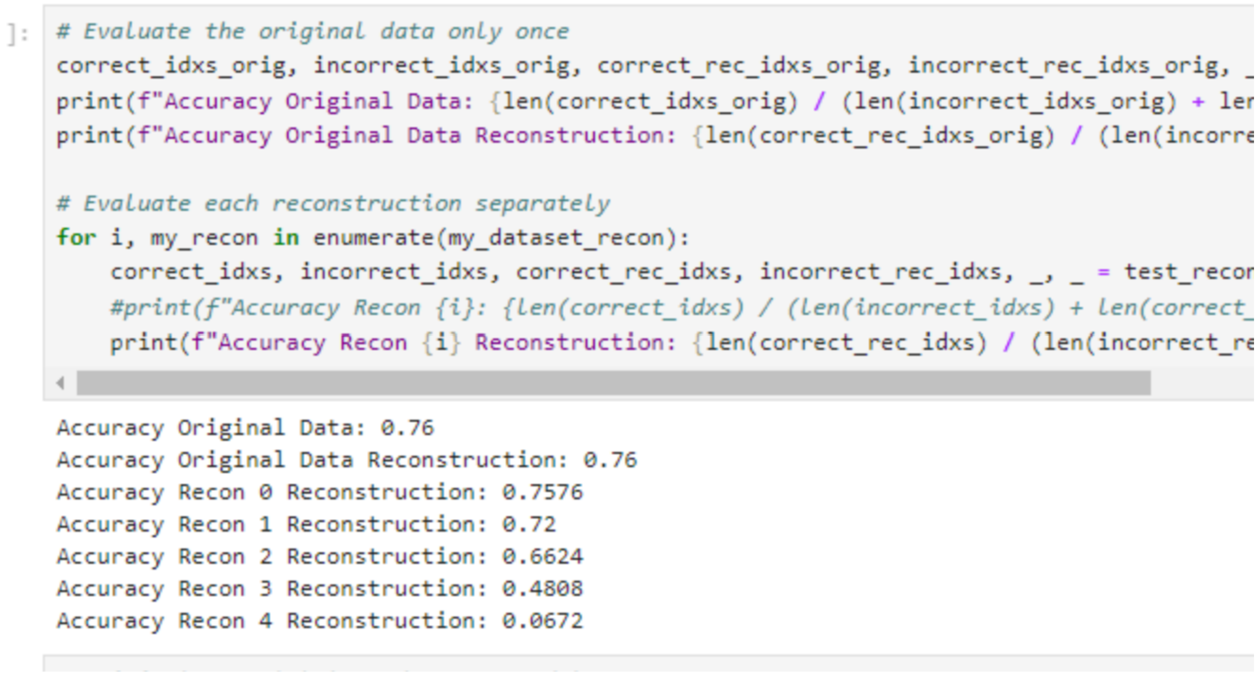
In conclusion, the Hierarchical Autoencoder represents a powerful approach to hierarchical feature learning and reconstruction. Its hierarchical architecture allows for the progressive understanding of intricate patterns within complex datasets. This provides a thorough overview of the HAE's functioning, emphasizing its hierarchical nature and the ordered sequence of functions invoked during training and inference. Gaining insights into these processes is pivotal for leveraging the full capabilities of the HAE in diverse applications.

2. Reference Classifier

File: updated_test_hae.ipynb

1. **Evaluate accuracies of the reconstructions on a reference classifier (for different levels).**

Code snippet:



```
]: # Evaluate the original data only once
   correct_idxs_orig, incorrect_idxs_orig, correct_rec_idxs_orig, incorrect_rec_idxs_orig, _
   print(f"Accuracy Original Data: {len(correct_idxs_orig) / (len(incorrect_idxs_orig) + len
   print(f"Accuracy Original Data Reconstruction: {len(correct_rec_idxs_orig) / (len(incorre

   # Evaluate each reconstruction separately
   for i, my_recon in enumerate(my_dataset_recon):
       correct_idxs, incorrect_idxs, correct_rec_idxs, incorrect_rec_idxs, _, _ = test_recon
       #print(f"Accuracy Recon {i}: {len(correct_idxs) / (len(incorrect_idxs) + len(correct_
       print(f"Accuracy Recon {i} Reconstruction: {len(correct_rec_idxs) / (len(incorrect_re
```

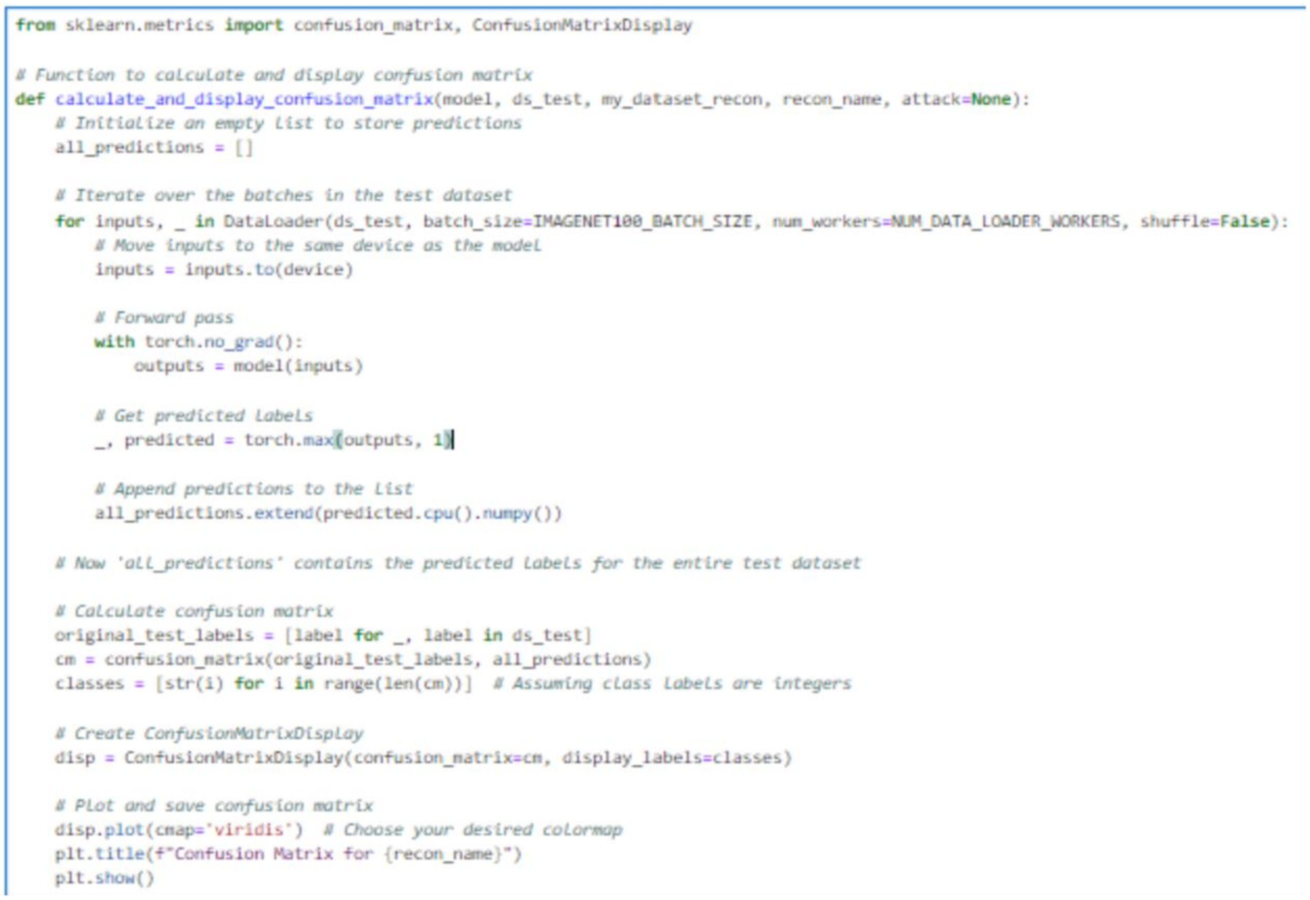
```
Accuracy Original Data: 0.76
Accuracy Original Data Reconstruction: 0.76
Accuracy Recon 0 Reconstruction: 0.7576
Accuracy Recon 1 Reconstruction: 0.72
Accuracy Recon 2 Reconstruction: 0.6624
Accuracy Recon 3 Reconstruction: 0.4808
Accuracy Recon 4 Reconstruction: 0.0672
```

This evaluation strategy allows to assess the model's performance on both the original data and the reconstructed data from each layer separately. It helps in understanding how well the model is performing on the original data and how the reconstruction quality evolves across different layers. The loops are created in such a way that original accuracy is not evaluated many times here.

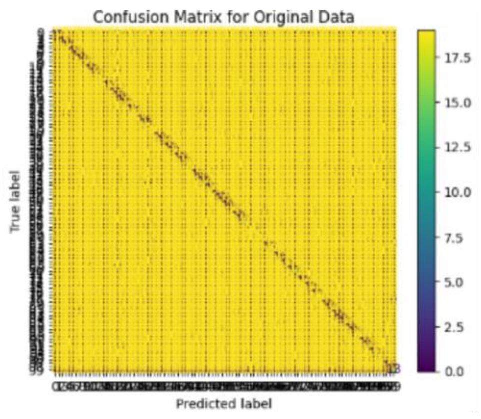
2. **add the confusion matrix**

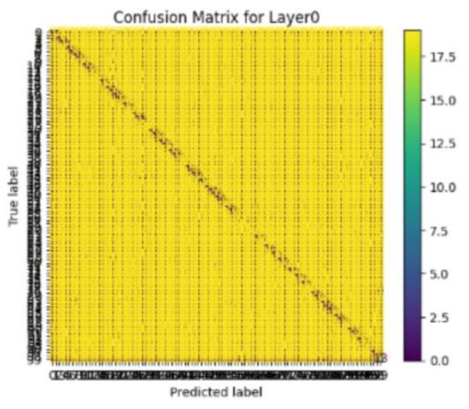
File: updated_test_hae.ipynb

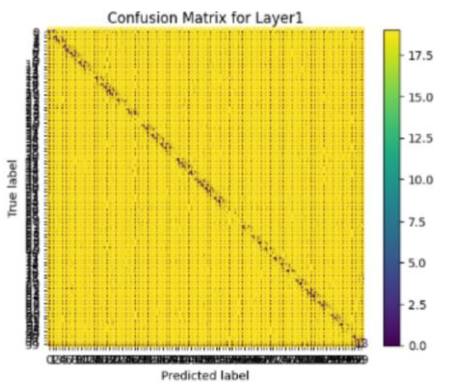
Code snippet:

