import numpy as np

import IPython.display as display from matplotlib import pyplot as plt import io

import base64

ys = 200 + np.random.randn(100) x = [x for x in range(len(ys))]

fig = plt.figure(figsize=(4, 3), facecolor='w') plt.plot(x, ys, '-')

plt.fill\_between(x, ys, 195, where=(ys > 195), facecolor='g', alpha=0.6)

plt.title("Sample Visualization", fontsize=10)

data = io.BytesIO() plt.savefig(data)

image = F"data:image/png;base64,

{base64.b64encode(data.getvalue()).decode()}" alt = "Sample Visualization"

display.display(display.Markdown(F"""![{alt}]({image})""")) plt.close(fig)

<IPython.core.display.Markdown object>

* [Video Interpolation](https://tensorflow.org/hub/tutorials/tweening_conv3d): Predict what happened in a video between the first and the last frame.

*# Step 1: Install and Import Necessary Libraries*

import torch

import torch.nn as nn import torch.optim as optim

from torch.utils.data import DataLoader import torchvision

import torchvision.transforms as transforms import matplotlib.pyplot as plt

import numpy as np

*# Check if CUDA is available*

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu") print("Using device:", device)

*# Step 2: Prepare the CIFAR-10 Dataset with Data Augmentation*

transform\_train = transforms.Compose([

transforms.RandomHorizontalFlip(), transforms.RandomRotation(10), transforms.ToTensor(),

transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))

])

transform\_test = transforms.Compose([ transforms.ToTensor(),

transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))

])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform\_train)

trainloader = DataLoader(trainset, batch\_size=64, shuffle=True)

testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform\_test)

testloader = DataLoader(testset, batch\_size=64, shuffle=False)

*# Step 3: Define the VAE Model with Regularization Techniques*

class VAE(nn.Module):

def init (self, latent\_dim=256, beta=4): super(VAE, self). init () self.latent\_dim = latent\_dim

self.beta = beta

*# Encoder*

self.enc1 = nn.Conv2d(3, 32, kernel\_size=3, stride=2, padding=1)

self.enc\_bn1 = nn.BatchNorm2d(32)

self.enc2 = nn.Conv2d(32, 64, kernel\_size=3, stride=2, padding=1)

self.enc\_bn2 = nn.BatchNorm2d(64)

self.enc3 = nn.Conv2d(64, 128, kernel\_size=3, stride=2, padding=1)

self.enc\_bn3 = nn.BatchNorm2d(128) self.fc1 = nn.Linear(128 \* 4 \* 4, 1024)

self.fc21 = nn.Linear(1024, latent\_dim) *# Mean of Z*

self.fc22 = nn.Linear(1024, latent\_dim) *# Log-variance of Z*

*# Decoder*

self.fc3 = nn.Linear(latent\_dim, 1024) self.fc4 = nn.Linear(1024, 128 \* 4 \* 4)

self.dec1 = nn.ConvTranspose2d(128, 64, kernel\_size=3, stride=2, padding=1, output\_padding=1)

self.dec\_bn1 = nn.BatchNorm2d(64)

self.dec2 = nn.ConvTranspose2d(64, 32, kernel\_size=3, stride=2, padding=1, output\_padding=1)

self.dec\_bn2 = nn.BatchNorm2d(32)

self.dec3 = nn.ConvTranspose2d(32, 3, kernel\_size=3, stride=2, padding=1, output\_padding=1)

self.dropout = nn.Dropout(p=0.2) *# Dropout with 20% rate*

def encode(self, x):

x = torch.relu(self.enc\_bn1(self.enc1(x))) x = torch.relu(self.enc\_bn2(self.enc2(x))) x = torch.relu(self.enc\_bn3(self.enc3(x))) x = x.view(-1, 128 \* 4 \* 4)

x = torch.relu(self.fc1(x))

return self.fc21(x), self.fc22(x) *# mean and log-variance*

def reparameterize(self, mu, logvar): std = torch.exp(0.5\*logvar)

eps = torch.randn\_like(std) return mu + eps \* std

def decode(self, z):

z = torch.relu(self.fc3(z))

z = torch.relu(self.fc4(z)).view(-1, 128, 4, 4) z = torch.relu(self.dec\_bn1(self.dec1(z)))

z = torch.relu(self.dec\_bn2(self.dec2(z))) return torch.sigmoid(self.dec3(z))

def forward(self, x):

mu, logvar = self.encode(x)

z = self.reparameterize(mu, logvar) recon\_x = self.decode(z)

