Lab 6

Transfer Learning & Hyperparameter Tuning

(เนื่องจากกลุ่มของผมทดลอง run Ray tune แล้วพบปัญหาว่าเมื่อ run ได้ถึง 30 กว่าชั่วโมงแล้ว หน้าจอ vscode ที่ ssh ไปที่ apex ได้มีกล่องให้ใส่รหัสเข้า apex ใหม่ เมื่อกรอก พบว่าระบบได้มีการสั่งให้ reload window ทำให้ vscode ของกลุ่มผม reload หน้าใหม่ทำให้การ tune ไม่ run ต่อจากเดิมจะต้อง run ใหม่ทุกครั้ง(เป็นมามากกว่า 3 ครั้งแล้วครับ) จึงได้ไปปรึกษาอาจารย์ในคาบว่าสามารถส่ง code ได้ไหมหาก run ไม่ครบ trials ที่กำหนดไว้ ซึ่งอาจารย์ให้คำตอบว่าได้ ต้องมี trials จำนวนนึง ซึ่งกลุ่มของผมได้ใส่รายละเอียดแนบ code ในช่วงของ grid search และ random search ครับ และมีบางส่วนที่เกิด window reload ในขณะ train เช่นกันทำให้โปรแกรมไม่ run ต่อครับ)

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision.models as models
import torchvision.transforms as transforms
from torchvision.datasets import ImageFolder
from torch.utils.data import DataLoader,Subset,Dataset

import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm
import random
import os
import cv2
from skimage.util import random_noise
from sklearn.model_selection import train_test_split
from skimage.metrics import peak_signal_noise_ratio as psnr
from skimage.metrics import structural_similarity as ssim

seed = 4912
random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)
if torch.cuda.is_available():
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
```

[1] Import Library ที่จำเป็นที่ต้องใช้ใน Lab 5.1 นี้

Data Preparation

```python
class CustomImageDataset(Dataset):
    def __init__(self, image_paths, gauss_noise=False, gauss_blur=False, resize=128, p=0.5):
        self.p = p  # Probability for applying noise/blur
        self.resize = resize  # Final image size
        self.gauss_noise = gauss_noise  # Whether to apply Gaussian noise
        self.gauss_blur = gauss_blur  # Whether to apply Gaussian blur
        self.image_paths = image_paths  # List of image paths

        # Transformation to resize the image to the required size
        self.transform = transforms.Compose([
            transforms.ToPILImage(),
            transforms.Resize((resize, resize)),
            transforms.ToTensor(),
        ])

    def __len__(self):
        return len(self.image_paths)
```

```python
    def __getitem__(self, idx):
        # Load image from the file path
        image_path = self.image_paths[idx]
        image = cv2.imread(image_path)  # Read image using OpenCV (BGR format)
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)  # Convert to RGB

        # Apply transformations to get the ground truth image (resize and convert to tensor)
        gt_image = self.transform(image)

        # Add Gaussian noise to the image with probability p
        if self.gauss_noise and random.random() < self.p:
            noise_mean = random.uniform(-50, 50) / 255.0  # Random noise mean in range [-50, 50] scaled
            image = random_noise(image, mode='gaussian', mean=noise_mean, var=0.01)  # Add Gaussian noise
            image = np.array(255 * image, dtype='uint8')

        # Apply Gaussian blur to the image with probability p
        if self.gauss_blur and random.random() < self.p:
            blur_kernel_size = random.choice(range(3, 12, 2))  # Random kernel size in the range [3, 11]
            image = cv2.GaussianBlur(image, (blur_kernel_size, blur_kernel_size), 2)  # Apply Gaussian blur

        # Resize the noisy image and convert to tensor
        noisy_image = self.transform(image)

        return noisy_image, gt_image
```

[2] สร้าง class: CustomImageDataset เพื่อช่วยในการสร้าง dataset โดยเพิ่ม Noise และ Blur ในรูปภาพ เพื่อใช้ในการฝึกโมเดล ให้มีความทนต่อสภาพแวดล้อมจริง และสามารถทำงานได้อย่างมีประสิทธิภาพในสถานการณ์ต่างๆ

- **__init__(self, image_paths, gauss_noise=False, gauss_blur=False, resize=128, p=0.5):**
    - รับค่า image_paths ซึ่งเป็นเส้นทางไปยังภาพ
    - รับค่า gauss_noise ซึ่งเป็นค่า Boolean ระบุว่าจะเพิ่มเสียงรบกวนหรือไม่
    - รับค่า gauss_blur ซึ่งเป็นค่า Boolean ระบุว่าจะเบลอภาพหรือไม่
    - รับค่า resize ซึ่งเป็นขนาดของภาพที่ต้องการ
    - รับค่า p ซึ่งเป็นความน่าจะเป็นที่จะเพิ่มเสียงรบกวนหรือเบลอภาพ
    - การทำงาน:
        - สร้าง transform ซึ่งเป็นชุดการแปลงภาพเพื่อปรับขนาดและแปลงเป็น tensor

- **__len__:** คืนค่าจำนวนภาพทั้งหมดใน path: image_paths
- **__getitem__:** คืนค่าภาพลำดับที่ idx พร้อมทั้งเพิ่มสัญญาณรบกวนหรือการเบลอตามความน่าจะเป็นที่กำหนด
    - โหลดภาพ: อ่านภาพจากเส้นทางไฟล์ที่กำหนดโดย image_paths[idx] โดยแปลงภาพจาก BGR เป็น RGB
    - สร้างภาพต้นแบบ (ground truth): โดยนำภาพไปแปลงขนาดจาก transform ก่อน
    - เพิ่มสัญญาณรบกวน: หาก gauss_noise เป็น True และมีการสุ่มตัวเลขน้อยกว่า p จะเพิ่มสัญญาณรบกวนแบบ Gaussian โดยค่าเฉลี่ยของสัญญาณรบกวนแบบ Gaussian จะสุ่มระหว่าง -50.0/255.0 ถึง 50.0/255.0 เพื่อใช้ในการเพิ่มสัญญาณรบกวนเข้าไปในภาพ
    - เพิ่มการเบลอ: หาก gauss_blur เป็น True และมีการสุ่มตัวเลขน้อยกว่า p จะเพิ่มการเบลอแบบ Gaussian โดยขนาดของเคอร์เนลที่ใช้ในการเบลอภาพแบบ Gaussian จะสุ่มเลือกตัวเลขคี่ระหว่าง 3 ถึง 11
    - แปลงภาพที่มีสัญญาณรบกวนหรือการเบลอ: ปรับขนาดภาพและแปลงเป็นเทนเซอร์ PyTorch
    - โดยจะ return:
        - noisy_image: ภาพที่มีสัญญาณรบกวนหรือการเบลอ
        - gt_image: ภาพต้นแบบ (ground truth) ที่ไม่มีสัญญาณรบกวนหรือการเบลอ

```python
### START CODE HERE ###
def imshow_grid(images, ncols=4):
    fig, axes = plt.subplots(nrows=len(images) // ncols, ncols=ncols, figsize=(12, 12))
    axes = axes.flatten()

    for img, ax in zip(images, axes):
        ax.imshow(np.transpose(img.numpy(), (1, 2, 0)))  # Convert the image to the right format
        ax.axis('off')

    plt.tight_layout()
    plt.show()
### END CODE HERE ###
```

[3] ฟังก์ชันนี้ใช้สำหรับการแสดงผลภาพหลายภาพเป็นตาราง grid (This function is used to display multiple images in a neat grid layout, with the number of columns specified by ncols. The images are converted from PyTorch tensors to a format that can be displayed using Matplotlib.)

```
### START CODE HERE ###
relative_dir = "img_align_celeba"
data_dir = os.path.abspath(relative_dir)
# print(data_dir)
image_paths = [os.path.join(data_dir, img) for img in os.listdir(data_dir) if img.endswith(".jpg")]

dataset = CustomImageDataset(image_paths, gauss_noise=True, gauss_blur=5, resize=128, p=0.5)
dataloader = DataLoader(dataset, batch_size=8, shuffle=True)
### END CODE HERE ###
```

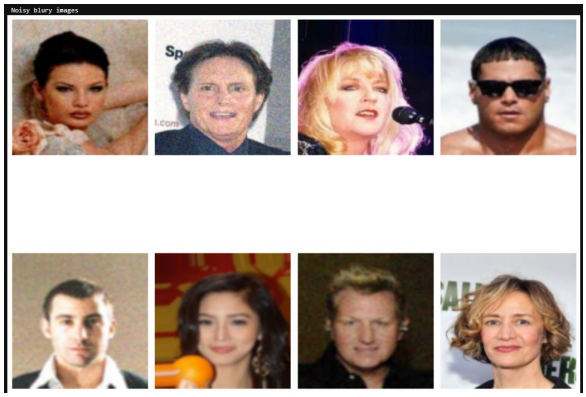
[4] This code block sets up the dataset and data loader for processing a set of images stored in a directory. It applies Gaussian noise and blur to the images, resizes them, and prepares them for use in a model, loading them in batches with shuffling enabled. โดยที่

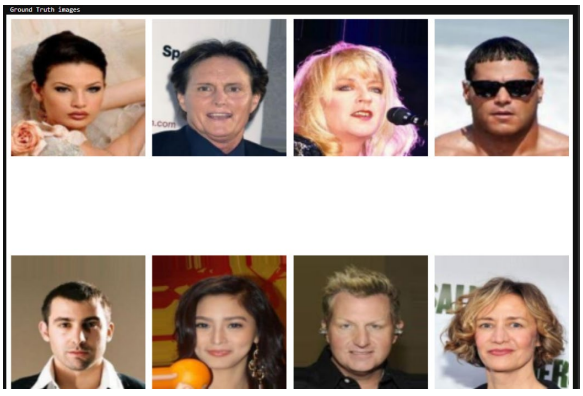
การสร้างชุดข้อมูลและ DataLoader

- dataset = CustomImageDataset(image_paths, gauss_noise=True, gauss_blur=5, resize=128, p=0.5): สร้าง object จาก CustomImageDataset โดยใช้ image_paths เป็นแหล่งข้อมูลและกำหนดพารามิเตอร์ต่างๆ:
  - gauss_noise=True: เพิ่มสัญญาณรบกวนแบบ Gaussian
  - gauss_blur=5: เพิ่มการเบลอแบบ Gaussian โดยใช้ขนาดเคอร์เนล 5
  - resize=128: ปรับขนาดภาพให้เป็น 128x128
  - p=0.5: ความน่าจะเป็นที่จะใช้สัญญาณรบกวนหรือการเบลอคือ 0.5
- dataloader = DataLoader(dataset, batch_size=8, shuffle=True): สร้างวัตถุ DataLoader เพื่อใช้ในการโหลดและแบ่งชุดข้อมูลออกเป็นแบตช์ (batch) สำหรับการฝึกโมเดล
  - batch_size=8: ขนาดของแต่ละแบตช์คือ 8 ภาพ
  - shuffle=True: สับเปลี่ยนลำดับภาพในแต่ละแบตช์แบบสุ่ม

```
### START CODE HERE ###
batch, gt_img = next(iter(dataloader))

# Display a batch of noisy images and ground truth images
print("Noisy blury images")
imshow_grid(batch)
print("Ground Truth images")
imshow_grid(gt_img)
### END CODE HERE ###
```





Noisy blury images Ground Truth images

[5] batch, gt_img = next(iter(dataloader)): โหลดแบตช์แรกจาก dataloader โดย batch จะเป็นรายการของภาพที่มีสัญญาณรบกวนและการเบลอ และ gt_img จะเป็นรายการของภาพต้นแบบ (ground truth) แสดง batch, gt_image ผแบบ grid (This code block retrieves a batch of noisy or blurry images and their corresponding ground truth images from the DataLoader, then displays them in grid format for visual inspection. The messages printed to the console clarify which images are being displayed.)

Create Autoencoder model

```
### START CODE HERE ###
class DownSamplingBlock(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size=3, stride=1, padding=1):
        super(DownSamplingBlock, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size=kernel_size, stride=stride, padding=padding)
        self.bn = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU()
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

    def forward(self, x):
        x = self.conv(x)
        x = self.bn(x)
        x = self.relu(x)
        x = self.pool(x)
        return x
```

[6.1] DownSamplingBlock: บล็อกสำหรับลดขนาดของ feature map ในชั้นของ Encoder

- The DownSamplingBlock reduces the spatial dimensions of the input feature map using a combination of convolution, batch normalization, ReLU activation, and max-pooling.