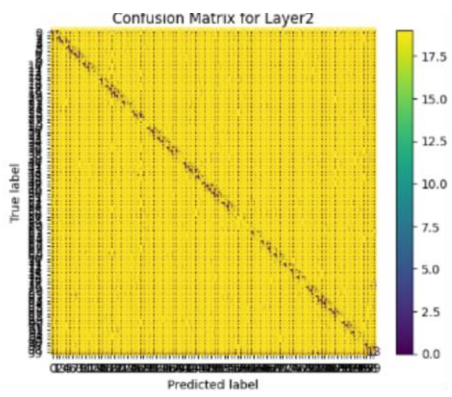


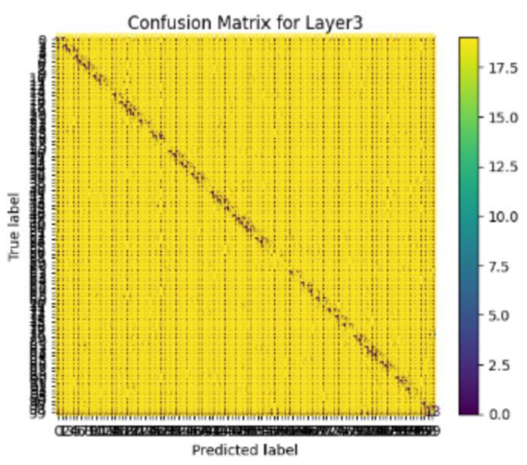
```python
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

# Function to calculate and display confusion matrix
def calculate_and_display_confusion_matrix(model, ds_test, my_dataset_recon, recon_name, attack=None):
    # Initialize an empty list to store predictions
    all_predictions = []

    # Iterate over the batches in the test dataset
    for inputs, _ in DataLoader(ds_test, batch_size=IMAGENET100_BATCH_SIZE, num_workers=NUM_DATA_LOADER_WORKERS, shuffle=False):
        # Move inputs to the same device as the model
        inputs = inputs.to(device)

        # Forward pass
        with torch.no_grad():
            outputs = model(inputs)

        # Get predicted labels
        _, predicted = torch.max(outputs, 1)

        # Append predictions to the list
        all_predictions.extend(predicted.cpu().numpy())

    # Now 'all_predictions' contains the predicted labels for the entire test dataset

    # Calculate confusion matrix
    original_test_labels = [label for _, label in ds_test]
    cm = confusion_matrix(original_test_labels, all_predictions)
    classes = [str(i) for i in range(len(cm))]  # Assuming class labels are integers

    # Create ConfusionMatrixDisplay
    disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=classes)

    # Plot and save confusion matrix
    disp.plot(cmap='viridis')  # Choose your desired colormap
    plt.title(f"Confusion Matrix for {recon_name}")
    plt.show()
```

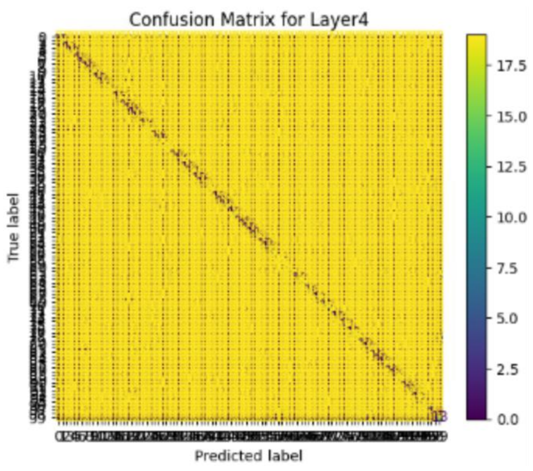












- **Function Definition:**

The function takes the following parameters:

model: The trained PyTorch model.

ds_test: The test dataset.

my_dataset_recon: A set of reconstructed datasets.

recon_name: A string representing the name of the reconstruction (e.g., "Original Data" or "Layer{i}").

attack: An optional parameter representing the attack to be applied during evaluation.

- **Predictions for the Test Dataset:**

The function uses a loop to iterate over batches in the test dataset (ds_test).

The model's predictions are obtained through a forward pass (outputs = model(inputs)), and the predicted labels are extracted.

- **Confusion Matrix Calculation:**

The true labels for the test dataset (original_test_labels) are collected.

The confusion matrix is calculated using sklearn.metrics.confusion_matrix.

ConfusionMatrixDisplay is then used to visualize the confusion matrix.

- **Display Confusion Matrices:**

The function is called twice:

First, for the original test dataset (calculate_and_display_confusion_matrix(model, ds_test, ds_test, "Original Data", attack=None)).

Second, for each reconstruction in my_dataset_recon (for i, my_recon in enumerate(my_dataset_recon)).

- **Visualization:**

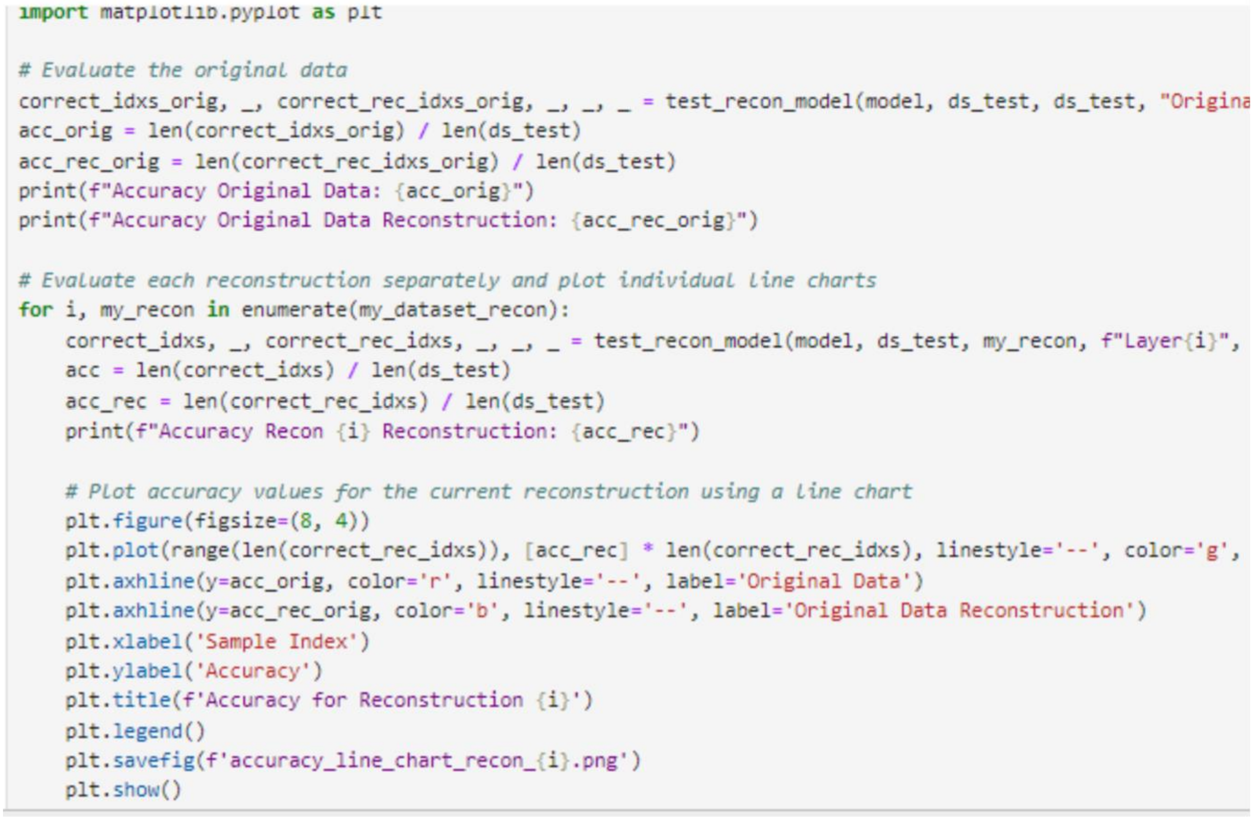
The confusion matrix is displayed using ConfusionMatrixDisplay, and the title reflects whether it is for the original data or a specific reconstruction.

This code allows for a comprehensive analysis of the model's performance on both the original test dataset and various reconstructed datasets, helping to identify how well the model generalizes and whether it is robust to different types of reconstructions.

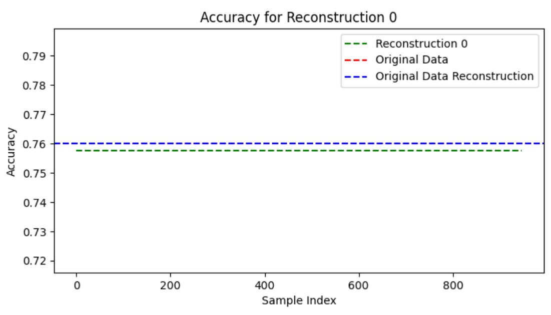
3. **Creating and saving the accuracy plots**

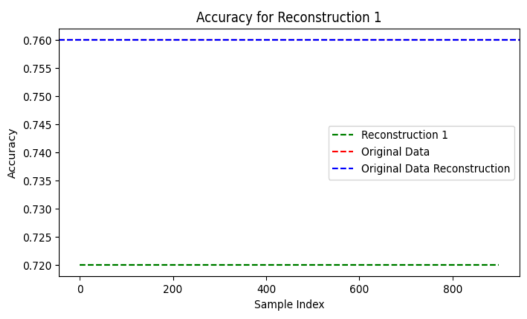
   File: updated_test_hae.ipynb

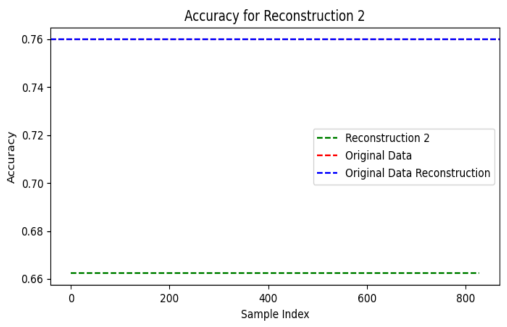
Code snippet:

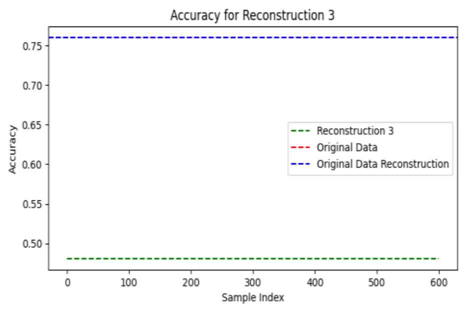


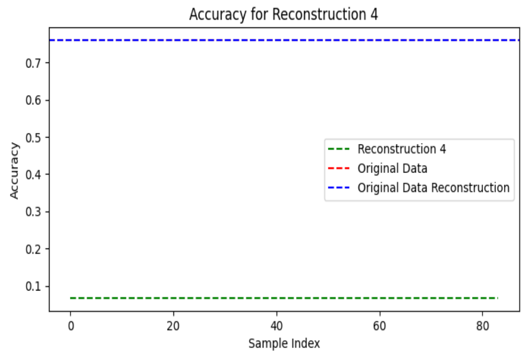
```
import matplotlib.pyplot as plt

# Evaluate the original data
correct_idxs_orig, _, correct_rec_idxs_orig, _, _, _ = test_recon_model(model, ds_test, ds_test, "Origina
acc_orig = len(correct_idxs_orig) / len(ds_test)
acc_rec_orig = len(correct_rec_idxs_orig) / len(ds_test)
print(f"Accuracy Original Data: {acc_orig}")
print(f"Accuracy Original Data Reconstruction: {acc_rec_orig}")

# Evaluate each reconstruction separately and plot individual line charts
for i, my_recon in enumerate(my_dataset_recon):
    correct_idxs, _, correct_rec_idxs, _, _, _ = test_recon_model(model, ds_test, my_recon, f"Layer{i}",
    acc = len(correct_idxs) / len(ds_test)
    acc_rec = len(correct_rec_idxs) / len(ds_test)
    print(f"Accuracy Recon {i} Reconstruction: {acc_rec}")

    # Plot accuracy values for the current reconstruction using a line chart
    plt.figure(figsize=(8, 4))
    plt.plot(range(len(correct_rec_idxs)), [acc_rec] * len(correct_rec_idxs), linestyle='--', color='g',
    plt.axhline(y=acc_orig, color='r', linestyle='--', label='Original Data')
    plt.axhline(y=acc_rec_orig, color='b', linestyle='--', label='Original Data Reconstruction')
    plt.xlabel('Sample Index')
    plt.ylabel('Accuracy')
    plt.title(f'Accuracy for Reconstruction {i}')
    plt.legend()
    plt.savefig(f'accuracy_line_chart_recon_{i}.png')
    plt.show()
```

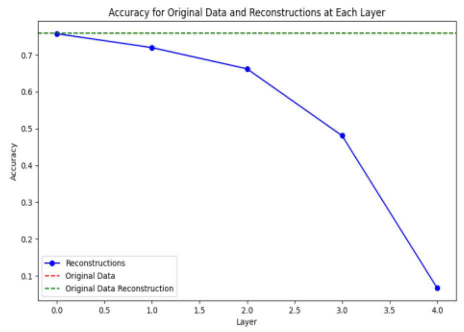












Iterates over each reconstruction in my_dataset_recon.

Calls the test_recon_model function to evaluate the accuracy of the model on the test dataset with the current reconstruction (my_recon) and without any attack.

Calculates and prints the accuracy for each reconstruction.

Creates a separate line chart for each reconstruction using Matplotlib.

- **In each line chart:**

The x-axis represents the index of the samples.

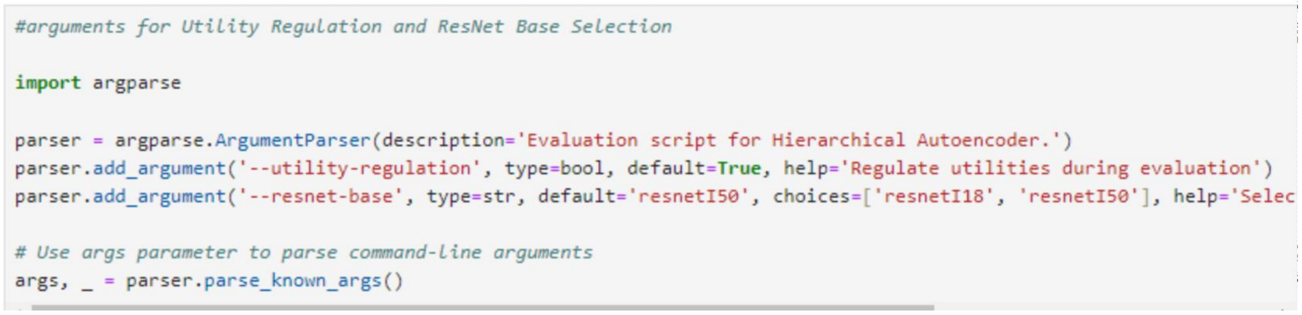
The y-axis represents the accuracy.

Also, created another code, which will give all reconstructed and original data accuracies in one plot. Which clearly shows that original data accuracy is high comparatively

4. **creation of arguments that regulate these utilities and selection of the resnet base for the reference model.**

   File: answer.py, train_resnet_lightning.py

Code snippet:



```
#arguments for Utility Regulation and ResNet Base Selection

import argparse

parser = argparse.ArgumentParser(description='Evaluation script for Hierarchical Autoencoder.')
parser.add_argument('--utility-regulation', type=bool, default=True, help='Regulate utilities during evaluation')
parser.add_argument('--resnet-base', type=str, default='resnetI50', choices=['resnetI18', 'resnetI50'], help='Selec

# Use args parameter to parse command-line arguments
args, _ = parser.parse_known_args()
```

- **Import argparse:** Import the module for handling command-line arguments.
- **Define Arguments:**

--utility-regulation: Boolean (default True).

--resnet-base: String, choices: 'resnetI18', 'resnetI50' (default 'resnetI50').

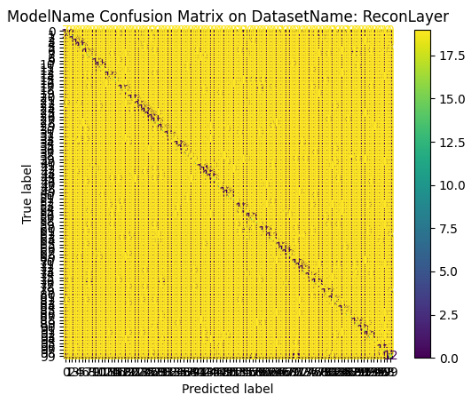
- **Parse Arguments:** args, _ = parser.parse_known_args() to get argument values.
- **Access Values:** Access values using args.utility_regulation and args.resnet_base.
- **Command:** python train_resnet_lightning.py --utility-regulation True --resnet-base resnetI50

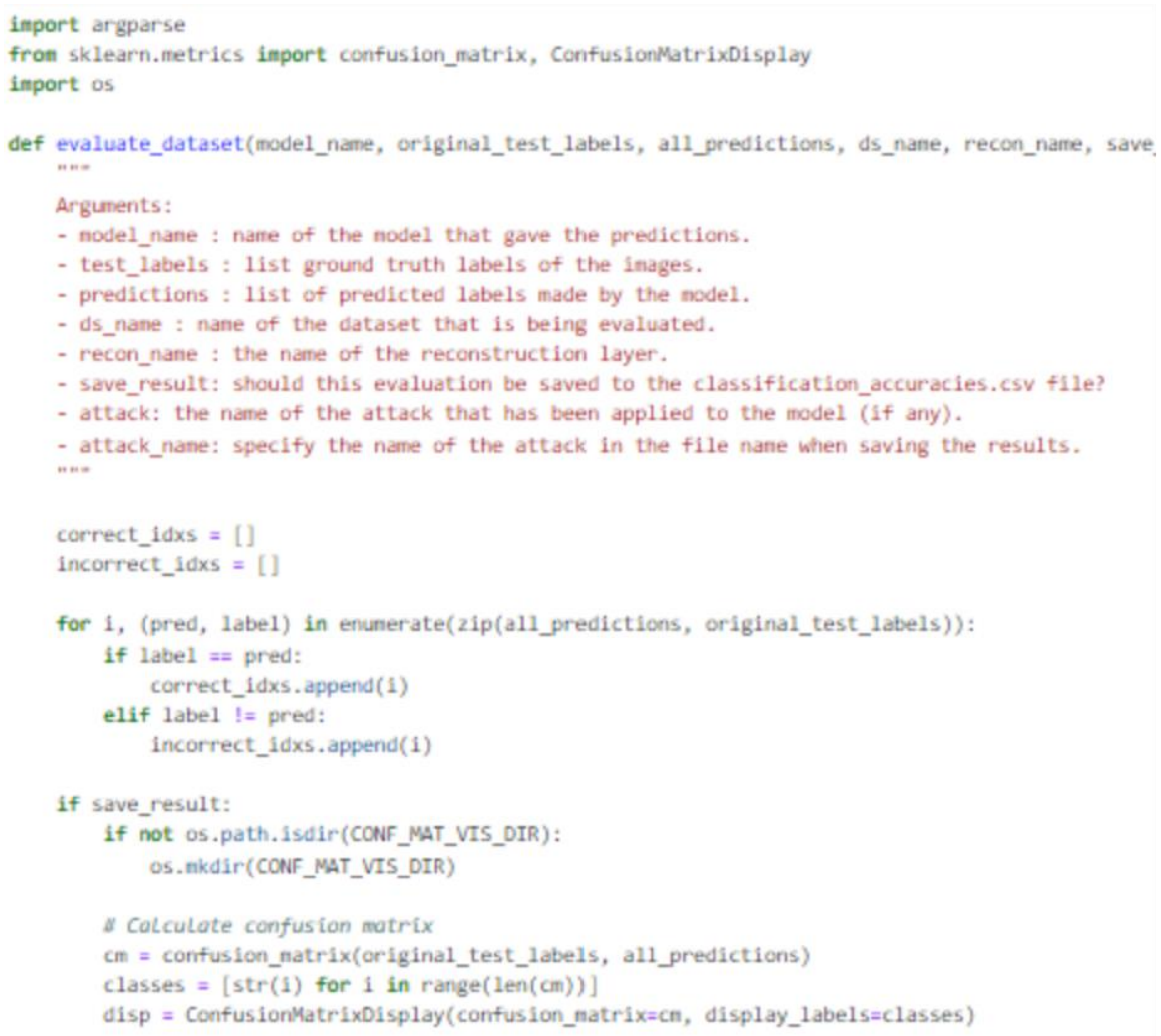
5. **Add other optional arguments, depending what functionalities you chose to implement**