*# Clamp the output to ensure it's between 0 and 1*

recon\_x = torch.clamp(recon\_x, 0., 1.) return recon\_x, mu, logvar

def loss\_function(self, recon\_x, x, mu, logvar):

BCE = nn.functional.binary\_cross\_entropy(recon\_x.view(-1, 3\*32\*32), x.view(-1, 3\*32\*32), reduction='sum')

*# KL Divergence loss (Modified for Beta-VAE)*

MSE = torch.sum(mu\*\*2 + torch.exp(logvar) - logvar - 1) \* -0.5 return BCE + self.beta \* MSE

*# Step 4: Training the VAE Model with L2 Regularization and Hyperparameter Tuning*

latent\_dim = 256

beta = 4 *# Beta-VAE parameter (adjust as needed)*

lr = 1e-3 epochs = 10

*# Initialize model and optimizer*

model = VAE(latent\_dim=latent\_dim, beta=beta).to(device)

optimizer = optim.Adam(model.parameters(), lr=lr, weight\_decay=1e-5)

*# Weight decay for L2 regularization*

*# Training loop*

for epoch in range(epochs): model.train() train\_loss = 0

for data in trainloader: imgs, \_ = data

imgs = imgs.to(device)

optimizer.zero\_grad()

recon\_imgs, mu, logvar = model(imgs)

loss = model.loss\_function(recon\_imgs, imgs, mu, logvar) loss.backward()

optimizer.step() train\_loss += loss.item()

print(f"Epoch {epoch + 1}/{epochs}, Loss: {train\_loss / len(trainloader.dataset)}")

*# Step 5: Visualize Reconstruction and Generated Samples*

def visualize\_reconstruction(model, num\_images=10): model.eval()

data\_iter = iter(testloader) images, \_ = data\_iter.next()

with torch.no\_grad():

recon\_images, \_, \_ = model(images.to(device))

*# Display original and reconstructed images*

fig, axes = plt.subplots(2, num\_images, figsize=(20, 4)) for i in range(num\_images):

axes[0, i].imshow(images[i].permute(1, 2, 0).cpu().numpy()) axes[0, i].axis('off')

axes[1, i].imshow(recon\_images[i].permute(1, 2, 0).cpu().numpy())

axes[1, i].axis('off') plt.show()

visualize\_reconstruction(model)

Using device: cpu

Files already downloaded and verified Files already downloaded and verified

RuntimeError Traceback (most recent call last)

<ipython-input-4-dcccee48299e> in <cell line: 0>()

120 optimizer.zero\_grad()

121 recon\_imgs, mu, logvar = model(imgs)

--> 122 loss = model.loss\_function(recon\_imgs, imgs, mu, logvar)

123 loss.backward()

124 optimizer.step()

<ipython-input-4-dcccee48299e> in loss\_function(self, recon\_x, x, mu,

logvar)

93

94 def loss\_function(self, recon\_x, x, mu, logvar):

---> 95 BCE =

nn.functional.binary\_cross\_entropy(recon\_x.view(-1, 3\*32\*32), x.view(- 1, 3\*32\*32), reduction='sum')

96

97 # KL Divergence loss (Modified for Beta-VAE)

/usr/local/lib/python3.11/dist-packages/torch/nn/functional.py in binary\_cross\_entropy(input, target, weight, size\_average, reduce, reduction)

3552 weight = weight.expand(new\_size)

3553

-> 3554 return torch.\_C.\_nn.binary\_cross\_entropy(input, target, weight, reduction\_enum)

3555

3556

RuntimeError: all elements of target should be between 0 and 1

*# Step 1: Install and Import Necessary Libraries*

import torch

import torch.nn as nn import torch.optim as optim

from torch.utils.data import DataLoader import torchvision

import torchvision.transforms as transforms import matplotlib.pyplot as plt

import numpy as np

*# Check if CUDA is available*

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu") print("Using device:", device)

*# Step 2: Prepare the CIFAR-10 Dataset with Data Augmentation*

transform\_train = transforms.Compose([

transforms.RandomHorizontalFlip(), transforms.RandomRotation(10), transforms.ToTensor(),

transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))

])

transform\_test = transforms.Compose([ transforms.ToTensor(),

transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))

])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform\_train)

trainloader = DataLoader(trainset, batch\_size=64, shuffle=True)

testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform\_test)

testloader = DataLoader(testset, batch\_size=64, shuffle=False)