```python
class UpSamplingBlock(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size=3, stride=1, padding=1):
        super(UpSamplingBlock, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size=kernel_size, stride=stride, padding=padding)
        self.bn = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU()
        self.upsample = nn.Upsample(scale_factor=2, mode='bilinear', align_corners=True)

    def forward(self, x):
        x = self.conv(x)
        x = self.bn(x)
        x = self.relu(x)
        x = self.upsample(x)
        return x
```

[6.2] UpSamplingBlock: บล็อกสำหรับเพิ่มขนาดของ feature map ในชั้นของ Decoder

- The UpSamplingBlock increases the spatial dimensions of the input feature map using convolution, batch normalization, ReLU activation, and bilinear upsampling.

```python
class Autoencoder(nn.Module):
    def __init__(self, channels=[64, 128, 256], input_channels=3, output_channels=3):
        super(Autoencoder, self).__init__()

        # Encoder: Downsampling layers
        self.encoder = nn.ModuleList()
        in_channels = input_channels
        for out_channels in channels:
            self.encoder.append(DownSamplingBlock(in_channels, out_channels))
            in_channels = out_channels

        # Decoder: Upsampling layers (reverse of the encoder)
        self.decoder = nn.ModuleList()
        for out_channels in reversed(channels[:-1]):
            self.decoder.append(UpSamplingBlock(in_channels, out_channels))
            in_channels = out_channels
        # Final layer to map back to the original number of channels
        self.decoder.append(UpSamplingBlock(in_channels, output_channels))

    def forward(self, x):
        # Encoding
        for layer in self.encoder:
            x = layer(x)

        # Decoding
        for layer in self.decoder:
            x = layer(x)

        return x

### END CODE HERE ###
```

[6.3] The Autoencoder class defines a simple autoencoder model with customizable encoding and decoding layers.

- **Encoder**: Consists of downsampling layers that reduce the spatial dimensions while increasing the depth (number of channels) of the feature map.
- **Decoder**: Consists of upsampling layers that increase the spatial dimensions while reducing the depth, aiming to reconstruct the original image.
- The forward pass involves sequentially passing the input through the encoder to compress the data and then through the decoder to reconstruct the output.

Train Autoencoder

```python
### START CODE HERE ###
def train(model, opt, loss_fn, train_loader, test_loader, epochs=10, checkpoint_path=None, device='cpu'):
    print("🚊Training on", device)
    model = model.to(device)

    for epoch in range(epochs):
        # Training phase
        model.train()
        total_train_loss = 0
        train_bar = tqdm(train_loader, desc=f'🚀Training Epoch [{epoch+1}/{epochs}]', unit='batch')
        for images, gt in train_bar:
            images, gt = images.to(device), gt.to(device)

            # Forward pass
            opt.zero_grad()
            outputs = model(images)
            loss = loss_fn(outputs, gt)
            loss.backward()
            opt.step()

            total_train_loss += loss.item()
            train_bar.set_postfix(loss=loss.item())

        avg_train_loss = total_train_loss / len(train_loader)
```

```python
# Testing phase
model.eval()
total_test_loss = 0
total_psnr = 0
total_ssim = 0

test_bar = tqdm(test_loader, desc='📄Testing', unit='batch')
with torch.no_grad():
    for images, gt in test_bar:
        images, gt = images.to(device), gt.to(device)

        outputs = model(images)
        loss = loss_fn(outputs, gt)
        total_test_loss += loss.item()

        # Convert outputs and gt to numpy arrays for PSNR and SSIM calculations
        outputs_np = outputs.permute(0, 2, 3, 1).cpu().numpy()
        gt_np = gt.permute(0, 2, 3, 1).cpu().numpy()

        # Calculate PSNR and SSIM for the batch
        batch_psnr = np.mean([psnr(gt_np[i], outputs_np[i]) for i in range(gt_np.shape[0])])

        # Set win_size to 3 and use channel_axis for SSIM, specifying data_range
        batch_ssim = np.mean([ssim(gt_np[i], outputs_np[i], win_size=3, channel_axis=-1, data_range=1.0) for i in range(gt_np.shape[0])])

        total_psnr += batch_psnr
        total_ssim += batch_ssim

        test_bar.set_postfix(loss=loss.item(), psnr=batch_psnr, ssim=batch_ssim)

avg_test_loss = total_test_loss / len(test_loader)
avg_psnr = total_psnr / len(test_loader)
avg_ssim = total_ssim / len(test_loader)
```

```python
        # Log summary for the epoch
        print(f'Summary for Epoch {epoch+1}/{epochs}:')
        print(f'    Train    avg_loss: {avg_train_loss}')
        print(f'    Test     avg_loss: {avg_test_loss}')
        print(f'                PSNR : {avg_psnr}')
        print(f'                SSIM : {avg_ssim}')

        # Save model at the last epoch
        if epoch == epochs - 1 and checkpoint_path:
            torch.save(model.state_dict(), checkpoint_path)
            print(f'Model saved at {checkpoint_path}')

### END CODE HERE ###
```

[7] ฟังก์ชัน train สำหรับการฝึกโมเดล Autoencoder: ทำหน้าที่ในการฝึกโมเดล Autoencoder โดยใช้ข้อมูลชุดฝึก (train_loader) และข้อมูลชุดทดสอบ (test_loader) โดยใช้ PSNR และ SSIM เป็นตัววัด image quality

ขั้นตอนการทำงาน:

- เฟสการฝึก (Training phase):
  - ตั้งค่าโมเดลให้เป็นโหมดฝึก (model.train())
  - total_train_loss: ตัวแปรสะสมผลรวมของค่าสูญเสียในการฝึก
  - วนลูปผ่านข้อมูลชุดฝึก (train_loader) โดยใช้ tqdm แสดงแถบความคืบหน้า
    - ย้ายข้อมูลภาพ (images) และข้อมูลภาพต้นแบบ (gt) ไปยังอุปกรณ์
    - Forward pass:
      - รีเซ็ตตัวสะสมการไล่ระดับ (gradient) ของตัวเพิ่มประสิทธิภาพ (opt.zero_grad())
      - ป้อนข้อมูลภาพ (images) เข้าสู่โมเดล (outputs = model(images))
      - คำนวณค่าสูญเสีย (loss) โดยใช้ฟังก์ชัน loss_fn

- ย้อนกลับ (backpropagation) เพื่อคำนวณการไล่ระดับสำหรับพารามิเตอร์ของโมเดล (loss.backward())
- ปรับปรุงพารามิเตอร์ของโมเดล (opt.step())

- เพิ่มค่าสูญเสียของแบตช์ปัจจุบันลงใน total_train_loss
- อัพเดทข้อความแถบความคืบหน้าด้วยค่าสูญเสียของแบตช์ปัจจุบัน
- คำนวณค่าสูญเสียเฉลี่ยในการฝึก (avg_train_loss)

- เฟสการทดสอบ (Testing phase):
  - ตั้งค่าโมเดลให้เป็นโหมดประเมินผล (model.eval())
  - total_test_loss: ตัวแปรสะสมผลรวมของค่าสูญเสียในการทดสอบ
  - total_psnr: ตัวแปรสะสมผลรวมของค่า PSNR (Peak Signal-to-Noise Ratio)
  - total_ssim: ตัวแปรสะสมผลรวมของค่า SSIM (Structural Similarity Index Measure)
  - วนลูปผ่านข้อมูลชุดทดสอบ (test_loader) โดยใช้ tqdm แสดงแถบความคืบหน้า
    - ย้ายข้อมูลภาพ (images) และข้อมูลภาพต้นแบบ (gt) ไปยังอุปกรณ์
    - ปิดการคำนวณการไล่ระดับ (gradient) เพื่อประหยัดทรัพยากร (with torch.no_grad())
      - ป้อนข้อมูลภาพ (images) เข้าสู่โมเดล (outputs = model(images))
      - คำนวณค่าสูญเสีย (loss) โดยใช้ฟังก์ชัน loss_fn
      - เพิ่มค่าสูญเสียของแบตช์ปัจจุบันลงใน total_test_loss
      - แปลงข้อมูลผลลัพธ์ (outputs) และข้อมูลภาพต้นแบบ (gt) เป็น NumPy array เพื่อคำนวณ PSNR และ SSIM
      - วนลูปคำนวณ PSNR และ SSIM สำหรับแต่ละภาพในแบตช์
        - เพิ่มค่า PSNR และ SSIM ของแบตช์ปัจจุบัน

```
### START CODE HERE ###
# CODE FOR PRODUCTION USE
# ----------------------------------------------------------------------------------------------------
## START CODE HERE ###
files = os.listdir(data_dir)
files = [os.path.join(data_dir, file) for file in files]