File: answer.py

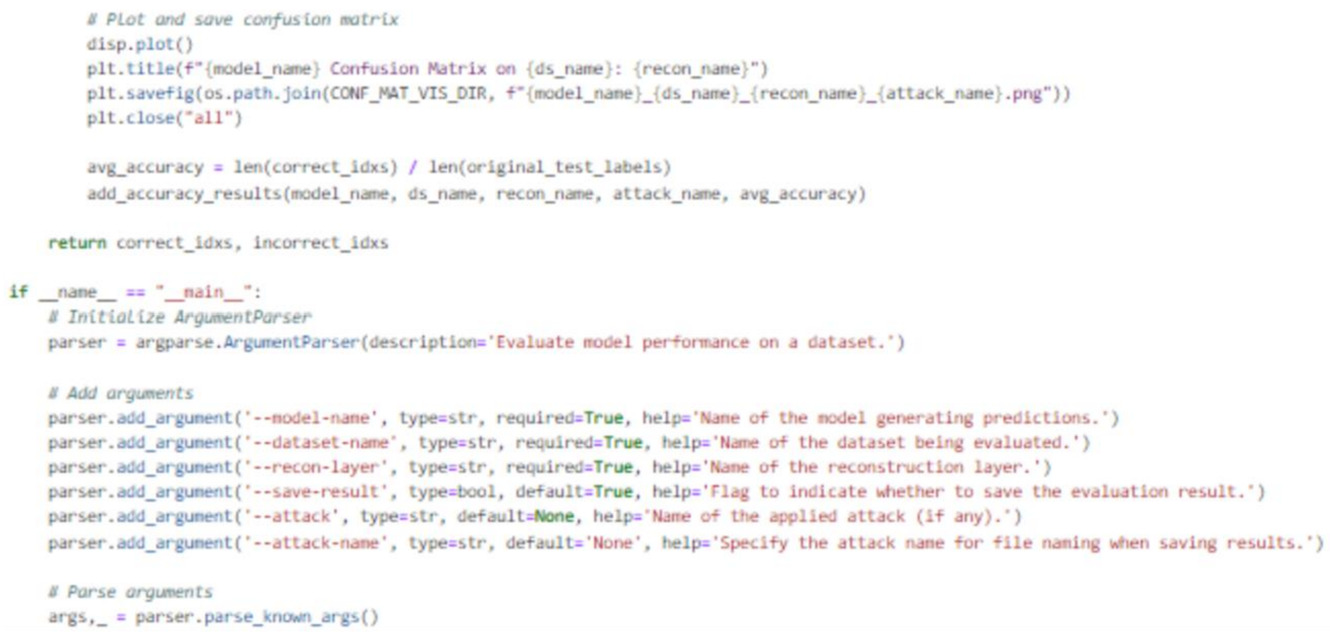
Code snippet:

Inputs:





```
import argparse
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import os

def evaluate_dataset(model_name, original_test_labels, all_predictions, ds_name, recon_name, save
    """
    Arguments:
    - model_name : name of the model that gave the predictions.
    - test_labels : list ground truth labels of the images.
    - predictions : list of predicted labels made by the model.
    - ds_name : name of the dataset that is being evaluated.
    - recon_name : the name of the reconstruction layer.
    - save_result: should this evaluation be saved to the classification_accuracies.csv file?
    - attack: the name of the attack that has been applied to the model (if any).
    - attack_name: specify the name of the attack in the file name when saving the results.
    """

    correct_idxs = []
    incorrect_idxs = []

    for i, (pred, label) in enumerate(zip(all_predictions, original_test_labels)):
        if label == pred:
            correct_idxs.append(i)
        elif label != pred:
            incorrect_idxs.append(i)

    if save_result:
        if not os.path.isdir(CONF_MAT_VIS_DIR):
            os.mkdir(CONF_MAT_VIS_DIR)

        # Calculate confusion matrix
        cm = confusion_matrix(original_test_labels, all_predictions)
        classes = [str(i) for i in range(len(cm))]
        disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=classes)
```



```
        # Plot and save confusion matrix
        disp.plot()
        plt.title(f"{model_name} Confusion Matrix on {ds_name}: {recon_name}")
        plt.savefig(os.path.join(CONF_MAT_VIS_DIR, f"{model_name}_{ds_name}_{recon_name}_{attack_name}.png"))
        plt.close("all")

        avg_accuracy = len(correct_idxs) / len(original_test_labels)
        add_accuracy_results(model_name, ds_name, recon_name, attack_name, avg_accuracy)

    return correct_idxs, incorrect_idxs

if __name__ == "__main__":
    # Initialize ArgumentParser
    parser = argparse.ArgumentParser(description='Evaluate model performance on a dataset.')

    # Add arguments
    parser.add_argument('--model-name', type=str, required=True, help='Name of the model generating predictions.')
    parser.add_argument('--dataset-name', type=str, required=True, help='Name of the dataset being evaluated.')
    parser.add_argument('--recon-layer', type=str, required=True, help='Name of the reconstruction layer.')
    parser.add_argument('--save-result', type=bool, default=True, help='Flag to indicate whether to save the evaluation result.')
    parser.add_argument('--attack', type=str, default=None, help='Name of the applied attack (if any).')
    parser.add_argument('--attack-name', type=str, default='None', help='Specify the attack name for file naming when saving results.')

    # Parse arguments
    args,_ = parser.parse_known_args()
```

- **Function evaluate_dataset:**

Evaluates a model's performance on a dataset.

Takes parameters like model name, ground truth labels, predicted labels, dataset name, reconstruction layer name, etc.

Calculates correct and incorrect indices.

If specified, saves a confusion matrix plot and average accuracy.

- **Command-Line Argument Parsing:**

Uses the argparse module for handling command-line arguments.

Defines required arguments: --model-name, --dataset-name, and --recon-layer.

Defines optional arguments: --save-result, --attack, and --attack-name.

- **Command-Line Arguments:**

--model-name: Name of the model generating predictions (required).

--dataset-name: Name of the dataset being evaluated (required).

--recon-layer: Name of the reconstruction layer (required).

--save-result: Flag to indicate whether to save the evaluation result (optional, default is True).

--attack: Name of the applied attack (optional, default is None).

--attack-name: Specify the attack name for file naming when saving results (optional, default is 'None').

- **Parsing Arguments:**

Parses command-line arguments using argparse.

Captures parsed arguments in the args variable.

- **Using Parsed Arguments:**

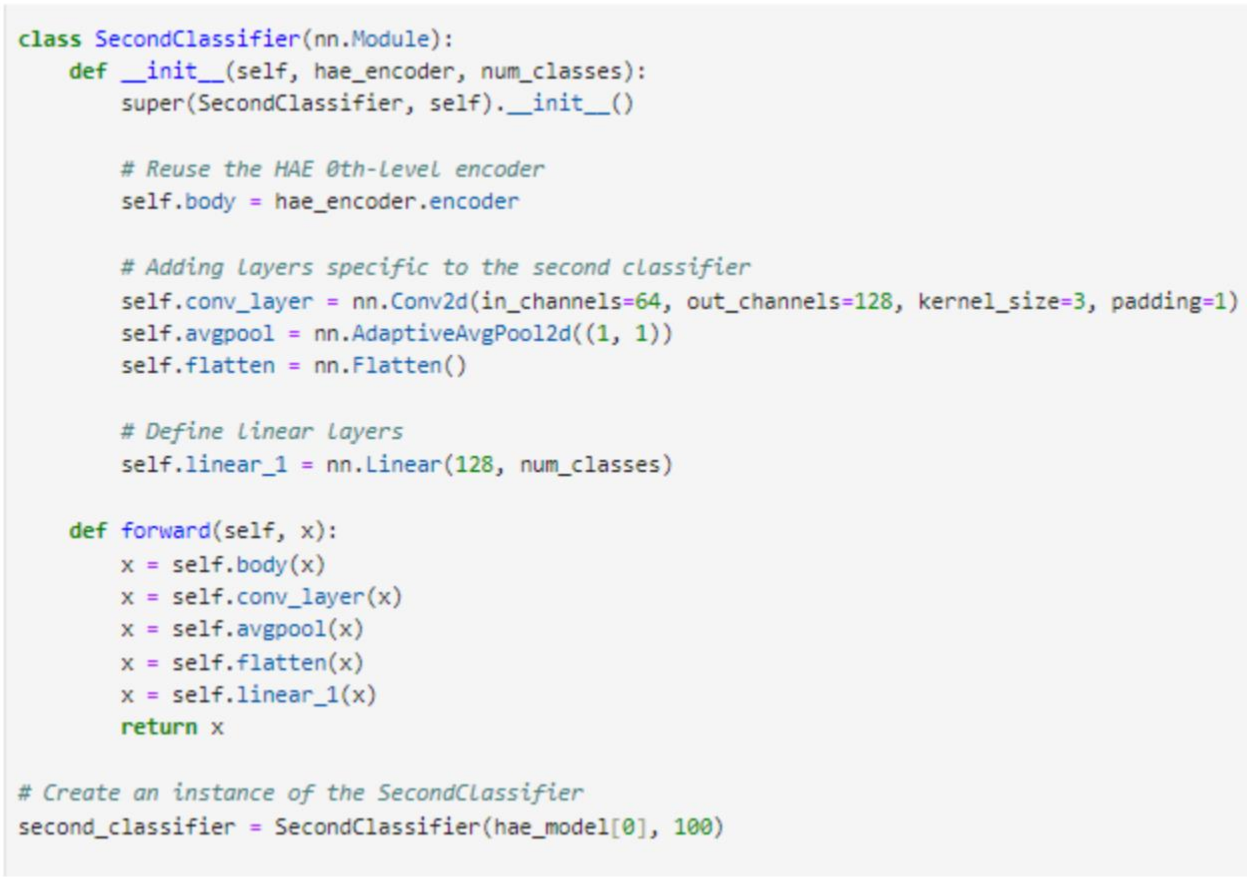
Calls the evaluate_dataset function with the parsed command-line argument values.