*# Step 3: Define the VAE Model with Regularization Techniques*

class VAE(nn.Module):

def init (self, latent\_dim=256, beta=4): super(VAE, self). init () self.latent\_dim = latent\_dim

self.beta = beta

*# Encoder*

self.enc1 = nn.Conv2d(3, 32, kernel\_size=3, stride=2, padding=1)

self.enc\_bn1 = nn.BatchNorm2d(32)

self.enc2 = nn.Conv2d(32, 64, kernel\_size=3, stride=2, padding=1)

self.enc\_bn2 = nn.BatchNorm2d(64)

self.enc3 = nn.Conv2d(64, 128, kernel\_size=3, stride=2, padding=1)

self.enc\_bn3 = nn.BatchNorm2d(128) self.fc1 = nn.Linear(128 \* 4 \* 4, 1024)

self.fc21 = nn.Linear(1024, latent\_dim) *# Mean of Z*

self.fc22 = nn.Linear(1024, latent\_dim) *# Log-variance of Z*

*# Decoder*

self.fc3 = nn.Linear(latent\_dim, 1024) self.fc4 = nn.Linear(1024, 128 \* 4 \* 4)

self.dec1 = nn.ConvTranspose2d(128, 64, kernel\_size=3, stride=2, padding=1, output\_padding=1)

self.dec\_bn1 = nn.BatchNorm2d(64)

self.dec2 = nn.ConvTranspose2d(64, 32, kernel\_size=3, stride=2, padding=1, output\_padding=1)

self.dec\_bn2 = nn.BatchNorm2d(32)

self.dec3 = nn.ConvTranspose2d(32, 3, kernel\_size=3, stride=2, padding=1, output\_padding=1)

self.dropout = nn.Dropout(p=0.2) *# Dropout with 20% rate*

def encode(self, x):

x = torch.relu(self.enc\_bn1(self.enc1(x))) x = torch.relu(self.enc\_bn2(self.enc2(x))) x = torch.relu(self.enc\_bn3(self.enc3(x))) x = x.view(-1, 128 \* 4 \* 4)

x = torch.relu(self.fc1(x))

return self.fc21(x), self.fc22(x) *# mean and log-variance*

def reparameterize(self, mu, logvar): std = torch.exp(0.5\*logvar)

eps = torch.randn\_like(std) return mu + eps \* std

def decode(self, z):

z = torch.relu(self.fc3(z))

z = torch.relu(self.fc4(z)).view(-1, 128, 4, 4) z = torch.relu(self.dec\_bn1(self.dec1(z)))

z = torch.relu(self.dec\_bn2(self.dec2(z))) return torch.sigmoid(self.dec3(z))

def forward(self, x):

mu, logvar = self.encode(x)

z = self.reparameterize(mu, logvar) recon\_x = self.decode(z)

*# Clamp the output to ensure it's between 0 and 1*

recon\_x = torch.clamp(recon\_x, 0., 1.) return recon\_x, mu, logvar

def loss\_function(self, recon\_x, x, mu, logvar):

BCE = nn.functional.binary\_cross\_entropy(recon\_x.view(-1, 3\*32\*32), x.view(-1, 3\*32\*32), reduction='sum')

*# KL Divergence loss (Modified for Beta-VAE)*

MSE = torch.sum(mu\*\*2 + torch.exp(logvar) - logvar - 1) \* -0.5 return BCE + self.beta \* MSE

*# Step 4: Training the VAE Model with L2 Regularization and Hyperparameter Tuning*

latent\_dim = 256

beta = 4 *# Beta-VAE parameter (adjust as needed)*

lr = 1e-3 epochs = 10

*# Initialize model and optimizer*

model = VAE(latent\_dim=latent\_dim, beta=beta).to(device)

optimizer = optim.Adam(model.parameters(), lr=lr, weight\_decay=1e-5)

*# Weight decay for L2 regularization*

*# Training loop*

for epoch in range(epochs): model.train() train\_loss = 0

for data in trainloader: imgs, \_ = data

imgs = imgs.to(device)

optimizer.zero\_grad()

recon\_imgs, mu, logvar = model(imgs)

loss = model.loss\_function(recon\_imgs, imgs, mu, logvar) loss.backward()

optimizer.step() train\_loss += loss.item()

print(f"Epoch {epoch + 1}/{epochs}, Loss: {train\_loss / len(trainloader.dataset)}")