# Split the dataset into training and testing sets
train_files, test_files = train_test_split(files, test_size=0.3, shuffle=True, random_state=2024)

train_dataset = CustomImageDataset(train_files, gauss_noise=True, gauss_blur=True)
test_dataset = CustomImageDataset(test_files, gauss_noise=True, gauss_blur=True)
trainloader = DataLoader(train_dataset, batch_size=16, shuffle=True)
testloader = DataLoader(test_dataset, batch_size=16, shuffle=False)
# ----------------------------------------------------------------------------------------------------
### END CODE HERE ###
```

[8] เตรียมชุดข้อมูลสำหรับการฝึกและทดสอบโมเดล Autoencoder

- การโหลดและแบ่งชุดข้อมูล
    - files = os.listdir(data_dir): โหลดรายการไฟล์ทั้งหมดจากไดเรกทอรี data_dir
    - files = [os.path.join(data_dir, file) for file in files]: สร้างรายการของเส้นทางไฟล์ภาพทั้งหมดในไดเรกทอรี
    - train_files, test_files = train_test_split(files, test_size=0.3, shuffle=True, random_state=2024): แบ่งชุดข้อมูลออกเป็นชุดฝึก (training set) และชุดทดสอบ (testing set) โดยใช้ฟังก์ชัน train_test_split
    - test_size=0.3: กำหนดขนาดของชุดทดสอบเป็น 30% ของข้อมูลทั้งหมด
    - shuffle=True: สับเปลี่ยนลำดับไฟล์ก่อนแบ่ง
    - random_state=2024: กำหนดค่าสุ่มเพื่อให้การแบ่งชุดข้อมูลสามารถทำซ้ำได้
- การสร้างชุดข้อมูลและ DataLoader
    - train_dataset = CustomImageDataset(train_files, gauss_noise=True, gauss_blur=True): สร้างชุดข้อมูลฝึกโดยใช้รายการ train_files และเพิ่มสัญญาณรบกวนและการเบลอ
    - test_dataset = CustomImageDataset(test_files, gauss_noise=True, gauss_blur=True): สร้างชุดข้อมูลทดสอบโดยใช้รายการ test_files และเพิ่มสัญญาณรบกวนและการเบลอ
    - trainloader = DataLoader(train_dataset, batch_size=16, shuffle=True): สร้าง DataLoader สำหรับชุดข้อมูลฝึก โดยกำหนดขนาดแบตช์เป็น 16 และสับเปลี่ยนลำดับแบตช์
    - testloader = DataLoader(test_dataset, batch_size=16, shuffle=False): สร้าง DataLoader สำหรับชุดข้อมูลทดสอบ โดยกำหนดขนาดแบตช์เป็น 16 และไม่สับเปลี่ยนลำดับแบตช์

```
### START CODE HERE ###
# # Initialize model, optimizer, and loss function
model = Autoencoder()
opt = optim.Adam(model.parameters(), lr=0.001)
loss_fn = nn.MSELoss()

# Train the model
train(model, opt, loss_fn, trainloader, testloader, epochs=1, checkpoint_path='autoencoder.pth', device='cuda' if torch.cuda.is_available() else 'cpu')

### END CODE HERE ###
```

[9] The code initializes the Autoencoder model along with the Adam optimizer and MSE loss function.

- The model is trained for 10 epoch on the CPU (or GPU if available), with training and testing metrics logged to the console.
- After training, the model is saved to the specified checkpoint file (autoencoder.pth), which can be loaded later for inference or further training.

โดยจะ train ทั้งหมด 10 รอบ:

```
Training on cuda

Training Epoch [1/10]: 100%|| 1313/1313 [04:13<00:00, 5.18batch/s, loss=0.0288]
Testing: 100%|| 563/563 [02:49<00:00,  3.33batch/s, loss=0.0102, psnr=20.4, ssim=0.592]

Summary for Epoch 1/10:
    Train    avg_loss: 0.016828232376729136
    Test     avg_loss: 0.011866646347365626
                PSNR : 19.824324182952207
                SSIM : 0.5469598770141602

Training Epoch [2/10]: 100%|| 1313/1313 [02:01<00:00, 10.81batch/s, loss=0.0354]
Testing: 100%|| 563/563 [01:03<00:00,  8.81batch/s, loss=0.0672, psnr=12, ssim=0.374]

Summary for Epoch 2/10:
    Train    avg_loss: 0.010980363824225322
    Test     avg_loss: 0.07131565424413494
                PSNR : 12.214592377605568
                SSIM : 0.3296174108982086
```

```
Training Epoch [3/10]: 100%|| 1313/1313 [01:41<00:00, 12.88batch/s, loss=0.00645]
Testing: 100%|| 563/563 [01:03<00:00,  8.87batch/s, loss=0.00687, psnr=21.9, ssim=0.655]

Summary for Epoch 3/10:
    Train    avg_loss: 0.011177784523175516
    Test     avg_loss: 0.008834293644815736
                PSNR : 21.2538661172138
                SSIM : 0.6332727670669556

Training Epoch [4/10]: 100%|| 1313/1313 [01:40<00:00, 13.03batch/s, loss=0.00886]
Testing: 100%|| 563/563 [01:03<00:00,  8.87batch/s, loss=0.0053, psnr=23.2, ssim=0.702]

Summary for Epoch 4/10:
    Train    avg_loss: 0.009010084485174 11
    Test     avg_loss: 0.007879532331048277
                PSNR : 21.75618046283166
                SSIM : 0.6365190148353577
```

```
Training Epoch [5/10]: 100%|| 1313/1313 [01:41<00:00, 12.91batch/s, loss=0.0142]
Testing: 100%|| 563/563 [01:04<00:00,  8.74batch/s, loss=0.00456, psnr=23.8, ssim=0.691]

Summary for Epoch 5/10:
    Train    avg_loss: 0.008178445336076248
    Test     avg_loss: 0.007263613175799273
                PSNR : 22.055557457817027
                SSIM : 0.6334452033042908

Training Epoch [6/10]: 100%|| 1313/1313 [01:42<00:00, 12.77batch/s, loss=0.00925]
Testing: 100%|| 563/563 [01:04<00:00,  8.75batch/s, loss=0.00581, psnr=22.8, ssim=0.696]

Summary for Epoch 6/10:
    Train    avg_loss: 0.007726023026145512
    Test     avg_loss: 0.006889203662585407 5
                PSNR : 22.076778052273998
                SSIM : 0.646493136882782
```

```
Training Epoch [7/10]: 100%|| 1313/1313 [01:42<00:00, 12.82batch/s, loss=0.00723]
Testing: 100%|| 563/563 [01:03<00:00,  8.84batch/s, loss=0.00624, psnr=22.8, ssim=0.724]

Summary for Epoch 7/10:
    Train    avg_loss: 0.00721455800438531
    Test     avg_loss: 0.006522119533250747
                PSNR : 22.35701727128971
                SSIM : 0.6659809947013855

Training Epoch [8/10]: 100%|| 1313/1313 [01:42<00:00, 12.79batch/s, loss=0.0129]
Testing: 100%|| 563/563 [01:04<00:00,  8.73batch/s, loss=0.00542, psnr=23.1, ssim=0.732]

Summary for Epoch 8/10:
    Train    avg_loss: 0.006736858496094876
    Test     avg_loss: 0.006095597948084981
                PSNR : 22.60657643600361
                SSIM : 0.679060697555542
```

สรุปผลการฝึกสอนโมเดล Autoencoder ใน Epoch ที่ 10 (จากทั้งหมด 10 Epoch):

- **Train avg_loss:** 0.00615
  - ค่าความผิดพลาดเฉลี่ยของข้อมูลฝึกสอนใน Epoch ที่ 10 มีค่าน้อยมาก ซึ่งหมายความว่าโมเดลสามารถทำนายค่าที่ใกล้เคียงกับข้อมูลจริงได้ดีขึ้นเรื่อยๆ
- **Test avg_loss:** 0.00584
  - ค่าความผิดพลาดเฉลี่ยของข้อมูลทดสอบก็มีค่าน้อยเช่นกัน แสดงให้เห็นว่าโมเดลสามารถทำงานได้ดีกับข้อมูลที่ไม่เคยเห็นมาก่อน
- **PSNR: 23.064**
  - ค่า Peak Signal-to-Noise Ratio (PSNR) เป็นตัวชี้วัดคุณภาพของภาพที่สร้างขึ้น โดยค่า PSNR ที่สูงขึ้นแสดงว่าภาพที่สร้างขึ้นมีคุณภาพดีขึ้น
- **SSIM: 0.673**
  - ค่า Structural Similarity Index Measure (SSIM) เป็นอีกหนึ่งตัวชี้วัดคุณภาพของภาพที่พิจารณาถึงโครงสร้างของภาพ ค่า SSIM ที่ใกล้เคียง 1 แสดงว่าภาพที่สร้างขึ้นมีความคล้ายคลึงกับภาพต้นฉบับมาก
- **Model saved at autoencoder.pth:** โมเดลที่ถูกฝึกแล้วถูกบันทึกไว้ในไฟล์ชื่อ autoencoder.pth เพื่อนำมาใช้งานในภายหลัง

Hyperparameter Grid Search with Raytune

```python
import ray
from ray import tune
from ray.air import session
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
from ray.air.config import RunConfig
from skimage.metrics import peak_signal_noise_ratio as psnr
from skimage.metrics import structural_similarity as ssim

ray.shutdown()
```

[10] Import library ที่จำเป็นสำหรับ lab นี้ โดยกำหนด ray.shutdown() เป็นฟังก์ชันที่ใช้สำหรับปิดระบบ Ray ทั้งหมดที่กำลังทำงานอยู่ และจะทำการยุติกระบวนการทำงานทั้งหมดที่เกี่ยวข้องกับ Ray

```python
def train_raytune(config):
    # Define data loaders (use your dataset)
    transform = transforms.Compose([
        transforms.Resize(128),  # Resize to 128x128
        transforms.ToTensor()
    ])

    trainloader = DataLoader(train_dataset, batch_size=config['batch_size'], shuffle=True)
    testloader = DataLoader(test_dataset, batch_size=config['batch_size'], shuffle=False)

    # Instantiate the model
    model = Autoencoder(channels=config['architecture'], input_channels=3, output_channels=3)
    model = model.to(config['device'])

    # Define optimizer
    if config['optimizer'] == 'Adam':
        optimizer = optim.Adam(model.parameters(), lr=config['lr'])
    elif config['optimizer'] == 'SGD':
        optimizer = optim.SGD(model.parameters(), lr=config['lr'], momentum=0.9)

    # Loss function
    criterion = nn.MSELoss()
```

```python
    # Training loop
    for epoch in range(config['num_epochs']):
        model.train()
        total_train_loss = 0
        for images, _ in trainloader:  # Assuming ground truth is the same input for autoencoder
            images = images.to(config['device'])

            # Forward pass
            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, images)
            loss.backward()
            optimizer.step()

            total_train_loss += loss.item()

        avg_train_loss = total_train_loss / len(trainloader)
```

```python
    # Validation Loop
    model.eval()
    total_val_loss = 0
    total_psnr = 0
    total_ssim = 0
    with torch.no_grad():
        for images, _ in testloader:
            images = images.to(config['device'])
            outputs = model(images)
            loss = criterion(outputs, images)
            total_val_loss += loss.item()

            # Convert images to numpy for PSNR and SSIM calculation
            images_np = images.cpu().numpy()
            outputs_np = outputs.cpu().numpy()

            # Calculate PSNR and SSIM for each image in the batch
            batch_psnr = 0
            batch_ssim = 0
            for i in range(images_np.shape[0]):
                img_gt = images_np[i].transpose(1, 2, 0)  # HWC format for PSNR and SSIM
                img_pred = outputs_np[i].transpose(1, 2, 0)