3. New Classifier

File: second_classifier.ipynb

1. Create a new classifier:

Code snippet:



```python
class SecondClassifier(nn.Module):
    def __init__(self, hae_encoder, num_classes):
        super(SecondClassifier, self).__init__()

        # Reuse the HAE 0th-level encoder
        self.body = hae_encoder.encoder

        # Adding layers specific to the second classifier
        self.conv_layer = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1)
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.flatten = nn.Flatten()

        # Define linear layers
        self.linear_1 = nn.Linear(128, num_classes)

    def forward(self, x):
        x = self.body(x)
        x = self.conv_layer(x)
        x = self.avgpool(x)
        x = self.flatten(x)
        x = self.linear_1(x)
        return x

# Create an instance of the SecondClassifier
second_classifier = SecondClassifier(hae_model[0], 100)
```

- **Class Definition (SecondClassifier):**

Inherits from nn.Module, the base class for all neural network modules in PyTorch.

Takes an HAE (Hierarchical Autoencoder) encoder (hae_encoder) and the number of output classes (num_classes) as parameters.

- **Constructor (__init__ method):**

Calls the superclass constructor using super to initialize the base class.

Reuses the 0th-level encoder from the HAE model (hae_encoder.encoder) and assigns it to self.body.

Adds specific layers for the second classifier: a 2D convolutional layer (self.conv_layer), adaptive average pooling (self.avgpool), and a linear layer (self.linear_1).

- **Forward Method (forward):**

Defines the forward pass for the network.

Applies the HAE encoder (self.body) to the input x.

Passes the result through the convolutional layer, average pooling, flattening, and linear layer.

- **Instance Creation and Model Summary:**

Creates an instance of SecondClassifier called second_classifier, providing an HAE encoder and the number of output classes.

Prints the summary of the second_classifier using the summary function, specifying the input size as (1, 3, 224, 224) (indicating a single-channel image with size 224x224).

2. **Acccuaries on new classifier:**

Accuracy: 0.008

Accuracy recon0: 0.008

Accuracy recon1: 0.008

Accuracy recon2: 0.008

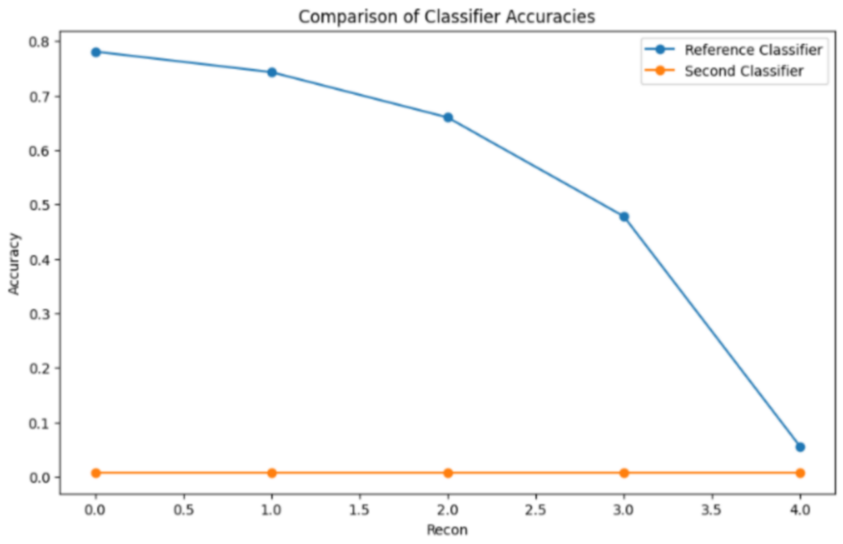
Accuracy recon3: 0.008

Accuracy recon4: 0.008

3. **Comparison of both classifiers:**

File: second_classifier.ipynb

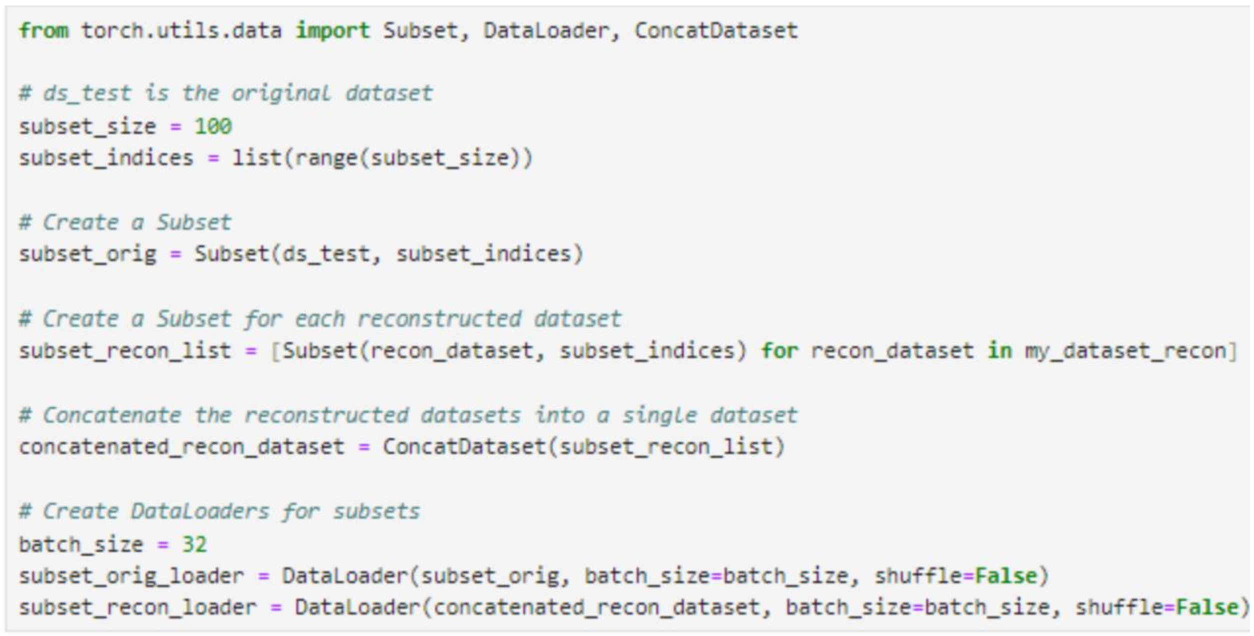
uses Matplotlib to create a line plot comparing accuracies of a reference classifier and a second classifier over reconstruction accuracies, and it saves the plot as an image



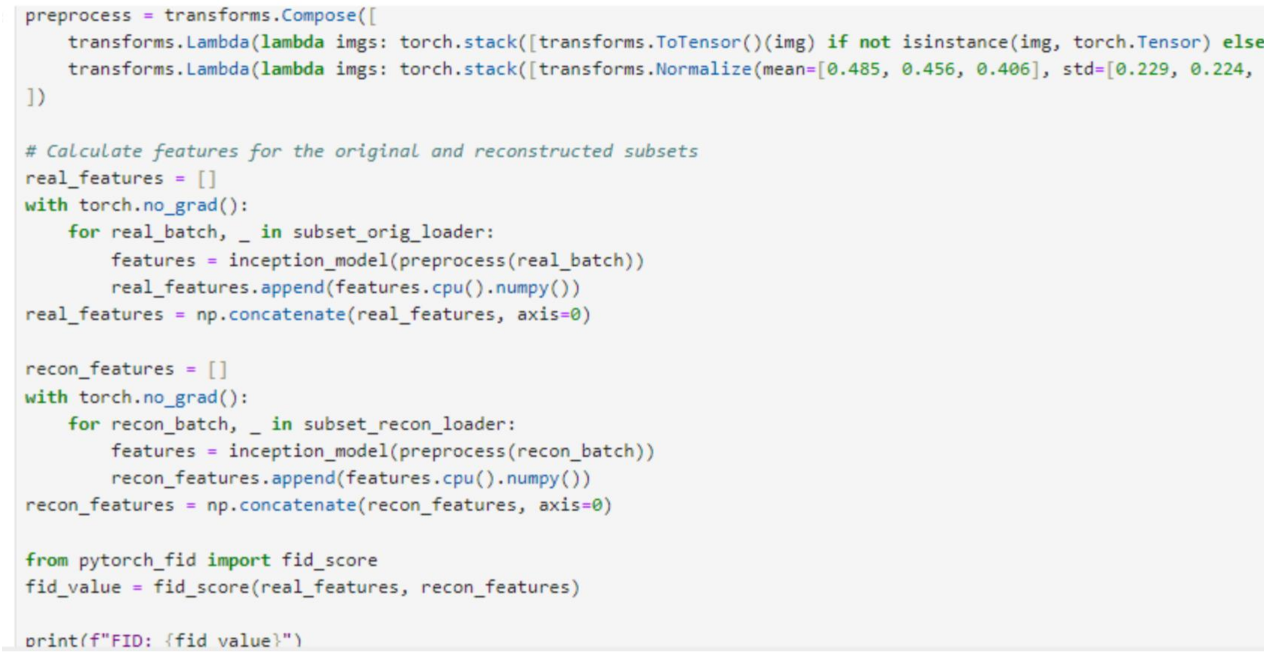
4. Frechet Inception Distance

File: updated_test_hae.ipynb

Code snippet:



```python
from torch.utils.data import Subset, DataLoader, ConcatDataset

# ds_test is the original dataset
subset_size = 100
subset_indices = list(range(subset_size))

# Create a Subset
subset_orig = Subset(ds_test, subset_indices)

# Create a Subset for each reconstructed dataset
subset_recon_list = [Subset(recon_dataset, subset_indices) for recon_dataset in my_dataset_recon]

# Concatenate the reconstructed datasets into a single dataset
concatenated_recon_dataset = ConcatDataset(subset_recon_list)

# Create DataLoaders for subsets
batch_size = 32
subset_orig_loader = DataLoader(subset_orig, batch_size=batch_size, shuffle=False)
subset_recon_loader = DataLoader(concatenated_recon_dataset, batch_size=batch_size, shuffle=False)
```



```python
preprocess = transforms.Compose([
    transforms.Lambda(lambda imgs: torch.stack([transforms.ToTensor()(img) if not isinstance(img, torch.Tensor) else
    transforms.Lambda(lambda imgs: torch.stack([transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224,
])

# Calculate features for the original and reconstructed subsets
real_features = []
with torch.no_grad():
    for real_batch, _ in subset_orig_loader:
        features = inception_model(preprocess(real_batch))
        real_features.append(features.cpu().numpy())
real_features = np.concatenate(real_features, axis=0)

recon_features = []
with torch.no_grad():
    for recon_batch, _ in subset_recon_loader:
        features = inception_model(preprocess(recon_batch))
        recon_features.append(features.cpu().numpy())
recon_features = np.concatenate(recon_features, axis=0)

from pytorch_fid import fid_score
fid_value = fid_score(real_features, recon_features)

print(f"FID: {fid_value}")
```

- **Create Subsets:**

Setting size for the subset.