*# Step 5: Visualize Reconstruction and Generated Samples*

def visualize\_reconstruction(model, num\_images=10): model.eval()

data\_iter = iter(testloader) images, \_ = data\_iter.next()

with torch.no\_grad():

recon\_images, \_, \_ = model(images.to(device))

*# Display original and reconstructed images*

fig, axes = plt.subplots(2, num\_images, figsize=(20, 4)) for i in range(num\_images):

axes[0, i].imshow(images[i].permute(1, 2, 0).cpu().numpy()) axes[0, i].axis('off')

axes[1, i].imshow(recon\_images[i].permute(1, 2, 0).cpu().numpy())

axes[1, i].axis('off') plt.show()

visualize\_reconstruction(model)

Using device: cpu

Files already downloaded and verified Files already downloaded and verified

RuntimeError Traceback (most recent call last)

<ipython-input-5-dcccee48299e> in <cell line: 0>()

120 optimizer.zero\_grad()

121 recon\_imgs, mu, logvar = model(imgs)

--> 122 loss = model.loss\_function(recon\_imgs, imgs, mu, logvar)

123 loss.backward()

124 optimizer.step()

<ipython-input-5-dcccee48299e> in loss\_function(self, recon\_x, x, mu,

logvar)

93

94 def loss\_function(self, recon\_x, x, mu, logvar):

---> 95 BCE =

nn.functional.binary\_cross\_entropy(recon\_x.view(-1, 3\*32\*32), x.view(- 1, 3\*32\*32), reduction='sum')

96

97 # KL Divergence loss (Modified for Beta-VAE)

/usr/local/lib/python3.11/dist-packages/torch/nn/functional.py in binary\_cross\_entropy(input, target, weight, size\_average, reduce, reduction)

3552 weight = weight.expand(new\_size)

3553

-> 3554 return torch.\_C.\_nn.binary\_cross\_entropy(input, target, weight, reduction\_enum)

3555

3556

RuntimeError: all elements of target should be between 0 and 1

*# Step 1: Install and Import Necessary Libraries*

import torch

import torch.nn as nn import torch.optim as optim

from torch.utils.data import DataLoader import torchvision

import torchvision.transforms as transforms import matplotlib.pyplot as plt

import numpy as np

*# Check if CUDA is available*

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu") print("Using device:", device)

*# Step 2: Prepare the CIFAR-10 Dataset with Data Augmentation*

transform\_train = transforms.Compose([

transforms.RandomHorizontalFlip(), transforms.RandomRotation(10), transforms.ToTensor(),

transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)) *#*

*Normalize to [-1, 1]*

])

transform\_test = transforms.Compose([ transforms.ToTensor(),

transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)) *#*

*Normalize to [-1, 1]*

])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform\_train)

trainloader = DataLoader(trainset, batch\_size=64, shuffle=True)

testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform\_test)

testloader = DataLoader(testset, batch\_size=64, shuffle=False)

*# Step 3: Define the VAE Model with Regularization Techniques*

class VAE(nn.Module):

def init (self, latent\_dim=256, beta=4): super(VAE, self). init () self.latent\_dim = latent\_dim

self.beta = beta

*# Encoder*

self.enc1 = nn.Conv2d(3, 32, kernel\_size=3, stride=2, padding=1)

self.enc\_bn1 = nn.BatchNorm2d(32)

self.enc2 = nn.Conv2d(32, 64, kernel\_size=3, stride=2, padding=1)

self.enc\_bn2 = nn.BatchNorm2d(64)

self.enc3 = nn.Conv2d(64, 128, kernel\_size=3, stride=2, padding=1)

self.enc\_bn3 = nn.BatchNorm2d(128) self.fc1 = nn.Linear(128 \* 4 \* 4, 1024)

self.fc21 = nn.Linear(1024, latent\_dim) *# Mean of Z*

self.fc22 = nn.Linear(1024, latent\_dim) *# Log-variance of Z*

*# Decoder*

self.fc3 = nn.Linear(latent\_dim, 1024) self.fc4 = nn.Linear(1024, 128 \* 4 \* 4)

self.dec1 = nn.ConvTranspose2d(128, 64, kernel\_size=3, stride=2, padding=1, output\_padding=1)

self.dec\_bn1 = nn.BatchNorm2d(64)

self.dec2 = nn.ConvTranspose2d(64, 32, kernel\_size=3,

stride=2, padding=1, output\_padding=1) self.dec\_bn2 = nn.BatchNorm2d(32)