                # PSNR and SSIM calculation with data_range specified
                batch_psnr += psnr(img_gt, img_pred, data_range=img_gt.max() - img_gt.min())
                batch_ssim += ssim(img_gt, img_pred, data_range=1.0, win_size=3, channel_axis=-1)

            total_psnr += batch_psnr / images_np.shape[0]
            total_ssim += batch_ssim / images_np.shape[0]

    avg_val_loss = total_val_loss / len(testloader)
    avg_val_psnr = total_psnr / len(testloader)
    avg_val_ssim = total_ssim / len(testloader)
```

```python
    # Report metrics to Ray Tune
    session.report({
        "train_loss": avg_train_loss,
        "val_loss": avg_val_loss,
        "val_psnr": avg_val_psnr,
        "val_ssim": avg_val_ssim
    })
```

[11] ฟังก์ชันนี้ทำการฝึกโมเดล Autoencoder โดยใช้ Ray Tune เพื่อปรับแต่งไฮเปอร์พารามิเตอร์

- It includes a training loop that updates the model based on the training data and a validation loop that evaluates the model's performance on unseen data.
- The function computes and reports key metrics like training loss, validation loss, PSNR, and SSIM to Ray Tune for hyperparameter optimization.

Function: train_raytune

- **def train_raytune(config):**
  - Defines a function to train and evaluate the model using Ray Tune for hyperparameter optimization.
  - **config**: A dictionary containing the hyperparameters and configurations used for training.

Data Preparation

- **transform = transforms.Compose([...])**
  - Defines a transformation pipeline to preprocess the images:

        - **transforms.Resize(128)**: Resizes images to 128x128 pixels.
        - **transforms.ToTensor()**: Converts the images to PyTorch tensors.
- **trainloader = DataLoader(train_dataset, batch_size=config['batch_size'], shuffle=True)**
    - Creates a DataLoader for the training dataset:
        - **train_dataset**: The training dataset (assumed to be defined elsewhere).
        - **batch_size=config['batch_size']**: The batch size specified in the config dictionary.
        - **shuffle=True**: Shuffles the data before each epoch.
- **testloader = DataLoader(test_dataset, batch_size=config['batch_size'], shuffle=False)**
    - Creates a DataLoader for the testing dataset:
        - **test_dataset**: The testing dataset (assumed to be defined elsewhere).
        - **batch_size=config['batch_size']**: The batch size specified in the config dictionary.
        - **shuffle=False**: Does not shuffle the data.

Model Initialization

- **model = Autoencoder(channels=config['architecture'], input_channels=3, output_channels=3)**
    - Instantiates the Autoencoder model:
        - **channels=config['architecture']**: The architecture (list of channels) is specified in the config dictionary.
        - **input_channels=3**: The number of input channels (e.g., for RGB images).
        - **output_channels=3**: The number of output channels.
- **model = model.to(config['device'])**
    - Moves the model to the specified device (CPU or GPU) from the config dictionary.

Optimizer Initialization

- **if config['optimizer'] == 'Adam':**
    - Checks if the optimizer specified in config is Adam:
        - **optimizer = optim.Adam(model.parameters(), lr=config['lr'])**:
            - Initializes the Adam optimizer with the specified learning rate.
- **elif config['optimizer'] == 'SGD':**

- Checks if the optimizer specified in config is SGD:
  - **optimizer = optim.SGD(model.parameters(), lr=config['lr'], momentum=0.9)**:
    - Initializes the SGD optimizer with the specified learning rate and a momentum of 0.9.

Loss Function

- **criterion = nn.MSELoss()**
  - Initializes the Mean Squared Error (MSE) loss function, which measures the average squared difference between the predicted and actual values.

Training Loop

- **for epoch in range(config['num_epochs']):**
  - Loops over the specified number of epochs from the config dictionary.
- **model.train()**
  - Sets the model to training mode.
- **total_train_loss = 0**
  - Initializes a variable to accumulate the total training loss for the epoch.
- **for images, _ in trainloader:**
  - Iterates over the batches in the training DataLoader:
    - **images = images.to(config['device'])**: Moves the images to the specified device.
    - **optimizer.zero_grad()**: Clears the gradients of all optimized parameters.
    - **outputs = model(images)**: Performs a forward pass through the model to get the predictions.
    - **loss = criterion(outputs, images)**: Computes the loss between the predictions and ground truth.
    - **loss.backward()**: Performs backpropagation to compute gradients of the loss with respect to model parameters.
    - **optimizer.step()**: Updates model parameters based on the computed gradients.
    - **total_train_loss += loss.item()**: Accumulates the loss for this batch to the total training loss.
- **avg_train_loss = total_train_loss / len(trainloader)**

- Computes the average training loss for the epoch.

Validation Loop

- **model.eval()**
  - Sets the model to evaluation mode.
- **total_val_loss = 0**
  - Initializes a variable to accumulate the total validation loss for the epoch.
- **total_psnr = 0**
  - Initializes a variable to accumulate the total PSNR (Peak Signal-to-Noise Ratio) values for the epoch.
- **total_ssim = 0**
  - Initializes a variable to accumulate the total SSIM (Structural Similarity Index) values for the epoch.
- **with torch.no_grad():**
  - Disables gradient computation during validation to save memory and computations.
- **for images, _ in testloader:**
  - Iterates over the batches in the testing DataLoader:
    - **images = images.to(config['device'])**: Moves the images to the specified device.
    - **outputs = model(images)**: Performs a forward pass through the model to get the predictions.
    - **loss = criterion(outputs, images)**: Computes the loss between the predictions and ground truth.
    - **total_val_loss += loss.item()**: Accumulates the loss for this batch to the total validation loss.
    - **images_np = images.cpu().numpy()**: Converts the images tensor to a NumPy array for PSNR and SSIM calculation.
    - **outputs_np = outputs.cpu().numpy()**: Converts the output tensor to a NumPy array.
    - **batch_psnr = 0** and **batch_ssim = 0**: Initialize variables to accumulate PSNR and SSIM values for the current batch.
    - **for i in range(images_np.shape[0]):**

        - Iterates over each image in the batch:
            - **img_gt = images_np[i].transpose(1, 2, 0)**: Converts the ground truth image to HWC format.
            - **img_pred = outputs_np[i].transpose(1, 2, 0)**: Converts the predicted image to HWC format.
            - **batch_psnr += psnr(img_gt, img_pred, data_range=img_gt.max() - img_gt.min())**: Computes PSNR for the image.
            - **batch_ssim += ssim(img_gt, img_pred, data_range=1.0, win_size=3, channel_axis=-1)**: Computes SSIM for the image.
        - **total_psnr += batch_psnr / images_np.shape[0]**: Accumulates the average PSNR for the batch.
        - **total_ssim += batch_ssim / images_np.shape[0]**: Accumulates the average SSIM for the batch.
- **avg_val_loss = total_val_loss / len(testloader)**
    - Computes the average validation loss for the epoch.
- **avg_val_psnr = total_psnr / len(testloader)**
    - Computes the average PSNR for the epoch.
- **avg_val_ssim = total_ssim / len(testloader)**
    - Computes the average SSIM for the epoch.

Reporting to Ray Tune

- **session.report({...})**
    - Reports the metrics for the current epoch to Ray Tune:
        - **train_loss**: The average training loss for the epoch.
        - **val_loss**: The average validation loss for the epoch.
        - **val_psnr**: The average PSNR for the epoch.
        - **val_ssim**: The average SSIM for the epoch.

```python
# CODE FOR PRODUCTION USE
# ------------------------------------------------------------------------------------------
### START CODE HERE ###
relative_dir_result = "ray_results/grid_search/"
if not os.path.exists(relative_dir_result):
    os.makedirs(relative_dir_result)
absolute_dir_result = os.path.abspath(relative_dir_result)

# Define a simple trial directory name creator to shorten paths
def trial_dirname_creator(trial):
    return f"{trial.trainable_name}_{trial.trial_id}"

# Initialize Ray
ray.init(num_gpus=1)

# Fix the device key and ensure it's included in the config
config = {
    'architecture': tune.grid_search([
        [32, 64, 128],
        [64, 128, 256],
        [64, 128, 256, 512]
    ]),
    'lr': tune.grid_search([1e-3, 8e-4, 1e-4, 1e-2]),
    'batch_size': tune.grid_search([16, 32]),
    'num_epochs': tune.grid_search([10, 50, 100]),
    'optimizer': tune.grid_search(['Adam', 'SGD']),
    'input_channels': 3,
    'output_channels': 3,
    'device': 'cuda' if torch.cuda.is_available() else 'cpu'  # Make sure device is included
}

# Define Tuner with custom trial_dirname_creator to shorten directory paths
tuner = tune.Tuner(
    trainable=train_raytune,
    param_space=config,
    tune_config=tune.TuneConfig(
        metric="val_psnr",
        mode="max",
        num_samples=1,  # Reduce to 1 sample for fast iteration
        trial_dirname_creator=trial_dirname_creator
    ),
    run_config=RunConfig(
        storage_path=absolute_dir_result
    )
)

# Run the Tuner
result = tuner.fit()
### END CODE HERE ###
```

[12] This code sets up a grid search using Ray Tune to optimize the hyperparameters for an autoencoder model.

- The hyperparameter search space includes model architectures, learning rates, batch sizes, epochs, and optimizer types.
- The tuning process is configured to maximize the validation PSNR metric, with results stored in a specified directory.
- The code is designed for production use, including features like shortened trial directory names and the use of GPUs when available.

Setting Up the Directory

- **relative_dir_result = "ray_results/grid_search/"**
  - Specifies the relative directory where the Ray Tune results will be stored.
- **if not os.path.exists(relative_dir_result):**
  - Checks if the directory specified by relative_dir_result exists.

- **os.makedirs(relative_dir_result)**
    - Creates the directory if it doesn't already exist.
- **absolute_dir_result = os.path.abspath(relative_dir_result)**
    - Converts the relative directory path to an absolute path, ensuring the full path is used for storage.

Trial Directory Name Creation

- **def trial_dirname_creator(trial):**
    - Defines a custom function to create shorter directory names for trials:
        - **trial.trainable_name**: The name of the training function or model.
        - **trial.trial_id**: The unique ID assigned to each trial.
        - **return f"{trial.trainable_name}_{trial.trial_id}"**: Returns a string combining the trainable name and trial ID.

Ray Initialization

- **ray.init(num_gpus=1)**
    - Initializes Ray with access to 1 GPU. If more or fewer GPUs are available, adjust the num_gpus parameter accordingly.

Configuring the Hyperparameter Space

- **config = {...}**
    - Defines a dictionary config that contains the hyperparameter search space:
        - **'architecture'**: Uses tune.grid_search to explore different model architectures.
        - **'lr'**: Specifies a grid search over different learning rates.
        - **'batch_size'**: Specifies a grid search over different batch sizes.
        - **'num_epochs'**: Specifies a grid search over different numbers of epochs.
        - **'optimizer'**: Specifies a grid search over different optimizers (Adam and SGD).
        - **'input_channels'**: Fixed at 3 (for RGB images).
        - **'output_channels'**: Fixed at 3 (for RGB images).
        - **'device'**: Automatically selects 'cuda' if a GPU is available; otherwise, it defaults to 'cpu'.

Tuner Setup

- **tuner = tune.Tuner(...)**
    - Initializes a Ray Tune Tuner object to manage the hyperparameter tuning process:
        - **trainable=train_raytune**: Specifies the training function to be used in each trial.
        - **param_space=config**: Passes the hyperparameter search space defined in config.
        - **tune_config=tune.TuneConfig(...)**: Configures the tuning process:
            - **metric="val_psnr"**: Specifies that the PSNR (Peak Signal-to-Noise Ratio) on the validation set is the primary metric to optimize.
            - **mode="max"**: Indicates that higher PSNR values are better, so the tuning process will aim to maximize this metric.
            - **num_samples=1**: Specifies that only one sample (one run) will be taken for each combination of hyperparameters to reduce iteration time.
            - **trial_dirname_creator=trial_dirname_creator**: Uses the custom directory name creator to shorten trial paths.
        - **run_config=RunConfig(...)**: Configures the runtime environment:
            - **storage_path=absolute_dir_result**: Specifies where to store the results using the absolute path to the result directory.

Running the Tuner

- **result = tuner.fit()**
    - Executes the hyperparameter tuning process using the configurations specified above.
    - **result**: Captures the outcomes of the tuning process, which can be analyzed later.