Generate a list of indices for the subset.

Create a subset of the original dataset.

Create a subset for list of reconstruction data.

- **Create DataLoaders for Subsets:**

Define a batch size for data loading.

Create a DataLoader for the original subset.

Create DataLoaders for each reconstructed subset.

- **Calculate Features using a Pretrained Inception Model:**

Function to extract features using a pretrained Inception model.

- **Calculate fid_score using pytorch_fid**

Load a pretrained Inception model.

Calculate features for the original subset.

Calculate features for the reconstructed subsets.

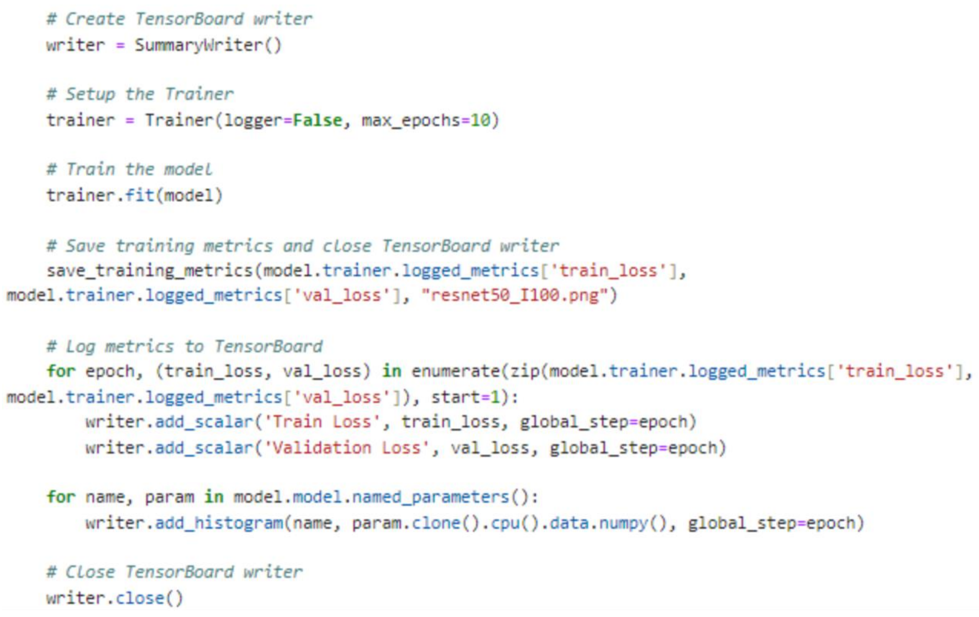
Initialize the fid_score metric from pytorch_fid.

Calculate FID values.

5. Tensorboard to model training

File: train_resnet_lightning.py

Code snippet:



```
    # Create TensorBoard writer
    writer = SummaryWriter()

    # Setup the Trainer
    trainer = Trainer(logger=False, max_epochs=10)

    # Train the model
    trainer.fit(model)

    # Save training metrics and close TensorBoard writer
    save_training_metrics(model.trainer.logged_metrics['train_loss'],
model.trainer.logged_metrics['val_loss'], "resnet50_I100.png")

    # Log metrics to TensorBoard
    for epoch, (train_loss, val_loss) in enumerate(zip(model.trainer.logged_metrics['train_loss'],
model.trainer.logged_metrics['val_loss']), start=1):
        writer.add_scalar('Train Loss', train_loss, global_step=epoch)
        writer.add_scalar('Validation Loss', val_loss, global_step=epoch)

    for name, param in model.model.named_parameters():
        writer.add_histogram(name, param.clone().cpu().data.numpy(), global_step=epoch)

    # Close TensorBoard writer
    writer.close()
```

- **Import Libraries:**

Import necessary libraries, including torch and torch.utils.tensorboard.

- **Create a SummaryWriter:**

Initialize a SummaryWriter to log training information for TensorBoard.

- **Add Scalars During Training:**

Inside training loop, log relevant scalars using writer.add_scalar for metrics like training loss.

- **Add Scalars for Validation:**

Log validation metrics similarly for a validation loop.

- **Add Scalars for Model Parameters:**

Log model parameters (e.g., histograms) to track their changes during training.

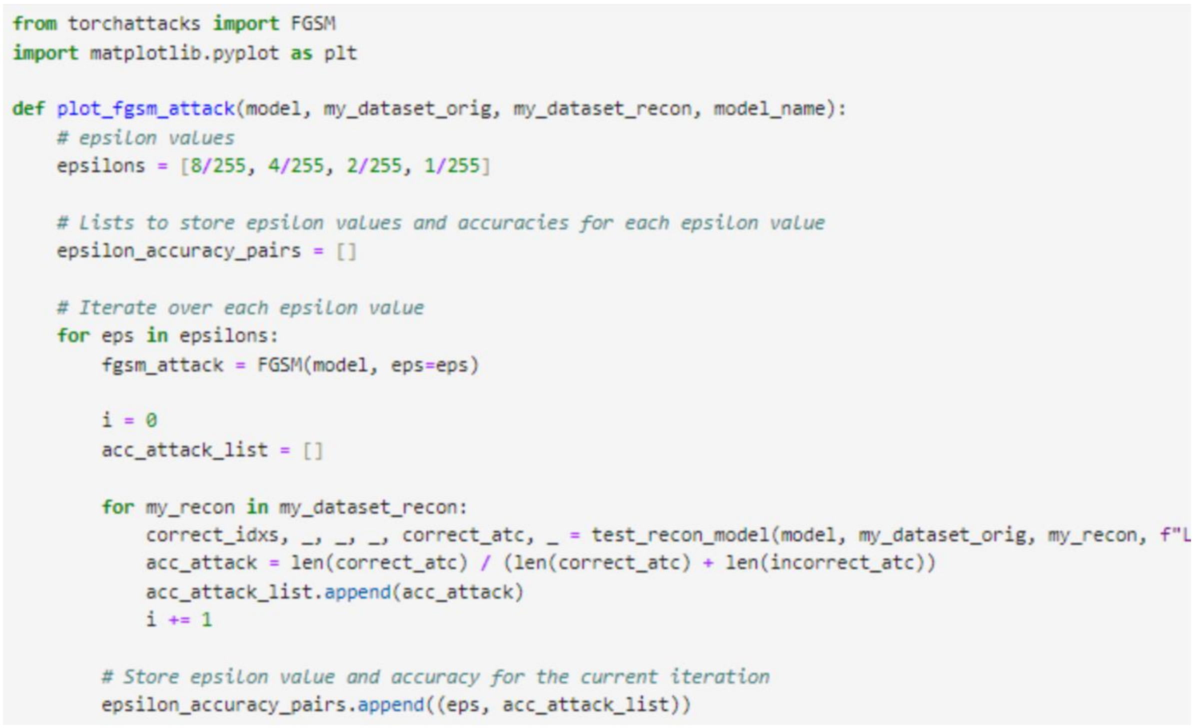
- **Close the SummaryWriter:**

Function to close the SummaryWriter once training is complete.

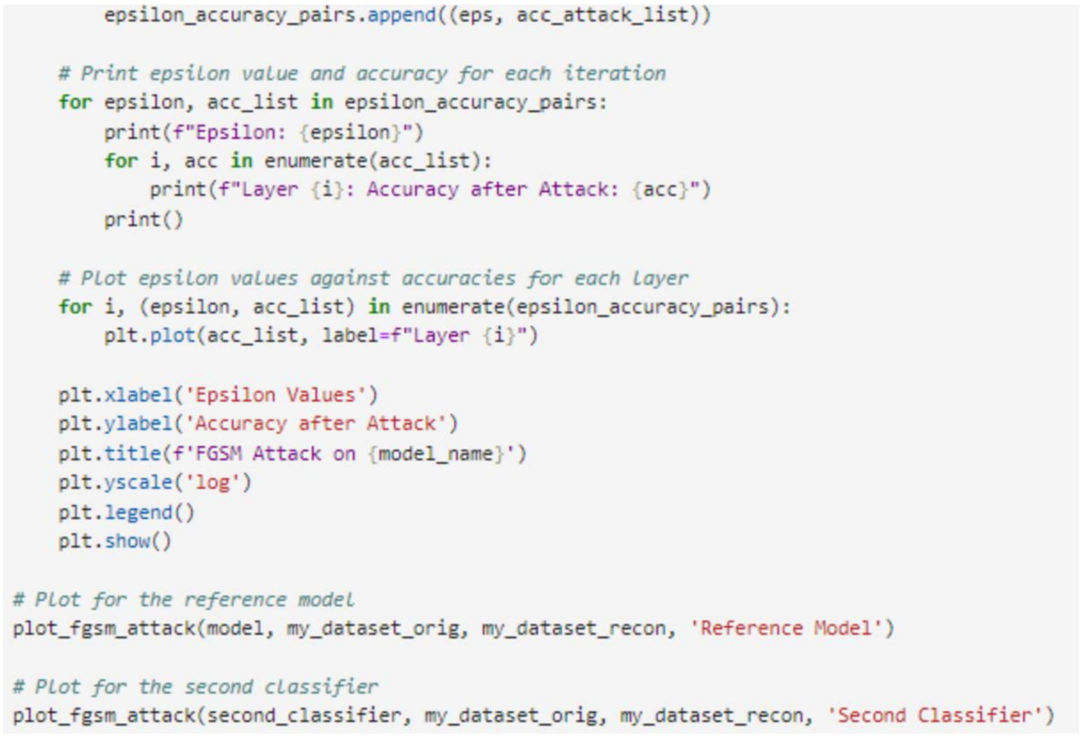
6. FGSM attack

File: second_classifier.ipynb

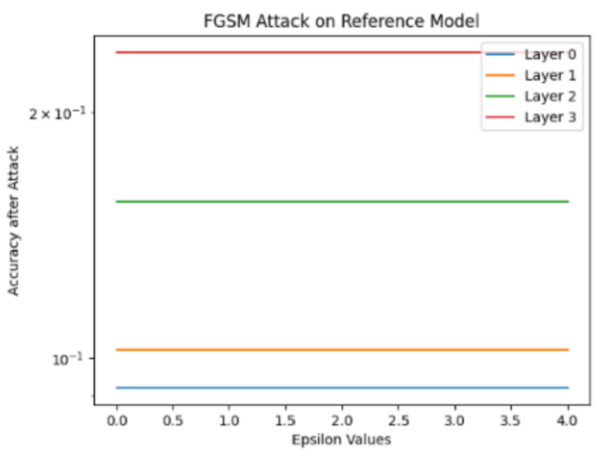
Code snippet:

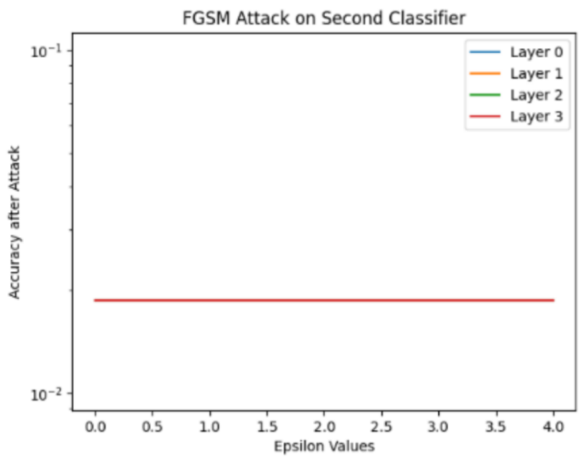


```python
from torchattacks import FGSM
import matplotlib.pyplot as plt

def plot_fgsm_attack(model, my_dataset_orig, my_dataset_recon, model_name):
    # epsilon values
    epsilons = [8/255, 4/255, 2/255, 1/255]

    # Lists to store epsilon values and accuracies for each epsilon value
    epsilon_accuracy_pairs = []

    # Iterate over each epsilon value
    for eps in epsilons:
        fgsm_attack = FGSM(model, eps=eps)

        i = 0
        acc_attack_list = []

        for my_recon in my_dataset_recon:
            correct_idxs, _, _, _, correct_atc, _ = test_recon_model(model, my_dataset_orig, my_recon, f"L
            acc_attack = len(correct_atc) / (len(correct_atc) + len(incorrect_atc))
            acc_attack_list.append(acc_attack)
            i += 1

        # Store epsilon value and accuracy for the current iteration
        epsilon_accuracy_pairs.append((eps, acc_attack_list))
```



```python
        epsilon_accuracy_pairs.append((eps, acc_attack_list))

    # Print epsilon value and accuracy for each iteration
    for epsilon, acc_list in epsilon_accuracy_pairs:
        print(f"Epsilon: {epsilon}")
        for i, acc in enumerate(acc_list):
            print(f"Layer {i}: Accuracy after Attack: {acc}")
        print()

    # Plot epsilon values against accuracies for each layer
    for i, (epsilon, acc_list) in enumerate(epsilon_accuracy_pairs):
        plt.plot(acc_list, label=f"Layer {i}")

    plt.xlabel('Epsilon Values')
    plt.ylabel('Accuracy after Attack')
    plt.title(f'FGSM Attack on {model_name}')
    plt.yscale('log')
    plt.legend()
    plt.show()

# Plot for the reference model
plot_fgsm_attack(model, my_dataset_orig, my_dataset_recon, 'Reference Model')

# Plot for the second classifier
plot_fgsm_attack(second_classifier, my_dataset_orig, my_dataset_recon, 'Second Classifier')
```





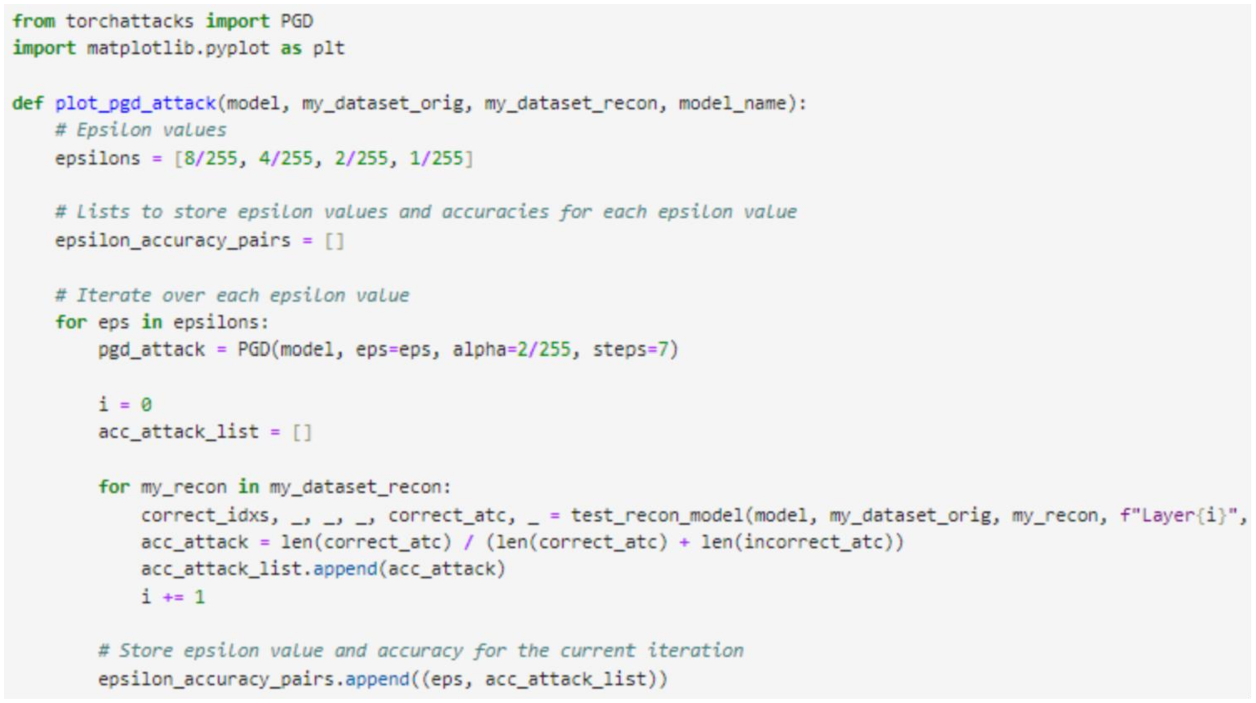
The code aims to assess the resilience of two models (reference model and second classifier) against adversarial attacks, specifically using the Fast Gradient Sign Method (FGSM). By creating a class which does the following.

- **Attack Intensity Exploration:** It explores the impact of varying attack intensities by testing different epsilon values (8/255, 4/255, 2/255, 1/255).
- **FGSM Attack Application:** The Fast Gradient Sign Method (FGSM) attack is applied to both models, altering the input data based on the calculated gradients to induce adversarial perturbations.
- **Accuracy Evaluation:** evaluates the accuracy of the models on reconstructed datasets after each FGSM attack. It calculates the accuracy for each layer of the models.
- **Epsilon-Accuracy Pairs:** stores epsilon values along with corresponding accuracy lists for each model. This information helps in understanding how the models' accuracy changes with different attack intensities.
- **Printed Results:** The accuracy results for each layer and epsilon value are printed for analysis. This includes epsilon values, layer indices, and accuracy after the FGSM attack.
- **Plotting:** generates plots for both models by passing both models through function, illustrating how accuracy varies across different layers and attack intensities.
- **Model Comparison:** By plotting accuracies for both models, it allows for a direct visual comparison of their resilience against adversarial attacks.
- **Insights:** The plots provide insights into how each model responds to different levels of adversarial perturbations. Lower accuracies indicate higher susceptibility to the FGSM attack.
  For second classifier: All layers: Extremely low accuracy after the attack (around 1.86%). Indicates high vulnerability.
  For reference classifier: Low accuracy after the attack (around 9.2%). Shows vulnerability, but relatively higher than the second classifier.