self.dec3 = nn.ConvTranspose2d(32, 3, kernel\_size=3, stride=2, padding=1, output\_padding=1)

self.dropout = nn.Dropout(p=0.2) *# Dropout with 20% rate*

def encode(self, x):

x = torch.relu(self.enc\_bn1(self.enc1(x))) x = torch.relu(self.enc\_bn2(self.enc2(x))) x = torch.relu(self.enc\_bn3(self.enc3(x))) x = x.view(-1, 128 \* 4 \* 4)

x = torch.relu(self.fc1(x))

return self.fc21(x), self.fc22(x) *# mean and log-variance*

def reparameterize(self, mu, logvar): std = torch.exp(0.5\*logvar)

eps = torch.randn\_like(std) return mu + eps \* std

def decode(self, z):

z = torch.relu(self.fc3(z))

z = torch.relu(self.fc4(z)).view(-1, 128, 4, 4) z = torch.relu(self.dec\_bn1(self.dec1(z)))

z = torch.relu(self.dec\_bn2(self.dec2(z))) return torch.sigmoid(self.dec3(z))

def forward(self, x):

mu, logvar = self.encode(x)

z = self.reparameterize(mu, logvar) recon\_x = self.decode(z)

*# Clamp the output to ensure it's between 0 and 1*

recon\_x = torch.clamp(recon\_x, 0., 1.) return recon\_x, mu, logvar

def loss\_function(self, recon\_x, x, mu, logvar):

*# Binary Cross-Entropy (BCE) loss*

BCE = nn.functional.binary\_cross\_entropy(recon\_x.view(-1, 3\*32\*32), x.view(-1, 3\*32\*32), reduction='sum')

*# KL Divergence loss (Modified for Beta-VAE)*

MSE = torch.sum(mu\*\*2 + torch.exp(logvar) - logvar - 1) \* -0.5 return BCE + self.beta \* MSE

*# Step 4: Training the VAE Model with L2 Regularization and Hyperparameter Tuning*

latent\_dim = 256

beta = 4 *# Beta-VAE parameter (adjust as needed)*

lr = 1e-3 epochs = 10

*# Initialize model and optimizer*

model = VAE(latent\_dim=latent\_dim, beta=beta).to(device)

optimizer = optim.Adam(model.parameters(), lr=lr, weight\_decay=1e-5)

*# Weight decay for L2 regularization*

*# Training loop*

for epoch in range(epochs): model.train() train\_loss = 0

for data in trainloader: imgs, \_ = data

imgs = imgs.to(device)

optimizer.zero\_grad()

recon\_imgs, mu, logvar = model(imgs)

loss = model.loss\_function(recon\_imgs, imgs, mu, logvar) loss.backward()

optimizer.step() train\_loss += loss.item()

print(f"Epoch {epoch + 1}/{epochs}, Loss: {train\_loss / len(trainloader.dataset)}")

*# Step 5: Visualize Reconstruction and Generated Samples*

def visualize\_reconstruction(model, num\_images=10): model.eval()

data\_iter = iter(testloader) images, \_ = data\_iter.next()

with torch.no\_grad():

recon\_images, \_, \_ = model(images.to(device))

*# Display original and reconstructed images*

fig, axes = plt.subplots(2, num\_images, figsize=(20, 4)) for i in range(num\_images):

axes[0, i].imshow(images[i].permute(1, 2, 0).cpu().numpy()) axes[0, i].axis('off')

axes[1, i].imshow(recon\_images[i].permute(1, 2, 0).cpu().numpy())

axes[1, i].axis('off') plt.show()

visualize\_reconstruction(model)

Using device: cpu

Files already downloaded and verified Files already downloaded and verified

RuntimeError Traceback (most recent call last)

<ipython-input-6-3ad01ca47c90> in <cell line: 0>()

121 optimizer.zero\_grad()

122 recon\_imgs, mu, logvar = model(imgs)

--> 123 loss = model.loss\_function(recon\_imgs, imgs, mu, logvar)

124 loss.backward()

125 optimizer.step()

<ipython-input-6-3ad01ca47c90> in loss\_function(self, recon\_x, x, mu, logvar)

94 def loss\_function(self, recon\_x, x, mu, logvar):

95 # Binary Cross-Entropy (BCE) loss

---> 96 BCE =

nn.functional.binary\_cross\_entropy(recon\_x.view(-1, 3\*32\*32), x.view(- 1, 3\*32\*32), reduction='sum')