```python
### START CODE HERE ###
import glob

relative_dir_result = "ray_results/grid_search/"
if not os.path.exists(relative_dir_result):
    os.makedirs(relative_dir_result)
absolute_dir_result = os.path.abspath(relative_dir_result)

# Get the most recently modified directory
latest_path = max(glob.glob(os.path.join(absolute_dir_result, '*/')), key=os.path.getmtime)

# Get the directory name (last part of the path)
latest_directory_name = os.path.basename(os.path.normpath(latest_path))

# Print the latest directory name
print(f"The latest directory in {relative_dir_result}: {latest_directory_name}")

# Load the checkpoint file
# Relative path
checkpoint_path = os.path.join(relative_dir_result, latest_directory_name)
absolute_checkpoint_path = os.path.abspath(checkpoint_path)

analysis = tune.ExperimentAnalysis(absolute_checkpoint_path)

# Get all trials
all_trials = analysis.trials

# Filter out completed trials (status is TERMINATED or COMPLETE)
valid_trials = [trial for trial in all_trials if trial.status in ('TERMINATED', 'COMPLETE')]

# Print the number of all trials and the number of valid completed trials
num_all_trials = len(all_trials)
num_completed_trials = len(valid_trials)
print(f"Number of completed trials: {num_completed_trials} from all trials: {num_all_trials}")
```

```python
# Check if there are any valid trials
if valid_trials:
    # Find the best trial using the specified metric and mode
    best_trial = analysis.get_best_trial(metric="val_psnr", mode="max")

    if best_trial is not None:
        best_config = best_trial.config
        print(f"Best Trial: {best_trial}")
    else:
        print("No best trial found for the specified metric and mode.")
else:
    print("No valid completed trials found.")

### END CODE HERE ###
```

[13] วิเคราะห์ผลการทดลองที่ดำเนินการโดย Ray Tune

(เนื่องจากกลุ่มของผมทดลอง run Ray tune แล้วพบปัญหาว่าเมื่อ run ได้ถึง 30 กว่าชั่วโมงแล้ว หน้าจอ vscode ที่ ssh ไปที่ apex ได้มีกล่องให้ใส่รหัสเข้า apex ใหม่ เมื่อกรอก พบว่าระบบได้มีการสั่งให้ reload window ทำให้ vscode ของกลุ่มผม reload หน้าใหม่ทำให้การ tune ไม่ run ต่อจากเดิมจะต้อง run ใหม่ทุกครั้ง(เป็นมา 3 ครั้งแล้ว) จึงได้ไปปรึกษาอาจารย์ในคาบว่าสามารถส่ง code ได้ไหมหาก run ไม่ครบ trials ที่กำหนดไว้ ซึ่งอาจารย์ให้คำตอบว่าได้ ต้องมี trials จำนวนนึง ซึ่งกลุ่มของผมได้ใส่รายละเอียดแนบ code ในช่วงของ grid search และ random search ครับ)

ขั้นตอนการทำงาน:

- **กำหนดตำแหน่งผลลัพธ์:** กำหนดตำแหน่งที่เก็บผลลัพธ์ของการทดลอง Ray Tune
- **ค้นหาไดเรกทอรี่ล่าสุด:** ค้นหาไดเรกทอรี่ที่ถูกสร้างขึ้นล่าสุดในตำแหน่งที่กำหนด เนื่องจาก Ray Tune จะสร้างไดเรกทอรี่ใหม่สำหรับแต่ละการทดลอง
- **โหลดข้อมูลการทดลอง:** โหลดข้อมูลการทดลองจากไดเรกทอรี่ที่พบ
- **กรองผลลัพธ์:** กรองเฉพาะการทดลองที่เสร็จสิ้นแล้ว (STATUS เป็น TERMINATED หรือ COMPLETE)
- **ค้นหาการทดลองที่ดีที่สุด:** หากมีการทดลองที่เสร็จสิ้นแล้ว ก็จะค้นหาการทดลองที่ดีที่สุดตามเมตริกที่กำหนด (ในกรณีนี้คือ val_psnr) และโหมดการประเมิน (ในกรณีนี้คือ max)

ซึ่งผลลัพธ์นั้นคือ:

- **Grid Search: Ray tune ของกลุ่มผม run ได้ 51 trials จากทั้งหมด 144 trials ครับ**

```
The latest directory in ray_results/grid_search/: train_raytune_2024-09-25_15-05-14
```

```
Number of completed trials: 51 from all trials: 144
Best Trial: train_raytune_e5418_00049
```

ซึ่ง trails ที่ดีที่สุดอยู่ที่ trails ที่ 49

```
print("🎉 [INFO] Training is done!")
print("Best config is:", best_trial.config)
print("Best result is:", best_trial.last_result)

# Use the analysis object to create a DataFrame
df = analysis.dataframe()  # Get the DataFrame of all trials

# Define the folder and CSV name
csv_name = "grid_search.csv"

# Create the full path for the CSV file
csv_path = os.path.join(relative_dir_result, csv_name)

# Ensure the folder exists
os.makedirs(relative_dir_result, exist_ok=True)

# Save the DataFrame to a CSV file
df.to_csv(csv_path, index=False)

print(f"Results saved to: {csv_path}")
```

[14] This code finalizes the hyperparameter tuning process by printing the best configuration and result obtained.

- It then saves all the results from the tuning run into a CSV file within the specified directory, making it easy to analyze or share the results later.
- The CSV file is stored with the name grid_search.csv in the directory relative_dir_result.

ขั้นตอนการทำงาน:

- **แสดงผลลัพธ์ที่ดีที่สุด:** แสดงค่าพารามิเตอร์ที่ดีที่สุด (best_trial.config) และผลลัพธ์ที่ดีที่สุด (best_trial.last_result)
- **สร้าง DataFrame:** สร้าง DataFrame จากข้อมูลการทดลองทั้งหมด
- **กำหนดชื่อไฟล์ CSV:** กำหนดชื่อไฟล์ CSV ที่ต้องการบันทึก
- **สร้างไดเรกทอรี่ (ถ้ายังไม่มี):** สร้างไดเรกทอรี่ที่ต้องการบันทึกไฟล์ CSV (ถ้ายังไม่มี)
- **บันทึกข้อมูลเป็น CSV:** บันทึก DataFrame ลงในไฟล์ CSV

ผลลัพธ์:


```
[INFO] Training is done!
Best config is: {'architecture': [64, 128, 256], 'lr': 0.001, 'batch_size': 16, 'num_epochs': 100, 'optimizer': 'Adam', 'input_channels': 3, 'output_channels': 3, 'device': 'cuda'}
Best result is: {'train_loss': 0.0017341222872593002, 'val_loss': 0.0017634813756918532, 'val_psnr': 28.44117781983429, 'val_ssim': 0.7630671396699352, 'timestamp': 1727334591, 'checkpoint_dir_name': None, 'done': True, 'training_iteration': 100, 'trial_id': 'e5418_00049', 'date': '2024-09-26_14-09-51', 'time_this_iter_s': 148.1607837677002, 'time_total_s': 14642.166303157806, 'pid': 2115095, 'hostname': 'prism-4.apex.cmkl.ac.th', 'node_ip': '172.16.101.114', 'config': {'architecture': [64, 128, 256], 'lr': 0.001, 'batch_size': 16, 'num_epochs': 100, 'optimizer': 'Adam', 'input_channels': 3, 'output_channels': 3, 'device': 'cuda'}, 'time_since_restore': 14642.166303157806, 'iterations_since_restore': 100, 'experiment_tag': '49_architecture=64_128_256,batch_size=16,lr=0.0010,num_epochs=100,optimizer=Adam'}
Results saved to: ray_results/grid_search/grid_search.csv
```


ค่าปรับแต่งที่ดีที่สุด (Best config):

- **architecture:** [64, 128, 256] (ขนาดของแต่ละเลเยอร์ในเครือข่าย)
- **lr:** 0.001 (อัตราการเรียนรู้)
- **batch_size:** 16 (จำนวนข้อมูลที่ป้อนเข้าเครือข่ายในแต่ละรอบ)
- **num_epochs:** 100 (จำนวนรอบการฝึกทั้งหมด)
- **optimizer:** 'Adam'
- **input_channels:** 3 (จำนวนช่องสัญญาณของข้อมูลอินพุต - น่าจะเป็นภาพ 3 ช่อง RGB)
- **output_channels:** 3 (จำนวนช่องสัญญาณของข้อมูลเอาต์พุต - น่าจะเป็นภาพ 3 ช่อง RGB)
- **device:** 'cuda' (ใช้การประมวลผลแบบ GPU)

ผลลัพธ์ที่ดีที่สุด (Best result):

- **train_loss: 0.0017 (ค่าความผิดพลาดบนข้อมูลฝึก)**
- **val_loss:** 0.0018 (ค่าความผิดพลาดบนข้อมูลทดสอบ)

- **val_psnr:** 28.44 (ค่า PSNR บนข้อมูลทดสอบ - ค่า PSNR สูง แสดงว่าคุณภาพของภาพที่สร้างขึ้นดี)
- **val_ssim:** 0.763 (ค่า SSIM บนข้อมูลทดสอบ - ค่า SSIM ใกล้เคียง 1 แสดงว่าภาพที่สร้างขึ้นมีความคล้ายคลึงกับภาพต้นฉบับมาก)
- **num_epochs:** 100 (จำนวนรอบการฝึกทั้งหมด - ตรงกับค่าปรับแต่ง)
- **experiment_tag:** แท็กการทดลองที่ระบุค่าปรับแต่งที่ใช้