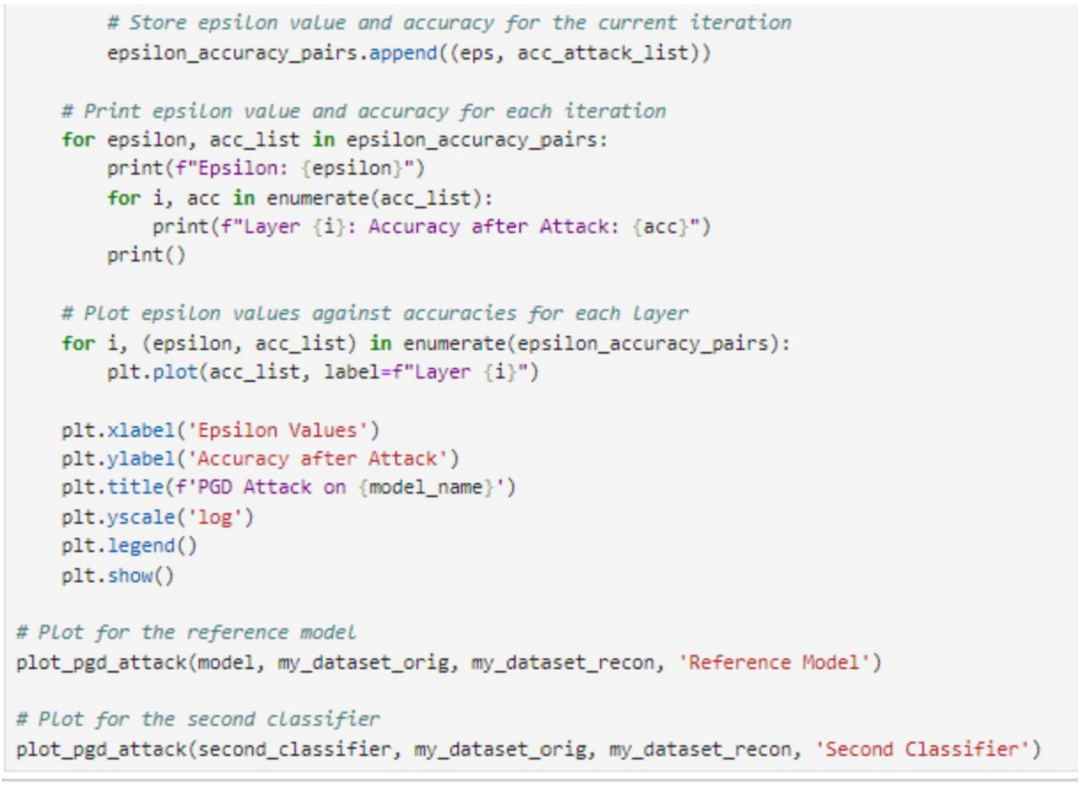
7. PGD attack

File: second_classifier.ipynb

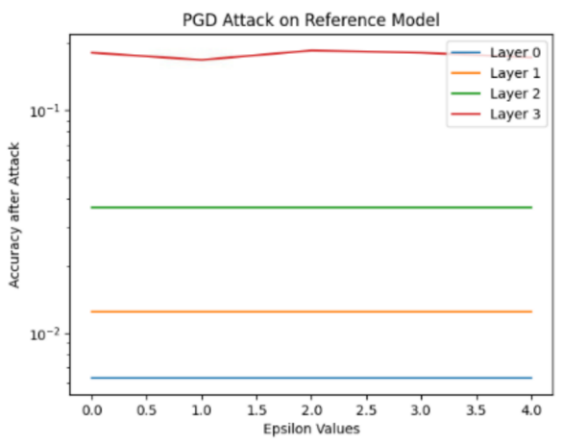
Code snippet:

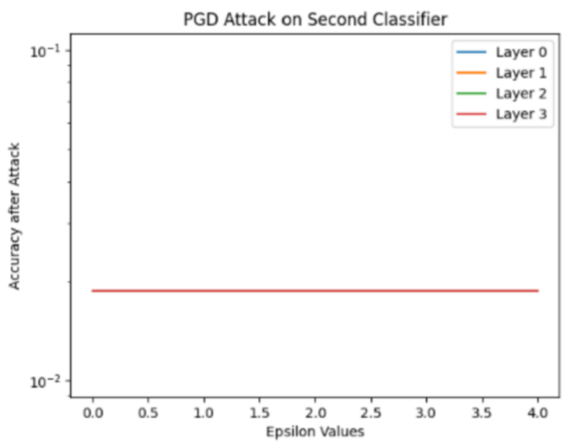


```
from torchattacks import PGD
import matplotlib.pyplot as plt

def plot_pgd_attack(model, my_dataset_orig, my_dataset_recon, model_name):
    # Epsilon values
    epsilons = [8/255, 4/255, 2/255, 1/255]

    # Lists to store epsilon values and accuracies for each epsilon value
    epsilon_accuracy_pairs = []

    # Iterate over each epsilon value
    for eps in epsilons:
        pgd_attack = PGD(model, eps=eps, alpha=2/255, steps=7)

        i = 0
        acc_attack_list = []

        for my_recon in my_dataset_recon:
            correct_idxs, _, _, _, correct_atc, _ = test_recon_model(model, my_dataset_orig, my_recon, f"Layer{i}",
            acc_attack = len(correct_atc) / (len(correct_atc) + len(incorrect_atc))
            acc_attack_list.append(acc_attack)
            i += 1

        # Store epsilon value and accuracy for the current iteration
        epsilon_accuracy_pairs.append((eps, acc_attack_list))
```



```
        # Store epsilon value and accuracy for the current iteration
        epsilon_accuracy_pairs.append((eps, acc_attack_list))

    # Print epsilon value and accuracy for each iteration
    for epsilon, acc_list in epsilon_accuracy_pairs:
        print(f"Epsilon: {epsilon}")
        for i, acc in enumerate(acc_list):
            print(f"Layer {i}: Accuracy after Attack: {acc}")
        print()

    # Plot epsilon values against accuracies for each layer
    for i, (epsilon, acc_list) in enumerate(epsilon_accuracy_pairs):
        plt.plot(acc_list, label=f"Layer {i}")

    plt.xlabel('Epsilon Values')
    plt.ylabel('Accuracy after Attack')
    plt.title(f'PGD Attack on {model_name}')
    plt.yscale('log')
    plt.legend()
    plt.show()

# Plot for the reference model
plot_pgd_attack(model, my_dataset_orig, my_dataset_recon, 'Reference Model')

# Plot for the second classifier
plot_pgd_attack(second_classifier, my_dataset_orig, my_dataset_recon, 'Second Classifier')
```





This is same as the previous one but used a PGD (Projected Gradient Descent) attack as another adversarial attack and plotted for both classifiers.

- **Insights:** Both models exhibit low resilience to the PGD attack, as the accuracy after the attack is generally low across different epsilon values.

The second classifier's resilience is comparable to that of the reference classifier, with both models showing vulnerability to the attack.

8. Pytorch Lightning Module

File: pytorch_resnet_lightning.py, train_resnet_lightning.py

- **PyTorch Lightning Module:**

PyTorch Lightning is a lightweight PyTorch wrapper tha t simplifies and standardizes the training loop.

- **PyTorch Lightning Module Purpose:**

Encapsulates the entire model, optimizer, and training/validation loops in a structured way.

Provides convenience functions for training and validation steps.

Automates common training configurations and practices.

Enhances code readability and maintainability.

- **Key Components in a Lightning Module:**

__init__: Initializes model components and hyperparameters.

forward: Defines the forward pass of the model.

training_step: Defines a training step.

validation_step: Defines a validation step.

training_epoch_end: Optional, performs operations at the end of each training epoch.

validation_epoch_end: Optional, performs operations at the end of each validation epoch.

configure_optimizers: Specifies the optimizer(s) for training.

- **Changes made to PyTorch_ResnetI Class:**

From Base Class to Lightning Module:

The class inherits from pl.LightningModule.

Methods like training_step, validation_step, etc., are now defined to align with Lightning Module requirements.

Changes in the structure are aimed at fitting into the PyTorch Lightning workflow.

- **Changes to train_resnet.py**

Instantiating the Lightning Module: Instead of directly creating an instance of PyTorch_ResnetI, created an instance of pl.Trainer and pass in the Lightning Module (PyTorch_ResnetI).

The training loop is now handled by the Lightning Trainer, reducing the amount of boilerplate code.

- **Configuration and Logging:**

Configuration parameters set using pl.Trainer options, making it easier to manage various training configurations.

Logging, such as tensorboard logging, can be easily integrated using Lightning's built-in features.

- **Training Script Simplification:**

The training script becomes more concise and focuses on the high-level setup of data loaders, model, and Trainer.

- **Benefits:**

Code Organization: Lightning Modules lead to cleaner, more modular code by separating concerns.

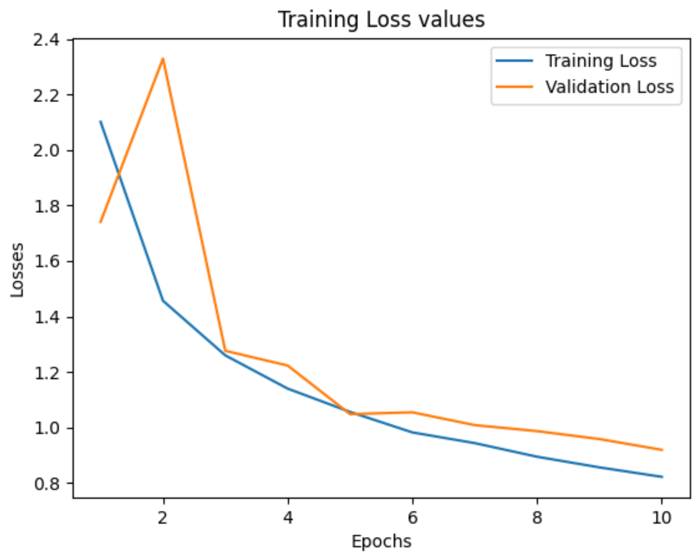
Reproducibility: Lightning provides a standardized training loop, improving reproducibility across experiments.

Ease of Use: Lightning Modules encapsulate common PyTorch patterns.

Flexibility: Lightning doesn't sacrifice flexibility.

In summary, converting the PyTorch_ResnetI class into a PyTorch Lightning Module streamlines the training process, enhances code organization, and provides additional features for logging, configuration, and reproducibility. The training script (train_resnet.py) becomes more concise, focusing on high-level setup and leveraging the capabilities of the Lightning Trainer.

Resnet_I_50.png



9. Different levels of lossy reconstructions

1. **Approach A: Create Reconstructions and Save as New Files**

- **Pros:**

Efficiency: Save reconstructions for reuse, reducing computation time during classifier evaluation.

Consistency: Ensure a consistent dataset for fair and reproducible evaluations across different classifiers.

Reduced Overhead: Avoid recomputing reconstructions for every evaluation, minimizing computational resources.

- **Cons:**

Storage Overhead: Saving reconstructions as files may consume additional storage space.

Outdated Reconstructions: Changes in the HAE model may require regenerating and saving reconstructions.

2. **Approach B: Run the HAE Model Within the Evaluation Script**

- **Pros:**

Dynamic Adaptability: Run the HAE model dynamically during evaluation, adapting to model changes.

No Additional Storage: No extra storage needed for reconstructions, reducing storage requirements.

- **Cons:**

Increased Overhead: Higher computation time as HAE model runs during each evaluation.

Inconsistency: Reconstructions may vary if the HAE model or its parameters change.

Conclusion: Approach A is best for efficiency and consistency, and Approach B for dynamic adaptability, depending on storage constraints and the frequency of model updates.