97

98 # KL Divergence loss (Modified for Beta-VAE)

/usr/local/lib/python3.11/dist-packages/torch/nn/functional.py in binary\_cross\_entropy(input, target, weight, size\_average, reduce, reduction)

3552 weight = weight.expand(new\_size)

3553

-> 3554 return torch.\_C.\_nn.binary\_cross\_entropy(input, target, weight, reduction\_enum)

3555

3556

RuntimeError: all elements of target should be between 0 and 1

*# Step 1: Install and Import Necessary Libraries*

import torch

import torch.nn as nn import torch.optim as optim

from torch.utils.data import DataLoader import torchvision

import torchvision.transforms as transforms import matplotlib.pyplot as plt

import numpy as np

*# Check if CUDA is available*

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

print("Using device:", device)

*# Step 2: Prepare the CIFAR-10 Dataset with Data Augmentation*

transform\_train = transforms.Compose([ transforms.RandomHorizontalFlip(), transforms.RandomRotation(10), transforms.ToTensor(),

transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)) *#*

*Normalize to [0, 1]*

])

transform\_test = transforms.Compose([ transforms.ToTensor(),

transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)) *#*

*Normalize to [0, 1]*

])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform\_train)

trainloader = DataLoader(trainset, batch\_size=64, shuffle=True)

testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform\_test)

testloader = DataLoader(testset, batch\_size=64, shuffle=False)

*# Step 3: Define the VAE Model with Regularization Techniques*

class VAE(nn.Module):

def init (self, latent\_dim=256, beta=4): super(VAE, self). init () self.latent\_dim = latent\_dim

self.beta = beta

*# Encoder*

self.enc1 = nn.Conv2d(3, 32, kernel\_size=3, stride=2, padding=1)

self.enc\_bn1 = nn.BatchNorm2d(32)

self.enc2 = nn.Conv2d(32, 64, kernel\_size=3, stride=2, padding=1)

self.enc\_bn2 = nn.BatchNorm2d(64)

self.enc3 = nn.Conv2d(64, 128, kernel\_size=3, stride=2, padding=1)

self.enc\_bn3 = nn.BatchNorm2d(128) self.fc1 = nn.Linear(128 \* 4 \* 4, 1024)

self.fc21 = nn.Linear(1024, latent\_dim) *# Mean of Z*

self.fc22 = nn.Linear(1024, latent\_dim) *# Log-variance of Z*

*# Decoder*

self.fc3 = nn.Linear(latent\_dim, 1024) self.fc4 = nn.Linear(1024, 128 \* 4 \* 4)

self.dec1 = nn.ConvTranspose2d(128, 64, kernel\_size=3, stride=2, padding=1, output\_padding=1)

self.dec\_bn1 = nn.BatchNorm2d(64)

self.dec2 = nn.ConvTranspose2d(64, 32, kernel\_size=3, stride=2, padding=1, output\_padding=1)

self.dec\_bn2 = nn.BatchNorm2d(32)

self.dec3 = nn.ConvTranspose2d(32, 3, kernel\_size=3, stride=2, padding=1, output\_padding=1)

self.dropout = nn.Dropout(p=0.2) *# Dropout with 20% rate*

def encode(self, x):

x = torch.relu(self.enc\_bn1(self.enc1(x))) x = torch.relu(self.enc\_bn2(self.enc2(x))) x = torch.relu(self.enc\_bn3(self.enc3(x))) x = x.view(-1, 128 \* 4 \* 4)

x = torch.relu(self.fc1(x))

return self.fc21(x), self.fc22(x) *# mean and log-variance*

def reparameterize(self, mu, logvar): std = torch.exp(0.5\*logvar)

eps = torch.randn\_like(std) return mu + eps \* std

def decode(self, z):

z = torch.relu(self.fc3(z))

z = torch.relu(self.fc4(z)).view(-1, 128, 4, 4) z = torch.relu(self.dec\_bn1(self.dec1(z)))

z = torch.relu(self.dec\_bn2(self.dec2(z))) return torch.sigmoid(self.dec3(z))

def forward(self, x):

mu, logvar = self.encode(x)

z = self.reparameterize(mu, logvar) recon\_x = self.decode(z)

*# Clamp the output to ensure it's between 0 and 1*

recon\_x = torch.clamp(recon\_x, 0., 1.) return recon\_x, mu, logvar

def loss\_function(self, recon\_x, x, mu, logvar):