```
### START CODE HERE ###
# Train the Autoencoder with the best hyperparameters
best_config = best_trial.config
model = Autoencoder(channels=best_config['architecture'], input_channels=3, output_channels=3)
opt = optim.Adam(model.parameters(), lr=best_config['lr']) if best_config['optimizer'] == 'Adam' else optim.SGD(model.parameters(), lr=best_config['lr'], momentum=0.9)
loss_fn = nn.MSELoss()

# Define the filename
checkpoint_filename = "best_autoencoder_grid_search.pth"

# Create the full path for the checkpoint file
checkpoint_path = os.path.join(absolute_dir_result, checkpoint_filename)

train(model, opt, loss_fn, trainloader, testloader, epochs=best_config['num_epochs'], checkpoint_path=checkpoint_path, device=best_config['device'])
### END CODE HERE ###
```

[15] โค้ดนี้จะฝึกสอนโมเดล Autoencoder โดยใช้ค่าพารามิเตอร์ที่ดีที่สุดที่พบจากการทดลอง Ray Tune

- It initializes the model and optimizer according to the best configuration, sets up the loss function, and defines where the final model checkpoint will be saved.
- The model is then trained with the optimal configuration, and the trained model is saved to the specified checkpoint file.

ผลลัพธ์:

```
Training on cuda

Training Epoch [1/100]: 100%| 1313/1313 [02:24<00:00,  9.09batch/s, loss=0.0113]
Testing: 100%| 563/563 [01:16<00:00,  7.36batch/s, loss=0.0113, psnr=20.2, ssim=0.658]

Summary for Epoch 1/100:
    Train     avg_loss: 0.01687523798048156
    Test      avg_loss: 0.012029286222548934
                PSNR : 19.646245028796308
                SSIM : 0.5983115434646606

Training Epoch [2/100]: 100%| 1313/1313 [01:42<00:00, 12.79batch/s, loss=0.0114]
Testing: 100%| 563/563 [01:04<00:00,  8.79batch/s, loss=0.00684, psnr=22.2, ssim=0.638]

Summary for Epoch 2/100:
    Train     avg_loss: 0.011010194173180594
    Test      avg_loss: 0.009046524790401833
                PSNR : 20.940815037309417
                SSIM : 0.6023789048194885
```

```
🚀Training Epoch [10/100]: 100%|████████████████████████████████████████| 1313/1313 [01:41<00:00, 12.88batch/s, loss=0.00642]
📄Testing: 100%|████████████████████████████████████████| 563/563 [01:05<00:00,  8.64batch/s, loss=0.00877, psnr=22, ssim=0.712]

Summary for Epoch 10/100:
    Train    avg_loss: 0.006151612866840127
    Test     avg_loss: 0.00675755416640274
               PSNR : 22.810635853413356
               SSIM : 0.6801338791847229

🚀Training Epoch [11/100]: 100%|████████████████████████████████████████| 1313/1313 [01:39<00:00, 13.15batch/s, loss=0.0207]
📄Testing: 100%|████████████████████████████████████████| 563/563 [01:06<00:00,  8.48batch/s, loss=0.00549, psnr=22.9, ssim=0.726]

Summary for Epoch 11/100:
    Train    avg_loss: 0.00590441801839899
    Test     avg_loss: 0.008367945781708769
               PSNR : 21.171706634392226
               SSIM : 0.6722270250320435
```

```
🚀Training Epoch [20/100]: 100%|████████████████████████████████████████| 1313/1313 [01:47<00:00, 12.26batch/s, loss=0.00436]
📄Testing: 100%|████████████████████████████████████████| 563/563 [01:06<00:00,  8.47batch/s, loss=0.00313, psnr=25.4, ssim=0.763]

Summary for Epoch 20/100:
    Train    avg_loss: 0.004896111146391562
    Test     avg_loss: 0.00447762765069732
               PSNR : 24.112640830720217
               SSIM : 0.7114842534065247

🚀Training Epoch [21/100]: 100%|████████████████████████████████████████| 1313/1313 [01:46<00:00, 12.36batch/s, loss=0.00546]
📄Testing: 100%|████████████████████████████████████████| 563/563 [01:05<00:00,  8.56batch/s, loss=0.00611, psnr=22.9, ssim=0.708]

Summary for Epoch 21/100:
    Train    avg_loss: 0.0047996125711725355
    Test     avg_loss: 0.004817051892408681
               PSNR : 24.03193452412464
               SSIM : 0.7077919840812683
```

```
Summary for Epoch 30/100:
    Train    avg_loss: 0.004372806792797894
    Test     avg_loss: 0.004275832418269654
               PSNR : 24.197417340508746
               SSIM : 0.7187628746032715

🚀Training Epoch [31/100]: 100%|████████████████████████████████████████| 1313/1313 [01:42<00:00, 12.78batch/s, loss=0.00727]
📄Testing: 100%|████████████████████████████████████████| 563/563 [01:05<00:00,  8.55batch/s, loss=0.00492, psnr=23.8, ssim=0.762]

Summary for Epoch 31/100:
    Train    avg_loss: 0.004418859829642418
    Test     avg_loss: 0.0044853540164704695
               PSNR : 24.25063648991799
               SSIM : 0.7054071426391602
```

```
🚀Training Epoch [40/100]: 100%|████████████████████████████████████████| 1313/1313 [02:40<00:00,  8.19batch/s, loss=0.00529]
📄Testing: 100%|████████████████████████████████████████| 563/563 [01:14<00:00,  7.55batch/s, loss=0.00447, psnr=23.8, ssim=0.747]

Summary for Epoch 40/100:
    Train    avg_loss: 0.0041982495410031635
    Test     avg_loss: 0.004804091310942443
               PSNR : 23.616863092500186
               SSIM : 0.7110446691513062

🚀Training Epoch [41/100]: 100%|████████████████████████████████████████| 1313/1313 [01:53<00:00, 11.54batch/s, loss=0.00493]
📄Testing: 100%|████████████████████████████████████████| 563/563 [01:06<00:00,  8.51batch/s, loss=0.00231, psnr=26.7, ssim=0.779]

Summary for Epoch 41/100:
    Train    avg_loss: 0.004163268658843938
    Test     avg_loss: 0.003980265355869122
               PSNR : 24.5376702529787
               SSIM : 0.7226147651672363
```

```
🚀Training Epoch [50/100]: 100%|████████████████████████████████████████| 1313/1313 [02:27<00:00,  8.93batch/s, loss=0.00359]
📄Testing: 100%|████████████████████████████████████████| 563/563 [01:21<00:00,  6.93batch/s, loss=0.0035, psnr=25.3, ssim=0.754]

Summary for Epoch 50/100:
    Train    avg_loss: 0.003967925424028617
    Test     avg_loss: 0.003931171825688273
                PSNR : 24.61926238772708
                SSIM : 0.7264600992202759

🚀Training Epoch [51/100]: 100%|████████████████████████████████████████| 1313/1313 [01:55<00:00, 11.37batch/s, loss=0.00346]
📄Testing: 100%|████████████████████████████████████████| 563/563 [01:02<00:00,  8.94batch/s, loss=0.0037, psnr=24.7, ssim=0.758]

Summary for Epoch 51/100:
    Train    avg_loss: 0.003959680711127029
    Test     avg_loss: 0.004005944779801114
                PSNR : 24.664197765971053
                SSIM : 0.7265975475311279
```

```
🚀Training Epoch [60/100]: 100%|████████████████████████████████████████| 1313/1313 [01:40<00:00, 13.11batch/s, loss=0.00242]
📄Testing: 100%|████████████████████████████████████████| 563/563 [01:03<00:00,  8.86batch/s, loss=0.00539, psnr=23.5, ssim=0.753]

Summary for Epoch 60/100:
    Train    avg_loss: 0.0038203808871968186
    Test     avg_loss: 0.004250013446071405
                PSNR : 24.42205224534084
                SSIM : 0.7219266891479492

🚀Training Epoch [61/100]: 100%|████████████████████████████████████████| 1313/1313 [02:19<00:00,  9.38batch/s, loss=0.00334]
📄Testing: 100%|████████████████████████████████████████| 563/563 [01:13<00:00,  7.64batch/s, loss=0.00346, psnr=25.1, ssim=0.778]

Summary for Epoch 61/100:
    Train    avg_loss: 0.003840124720256415
    Test     avg_loss: 0.0037276625664669815
                PSNR : 24.952665502905237
                SSIM : 0.735520601272583
```

(Window Reload เสียก่อนทำให้ code ไม่ run ต่อครับ)

```python
class FeatureMapVisualizer:
    def __init__(self, model, layers, save_dir):
        self.model = model
        self.layers = layers if isinstance(layers, list) else [layers]
        self.activations = {}
        self.save_dir = save_dir

        os.makedirs(self.save_dir, exist_ok=True)
        self._register_hooks()

    def _register_hooks(self):
        for name, layer in self.model.named_modules():
            if name in self.layers:
                layer.register_forward_hook(self._hook_fn(name))

    def _hook_fn(self, layer_name):
        def hook(module, input, output):
            print(f'Hooking layer: {layer_name}')
            self.activations[layer_name] = output.detach()
        return hook

    def visualize(self, input_images):
        for img_tensor in input_images:
            self.model(img_tensor.unsqueeze(0))  # Add batch dimension

            for layer_name, activation in self.activations.items():
                print(f'Visualizing and saving layer: {layer_name}')
                self._save_feature_maps(activation, layer_name)

    def _save_feature_maps(self, activation, layer_name):
        num_channels = activation.shape[1]
        grid_dim = math.ceil(math.sqrt(num_channels))  # Create a grid
        fig, axes = plt.subplots(grid_dim, grid_dim, figsize=(10, 10))

        for i in range(num_channels):
            ax = axes[i // grid_dim, i % grid_dim]
            ax.imshow(activation[0, i].cpu().numpy(), cmap='gray')
            ax.axis('off')

        plt.tight_layout()
        plt.savefig(os.path.join(self.save_dir, f'{layer_name}_feature_map.png'))
        plt.close()
```

[16] The FeatureMapVisualizer class is designed to visualize and save the feature maps (activations) of specific layers in a neural network model.

- It registers forward hooks to capture the output of the specified layers during the forward pass and saves these outputs as images.
- The class is useful for understanding how different layers of a model process input data, especially in deep convolutional neural networks.

```
# Store the layer names for encoder and decoder
layers_to_visualize = []

# Loop through the named children of the model to collect the layers to visualize
for name, layer in model.named_children():
    if name in ['encoder', 'decoder']:  # If the layer is either encoder or decoder
        for i, sub_layer in enumerate(layer):
            layer_name = f"{name}.{i}"  # Generate the layer name dynamically
            layers_to_visualize.append(layer_name)  # Store the layer names

# Print the layers to visualize for reference
print(f"Layers to visualize: {layers_to_visualize}")
```

```
Layers to visualize: ['encoder.0', 'encoder.1', 'encoder.2', 'decoder.0', 'decoder.1', 'decoder.2']
```

[17] This code dynamically collects the names of layers within the encoder and decoder of the model that you intend to visualize.

It ensures that you can easily reference and visualize specific layers by storing their names in a list.

The list layers_to_visualize will contain names like encoder.0, decoder.1, etc., which correspond to the layers within the encoder and decoder that you want to hook for visualization.

```
### START CODE HERE ###
# Load and preprocess images using OpenCV and transforms from Torchvision
input_images = [transforms.ToTensor()(cv2.cvtColor(cv2.imread(path), cv2.COLOR_BGR2RGB)) for path in image_paths[:1]]

relative_dir = "feature_maps/grid_search"
if not os.path.exists(relative_dir):
    os.makedirs(relative_dir)
absolute_dir_result = os.path.abspath(relative_dir)

# Visualize feature maps
visualizer = FeatureMapVisualizer(model, layers=layers_to_visualize, save_dir=absolute_dir_result)
visualizer.visualize(input_images)

### END CODE HERE ###
```

```
Hooking layer: encoder.0
Hooking layer: encoder.1
Hooking layer: encoder.2
Hooking layer: decoder.0
Hooking layer: decoder.1
Hooking layer: decoder.2
Visualizing and saving layer: encoder.0
Visualizing and saving layer: encoder.1
Visualizing and saving layer: encoder.2
Visualizing and saving layer: decoder.0
Visualizing and saving layer: decoder.1
Visualizing and saving layer: decoder.2
```

[18] โค้ดนี้จะโหลดภาพและแสดงภาพแผนที่ลักษณะ (Feature Map) ของแต่ละเลเยอร์ที่เลือกไว้ในโมเดล Autoencoder

- It sets up a directory for saving the visualized feature maps, ensuring that the results are stored in a specific location.
- The FeatureMapVisualizer is then used to generate and save the feature maps of the specified layers in the model, using the preprocessed images as input.
- This process provides insight into the internal workings of the model by visually inspecting the feature maps at different layers.

Hyperparameter Random Search with Raytune

```python
relative_dir_result = "ray_results/random_search/"
if not os.path.exists(relative_dir_result):
    os.makedirs(relative_dir_result)
absolute_dir_result = os.path.abspath(relative_dir_result)

Define a simple trial directory name creator to shorten paths
def trial_dirname_creator(trial):
    return f"{trial.trainable_name}_{trial.trial_id}"

ray.init(num_gpus=1)

# Define hyperparameter search space for random search
search_space = {
    'architecture': tune.choice([
        [32, 64, 128],
        [64, 128, 256],
        [64, 128, 256, 512]
    ]),
    'lr': tune.uniform(1e-4, 1e-2),
    'batch_size': tune.randint(16, 32),
    'num_epochs': tune.randint(10, 100),
    'optimizer': tune.choice(['Adam', 'SGD']),
    'device': 'cuda' if torch.cuda.is_available() else 'cpu'
}


# Set the trial directory creator to shorten the trial path names
tuner = tune.Tuner(
    trainable=train_raytune,
    param_space=search_space,
    tune_config=tune.TuneConfig(
        num_samples=80,  # Number of random search samples
        metric="val_psnr",
        mode="max",
        trial_dirname_creator=trial_dirname_creator  # Shorten trial directory names
    ),
    run_config=RunConfig(
        storage_path=absolute_dir_result
    )
)

# Run the random search
result = tuner.fit()
```

[19] This code sets up a random search using Ray Tune to optimize the hyperparameters for an autoencoder model.

- The hyperparameter search space includes model architectures, learning rates, batch sizes, epochs, and optimizer types, sampled randomly within specified ranges.
- The tuning process is configured to maximize the validation PSNR metric, with results stored in a specified directory.
- The code is designed for production use, including features like shortened trial directory names and the use of GPUs when available.

Setting Up the Directory

- **relative_dir_result = "ray_results/random_search/"**
  - Specifies the relative directory where the Ray Tune results will be stored for the random search.
- **if not os.path.exists(relative_dir_result):**
  - Checks if the directory specified by relative_dir_result exists.
- **os.makedirs(relative_dir_result)**
  - Creates the directory if it doesn't already exist.
- **absolute_dir_result = os.path.abspath(relative_dir_result)**
  - Converts the relative directory path to an absolute path, ensuring the full path is used for storage.

Trial Directory Name Creation

- **def trial_dirname_creator(trial):**
  - Defines a custom function to create shorter directory names for trials:
    - **trial.trainable_name**: The name of the training function or model.
    - **trial.trial_id**: The unique ID assigned to each trial.
    - **return f"{trial.trainable_name}_{trial.trial_id}"**: Returns a string combining the trainable name and trial ID.

Ray Initialization

- **ray.init(num_gpus=1)**
  - Initializes Ray with access to 1 GPU. If more or fewer GPUs are available, adjust the num_gpus parameter accordingly.

Defining the Search Space

- **search_space = {...}**
  - Defines the hyperparameter search space for random search:
    - **'architecture': tune.choice([...])**: Specifies different model architectures to choose from.
    - **'lr': tune.uniform(1e-4, 1e-2)**: Specifies a uniform distribution for the learning rate between 1e-4 and 1e-2.

- **'batch_size': tune.randint(16, 32)**: Specifies a random integer batch size between 16 and 32.
- **'num_epochs': tune.randint(10, 100)**: Specifies a random integer number of epochs between 10 and 100.
- **'optimizer': tune.choice(['Adam', 'SGD'])**: Specifies a choice between the Adam and SGD optimizers.
- **'device': 'cuda' if torch.cuda.is_available() else 'cpu'**: Automatically selects 'cuda' if a GPU is available; otherwise, defaults to 'cpu'.

Tuner Setup

- **tuner = tune.Tuner(...)**
  - Initializes a Ray Tune Tuner object to manage the random search process:
    - **trainable=train_raytune**: Specifies the training function to be used in each trial.
    - **param_space=search_space**: Passes the hyperparameter search space defined in search_space.
    - **tune_config=tune.TuneConfig(...)**: Configures the tuning process:
      - **num_samples=80**: Specifies that 80 random samples will be taken from the search space.
      - **metric="val_psnr"**: Specifies that the PSNR (Peak Signal-to-Noise Ratio) on the validation set is the primary metric to optimize.
      - **mode="max"**: Indicates that higher PSNR values are better, so the tuning process will aim to maximize this metric.
      - **trial_dirname_creator=trial_dirname_creator**: Uses the custom directory name creator to shorten trial paths.
    - **run_config=RunConfig(...)**: Configures the runtime environment:
      - **storage_path=absolute_dir_result**: Specifies where to store the results using the absolute path to the result directory.

Running the Random Search

- **result = tuner.fit()**
  - Executes the random search process using the configurations specified above.

- **result**: Captures the outcomes of the tuning process, which can be analyzed later.

```
### START CODE HERE ###
import glob

relative_dir_result = "ray_results/random_search/"
if not os.path.exists(relative_dir_result):
    os.makedirs(relative_dir_result)
absolute_dir_result = os.path.abspath(relative_dir_result)

# Get the most recently modified directory
latest_path = max(glob.glob(os.path.join(absolute_dir_result, '*/')), key=os.path.getmtime)

# Get the directory name (last part of the path)
latest_directory_name = os.path.basename(os.path.normpath(latest_path))

# Print the latest directory name
print(f"The latest directory in {relative_dir_result}: {latest_directory_name}")

# Load the checkpoint file
# Relative path
checkpoint_path = os.path.join(relative_dir_result, latest_directory_name)
absolute_checkpoint_path = os.path.abspath(checkpoint_path)

analysis = tune.ExperimentAnalysis(absolute_checkpoint_path)

# Get all trials
all_trials = analysis.trials

# Filter out completed trials (status is TERMINATED or COMPLETE)
valid_trials = [trial for trial in all_trials if trial.status in ('TERMINATED', 'COMPLETE')]
```

```
# Print the number of all trials and the number of valid completed trials
num_all_trials = len(all_trials)
num_completed_trials = len(valid_trials)
print(f"Number of completed trials: {num_completed_trials} from all trials: {num_all_trials}")

# Check if there are any valid trials
if valid_trials:
    # Find the best trial using the specified metric and mode
    best_trial = analysis.get_best_trial(metric="val_psnr", mode="max")

    if best_trial is not None:
        best_config = best_trial.config
        print(f"Best Trial: {best_trial}")
    else:
        print("No best trial found for the specified metric and mode.")
else:
    print("No valid completed trials found.")

### END CODE HERE ###
```

[20] วิเคราะห์ผลการทดลองที่ดำเนินการโดย Ray Tune

(เนื่องจากกลุ่มของผมทดลอง run Ray tune แล้วพบปัญหาว่าเมื่อ run ได้ถึง 30 กว่าชั่วโมงแล้ว หน้าจอ vscode ที่ ssh ไปที่ apex ได้มีกล่องให้ใส่รหัสเข้า apex ใหม่ เมื่อกรอก พบว่าระบบได้มีการสั่งให้ reload window ทำให้ vscode ของกลุ่มผม reload หน้าใหม่ทำให้การ tune ไม่ run ต่อจากเดิมจะต้อง run ใหม่ทุกครั้ง(เป็นมา 3 ครั้งแล้ว) จึงได้ไปปรึกษาอาจารย์ในคาบว่าสามารถส่ง code ได้ไหมหาก run ไม่ครบ trials ที่กำหนดไว้ ซึ่งอาจารย์ให้คำตอบว่าได้ ต้องมี trials จำนวนนึง ซึ่งกลุ่มของผมได้ใส่รายละเอียดแนบ code ในช่วงของ grid search และ random search ครับ)

ขั้นตอนการทำงาน:

- **กำหนดตำแหน่งผลลัพธ์:** กำหนดตำแหน่งที่เก็บผลลัพธ์ของการทดลอง Ray Tune
- **ค้นหาไดเรกทอรี่ล่าสุด:** ค้นหาไดเรกทอรี่ที่ถูกสร้างขึ้นล่าสุดในตำแหน่งที่กำหนด เนื่องจาก Ray Tune จะสร้างไดเรกทอรี่ใหม่สำหรับแต่ละการทดลอง
- **โหลดข้อมูลการทดลอง:** โหลดข้อมูลการทดลองจากไดเรกทอรี่ที่พบ
- **กรองผลลัพธ์:** กรองเฉพาะการทดลองที่เสร็จสิ้นแล้ว (STATUS เป็น TERMINATED หรือ COMPLETE)
- **ค้นหาการทดลองที่ดีที่สุด:** หากมีการทดลองที่เสร็จสิ้นแล้ว ก็จะค้นหาการทดลองที่ดีที่สุดตามเมตริกที่กำหนด (ในกรณีนี้คือ val_psnr) และโหมดการประเมิน (ในกรณีนี้คือ max)

ซึ่งผลลัพธ์นั้นคือ:

- **Radom Search: Ray tune ของกลุ่มผม run ได้ 62 trials จากทั้งหมด 80 trials ครับ**

```
The latest directory in ray_results/random_search/: train_raytune_2024-09-24_23-33-07
```

```
Number of completed trials: 62 from all trials: 80
Best Trial: train_raytune_ae0d2_00042
```

ซึ่ง trails ที่ดีที่สุดอยู่ที่ trails ที่ 42

```python
print("🎉 [INFO] Training is done!")
print("Best config is:", best_trial.config)
print("Best result is:", best_trial.last_result)
df = analysis.dataframe()  # Get the DataFrame of all trials


relative_dir_result = "ray_results/random_search"
absolute_dir_result = os.path.abspath(relative_dir_result)

# Define the folder and CSV name
csv_name = "random_search.csv"

# Create the full path for the CSV file
csv_path = os.path.join(absolute_dir_result, csv_name)

# Ensure the folder exists
os.makedirs(absolute_dir_result, exist_ok=True)

# Save the DataFrame to a CSV file
df.to_csv(csv_path, index=False)

print(f"Results saved to: {csv_path}")

# ray.shutdown()
```

[21] This code finalizes the hyperparameter tuning process by printing the best configuration and result obtained.

- It then saves all the results from the tuning run into a CSV file within the specified directory, making it easy to analyze or share the results later.
- The CSV file is stored with the name ransom_search.csv in the directory relative_dir_result.