*# Binary Cross-Entropy (BCE) loss*

BCE = nn.functional.binary\_cross\_entropy(recon\_x.view(-1, 3\*32\*32), x.view(-1, 3\*32\*32), reduction='sum')

*# KL Divergence loss (Modified for Beta-VAE)*

MSE = torch.sum(mu\*\*2 + torch.exp(logvar) - logvar - 1) \* -0.5 return BCE + self.beta \* MSE

*# Step 4: Training the VAE Model with L2 Regularization and Hyperparameter Tuning*

latent\_dim = 256

beta = 4 *# Beta-VAE parameter (adjust as needed)*

lr = 1e-3 epochs = 10

*# Initialize model and optimizer*

model = VAE(latent\_dim=latent\_dim, beta=beta).to(device)

optimizer = optim.Adam(model.parameters(), lr=lr, weight\_decay=1e-5)

*# Weight decay for L2 regularization*

*# Training loop*

for epoch in range(epochs): model.train() train\_loss = 0

for data in trainloader: imgs, \_ = data

imgs = imgs.to(device)

optimizer.zero\_grad()

recon\_imgs, mu, logvar = model(imgs)

loss = model.loss\_function(recon\_imgs, imgs, mu, logvar) loss.backward()

optimizer.step() train\_loss += loss.item()

print(f"Epoch {epoch + 1}/{epochs}, Loss: {train\_loss / len(trainloader.dataset)}")

*# Step 5: Visualize Reconstruction and Generated Samples*

def visualize\_reconstruction(model, num\_images=10): model.eval()

data\_iter = iter(testloader) images, \_ = data\_iter.next()

with torch.no\_grad():

recon\_images, \_, \_ = model(images.to(device))

*# Display original and reconstructed images*

fig, axes = plt.subplots(2, num\_images, figsize=(20, 4)) for i in range(num\_images):

axes[0, i].imshow(images[i].permute(1, 2, 0).cpu().numpy()) axes[0, i].axis('off')

axes[1, i].imshow(recon\_images[i].permute(1, 2, 0).cpu().numpy())

axes[1, i].axis('off') plt.show()

visualize\_reconstruction(model)

Using device: cpu

Files already downloaded and verified Files already downloaded and verified

RuntimeError Traceback (most recent call last)

<ipython-input-8-917ad4af5b70> in <cell line: 0>()

121 optimizer.zero\_grad()

122 recon\_imgs, mu, logvar = model(imgs)

--> 123 loss = model.loss\_function(recon\_imgs, imgs, mu, logvar)

124 loss.backward()

125 optimizer.step()

<ipython-input-8-917ad4af5b70> in loss\_function(self, recon\_x, x, mu, logvar)

94 def loss\_function(self, recon\_x, x, mu, logvar):

95 # Binary Cross-Entropy (BCE) loss

---> 96 BCE =

nn.functional.binary\_cross\_entropy(recon\_x.view(-1, 3\*32\*32), x.view(- 1, 3\*32\*32), reduction='sum')

97

98 # KL Divergence loss (Modified for Beta-VAE)

/usr/local/lib/python3.11/dist-packages/torch/nn/functional.py in binary\_cross\_entropy(input, target, weight, size\_average, reduce, reduction)

3552 weight = weight.expand(new\_size)

3553

-> 3554 return torch.\_C.\_nn.binary\_cross\_entropy(input, target, weight, reduction\_enum)

3555

3556

RuntimeError: all elements of target should be between 0 and 1

*# Step 1: Install and Import Necessary Libraries*

import torch

import torch.nn as nn import torch.optim as optim

from torch.utils.data import DataLoader import torchvision

import torchvision.transforms as transforms import matplotlib.pyplot as plt

import numpy as np

*# Check if CUDA is available*

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu") print("Using device:", device)

*# Step 2: Prepare the CIFAR-10 Dataset with Data Augmentation*

transform\_train = transforms.Compose([ transforms.RandomHorizontalFlip(), transforms.RandomRotation(10), transforms.ToTensor(),

transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)) *#*

*Normalize to [0, 1]*

])

transform\_test = transforms.Compose([ transforms.ToTensor(),

transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)) *#*

*Normalize to [0, 1]*

])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform\_train)

trainloader = DataLoader(trainset, batch\_size=64, shuffle=True)

testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform\_test)

testloader = DataLoader(testset, batch\_size=64, shuffle=False)