```
### START CODE HERE ###
# Get the best hyperparameters from the search
best_config = best_trial.config

# Initialize model, optimizer, and loss function with best hyperparameters
model = Autoencoder(channels=best_config['architecture'], input_channels=3, output_channels=3)
if best_config['optimizer'] == 'Adam':
    opt = optim.Adam(model.parameters(), lr=best_config['lr'])
else:
    opt = optim.SGD(model.parameters(), lr=best_config['lr'], momentum=0.9)
loss_fn = nn.MSELoss()

# Define the filename
checkpoint_filename = "best_autoencoder_random_search.pth"

# Create the full path for the checkpoint file
checkpoint_path = os.path.join(absolute_dir_result, checkpoint_filename)

# Train the model with the best configuration
train(model, opt, loss_fn, trainloader, testloader, epochs=best_config['num_epochs'], checkpoint_path=checkpoint_path, device=best_config['device'])
### END CODE HERE ###
```

[22] โค้ดนี้จะฝึกสอนโมเดล Autoencoder โดยใช้ค่าพารามิเตอร์ที่ดีที่สุดที่พบจากการทดลอง Ray Tune

- The model and optimizer are initialized with the best configuration, and the training process is executed with the optimal settings.
- The final trained model is saved to a checkpoint file (best_autoencoder_random_search.pth) in the specified results directory. This ensures that the best-performing model can be easily restored for further use or analysis.

ผลลัพธ์:

```
Training on cuda

Training Epoch [1/95]: 100%| | 1313/1313 [02:25<00:00,  9.04batch/s, loss=0.0186]
Testing: 100%| | 563/563 [01:04<00:00,  8.78batch/s, loss=0.0125, psnr=19.5, ssim=0.616]

Summary for Epoch 1/95:
    Train     avg_loss: 0.019449916197964462
    Test      avg_loss: 0.012506942041416787
                PSNR : 19.491403681505368
                SSIM : 0.5619143843650818

Training Epoch [2/95]: 100%| | 1313/1313 [01:40<00:00, 13.11batch/s, loss=0.0214]
Testing: 100%| | 563/563 [01:03<00:00,  8.86batch/s, loss=0.019, psnr=18.2, ssim=0.573]

Summary for Epoch 2/95:
    Train     avg_loss: 0.012493315771341279
    Test      avg_loss: 0.011874136160959888
                PSNR : 19.85780812664366
                SSIM : 0.5731703042984009
```

```
🚀Training Epoch [10/95]: 100%|████████████████████████████████████████████████████████| 1313/1313 [01:43<00:00, 12.67batch/s, loss=0.00983]
📄Testing: 100%|████████████████████████████████████████████████████████| 563/563 [01:10<00:00,  8.02batch/s, loss=0.00437, psnr=23.8, ssim=0.732]

Summary for Epoch 10/95:
    Train    avg_loss: 0.0068229275396673944
    Test     avg_loss: 0.006205345631773926
                PSNR : 22.69833925123382
                SSIM : 0.680097758769989

🚀Training Epoch [11/95]: 100%|████████████████████████████████████████████████████████| 1313/1313 [01:43<00:00, 12.70batch/s, loss=0.00555]
📄Testing: 100%|████████████████████████████████████████████████████████| 563/563 [01:09<00:00,  8.07batch/s, loss=0.00615, psnr=22.3, ssim=0.714]

Summary for Epoch 11/95:
    Train    avg_loss: 0.0062853542084877305
    Test     avg_loss: 0.006969662134239506
                PSNR : 21.93993112106428
                SSIM : 0.6652563810348511
```

```
🚀Training Epoch [20/95]: 100%|████████████████████████████████████████████████████████| 1313/1313 [01:35<00:00, 13.74batch/s, loss=0.0048]
📄Testing: 100%|████████████████████████████████████████████████████████| 563/563 [01:00<00:00,  9.29batch/s, loss=0.0045, psnr=23.8, ssim=0.745]

Summary for Epoch 20/95:
    Train    avg_loss: 0.005262187643966837
    Test     avg_loss: 0.00542235610052136
                PSNR : 23.102733917663713
                SSIM : 0.6984933018684387

🚀Training Epoch [21/95]: 100%|████████████████████████████████████████████████████████| 1313/1313 [01:40<00:00, 13.02batch/s, loss=0.00424]
📄Testing: 100%|████████████████████████████████████████████████████████| 563/563 [01:01<00:00,  9.09batch/s, loss=0.0034, psnr=25.1, ssim=0.759]

Summary for Epoch 21/95:
    Train    avg_loss: 0.0049289141777017625
    Test     avg_loss: 0.00484285085564595
                PSNR : 23.666361732179727
                SSIM : 0.7109025120735168
```

```
🚀Training Epoch [30/95]: 100%|████████████████████████████████████████████████████████| 1313/1313 [01:55<00:00, 11.38batch/s, loss=0.00417]
📄Testing: 100%|████████████████████████████████████████████████████████| 563/563 [01:02<00:00,  9.07batch/s, loss=0.00308, psnr=25.4, ssim=0.768]

Summary for Epoch 30/95:
    Train    avg_loss: 0.004446224482220747
    Test     avg_loss: 0.004151680975322004
                PSNR : 24.355992915501538
                SSIM : 0.7273935675621033

🚀Training Epoch [31/95]: 100%|████████████████████████████████████████████████████████| 1313/1313 [01:40<00:00, 13.11batch/s, loss=0.00347]
📄Testing: 100%|████████████████████████████████████████████████████████| 563/563 [01:03<00:00,  8.87batch/s, loss=0.00335, psnr=25.3, ssim=0.774]

Summary for Epoch 31/95:
    Train    avg_loss: 0.004483356645759209
    Test     avg_loss: 0.004120911919010628
                PSNR : 24.41093914568297
                SSIM : 0.7221164703369141
```

```
🚀Training Epoch [40/95]: 100%|████████████████████████████████████████████████████████| 1313/1313 [02:20<00:00,  9.33batch/s, loss=0.0106]
📄Testing: 100%|████████████████████████████████████████████████████████| 563/563 [01:15<00:00,  7.45batch/s, loss=0.00399, psnr=25.1, ssim=0.761]

Summary for Epoch 40/95:
    Train    avg_loss: 0.0042134629906866661
    Test     avg_loss: 0.004126837649539801
                PSNR : 24.58817798927084
                SSIM : 0.719488799571991

🚀Training Epoch [41/95]: 100%|████████████████████████████████████████████████████████| 1313/1313 [01:44<00:00, 12.55batch/s, loss=0.00357]
📄Testing: 100%|████████████████████████████████████████████████████████| 563/563 [01:03<00:00,  8.93batch/s, loss=0.00316, psnr=25.6, ssim=0.775]

Summary for Epoch 41/95:
    Train    avg_loss: 0.0041913310812767778
    Test     avg_loss: 0.00387177922180647
                PSNR : 24.801936687980824
                SSIM : 0.7313003540039062
```

```
🚀Training Epoch [49/95]: 100%|████████████████████████████████████████| 1313/1313 [01:39<00:00, 13.16batch/s,
loss=0.00615]
📄Testing: 100%|████████████████████████████████████████| 563/563 [01:01<00:00,  9.09batch/s, loss=0.00382, psnr=25.3,
ssim=0.777]

Summary for Epoch 49/95:
    Train     avg_loss: 0.004078233470574237
    Test      avg_loss: 0.003869095741527992
                PSNR : 24.860818468971786
                SSIM : 0.7334030866622925

🚀Training Epoch [50/95]: 100%|████████████████████████████████████████| 1313/1313 [01:37<00:00, 13.46batch/s,
loss=0.00302]
📄Testing: 100%|████████████████████████████████████████| 563/563 [01:04<00:00,  8.75batch/s, loss=0.00524, psnr=24.1,
ssim=0.764]

Summary for Epoch 50/95:
    Train     avg_loss: 0.004026928504465058
    Test      avg_loss: 0.003884031760736268
                PSNR : 24.744661629621785
                SSIM : 0.7358697056770325
```

(Window Reload เสียก่อนทำให้ code ไม่ run ต่อครับ)

```python
### START CODE HERE ###
input_images = []
for path in image_paths[:2]:  # Process the first 2 images
    image = cv2.imread(path)  # Read the image using cv2 (BGR format)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)  # Convert from BGR to RGB
    image = cv2.resize(image, (128, 128))  # Resize the image to the desired size
    image_tensor = transforms.ToTensor()(image)  # Convert to tensor using torchvision
    input_images.append(image_tensor)

# Stack into a batch of tensors
input_tensor = torch.stack(input_images).to(best_config['device'])

relative_dir = "feature_maps/random_search"
if not os.path.exists(relative_dir):
    os.makedirs(relative_dir)
absolute_dir_result = os.path.abspath(relative_dir)


# Visualize feature maps for all convolution layers
visualizer = FeatureMapVisualizer(model, layers=layers_to_visualize, save_dir=absolute_dir_result)
visualizer.visualize(input_tensor)
### END CODE HERE ###
```

[23] This code loads and preprocesses the first two images from the image_paths list, converting them to PyTorch tensors and stacking them into a batch.

- It sets up a directory for saving the visualized feature maps, ensuring that the results are stored in a specific location.
- The FeatureMapVisualizer is used to generate and save the feature maps of the specified layers in the model, using the preprocessed images as input.
- This process provides insight into how the model processes input data by visually inspecting the feature maps at different layers.