*# Step 3: Define the VAE Model with Regularization Techniques*

class VAE(nn.Module):

def init (self, latent\_dim=256, beta=4): super(VAE, self). init () self.latent\_dim = latent\_dim

self.beta = beta

*# Encoder*

self.enc1 = nn.Conv2d(3, 32, kernel\_size=3, stride=2, padding=1)

self.enc\_bn1 = nn.BatchNorm2d(32)

self.enc2 = nn.Conv2d(32, 64, kernel\_size=3, stride=2, padding=1)

self.enc\_bn2 = nn.BatchNorm2d(64)

self.enc3 = nn.Conv2d(64, 128, kernel\_size=3, stride=2, padding=1)

self.enc\_bn3 = nn.BatchNorm2d(128) self.fc1 = nn.Linear(128 \* 4 \* 4, 1024)

self.fc21 = nn.Linear(1024, latent\_dim) *# Mean of Z*

self.fc22 = nn.Linear(1024, latent\_dim) *# Log-variance of Z*

*# Decoder*

self.fc3 = nn.Linear(latent\_dim, 1024) self.fc4 = nn.Linear(1024, 128 \* 4 \* 4)

self.dec1 = nn.ConvTranspose2d(128, 64, kernel\_size=3, stride=2, padding=1, output\_padding=1)

self.dec\_bn1 = nn.BatchNorm2d(64)

self.dec2 = nn.ConvTranspose2d(64, 32, kernel\_size=3, stride=2, padding=1, output\_padding=1)

self.dec\_bn2 = nn.BatchNorm2d(32)

self.dec3 = nn.ConvTranspose2d(32, 3, kernel\_size=3, stride=2, padding=1, output\_padding=1)

self.dropout = nn.Dropout(p=0.2) *# Dropout with 20% rate*

def encode(self, x):

x = torch.relu(self.enc\_bn1(self.enc1(x))) x = torch.relu(self.enc\_bn2(self.enc2(x))) x = torch.relu(self.enc\_bn3(self.enc3(x))) x = x.view(-1, 128 \* 4 \* 4)

x = torch.relu(self.fc1(x))

return self.fc21(x), self.fc22(x) *# mean and log-variance*

def reparameterize(self, mu, logvar): std = torch.exp(0.5\*logvar)

eps = torch.randn\_like(std) return mu + eps \* std

def decode(self, z):

z = torch.relu(self.fc3(z))

z = torch.relu(self.fc4(z)).view(-1, 128, 4, 4) z = torch.relu(self.dec\_bn1(self.dec1(z)))

z = torch.relu(self.dec\_bn2(self.dec2(z))) return torch.sigmoid(self.dec3(z))

def forward(self, x):

mu, logvar = self.encode(x)

z = self.reparameterize(mu, logvar) recon\_x = self.decode(z)

*# Clamp the output to ensure it's between 0 and 1*

recon\_x = torch.clamp(recon\_x, 0., 1.) return recon\_x, mu, logvar

def loss\_function(self, recon\_x, x, mu, logvar):

*# Binary Cross-Entropy (BCE) loss with logits, directly using sigmoid inside BCEWithLogitsLoss*

BCE =

nn.functional.binary\_cross\_entropy\_with\_logits(recon\_x.view(-1, 3\*32\*32), x.view(-1, 3\*32\*32), reduction='sum')

*# KL Divergence loss (Modified for Beta-VAE)*

MSE = torch.sum(mu\*\*2 + torch.exp(logvar) - logvar - 1) \* -0.5 return BCE + self.beta \* MSE

*# Step 4: Training the VAE Model with L2 Regularization and Hyperparameter Tuning*

latent\_dim = 256

beta = 4 *# Beta-VAE parameter (adjust as needed)*

lr = 1e-3 epochs = 10

*# Initialize model and optimizer*

model = VAE(latent\_dim=latent\_dim, beta=beta).to(device)

optimizer = optim.Adam(model.parameters(), lr=lr, weight\_decay=1e-5)

*# Weight decay for L2 regularization*

*# Training loop*

for epoch in range(epochs): model.train() train\_loss = 0

for data in trainloader: imgs, \_ = data

imgs = imgs.to(device)

optimizer.zero\_grad()

recon\_imgs, mu, logvar = model(imgs)

loss = model.loss\_function(recon\_imgs, imgs, mu, logvar) loss.backward()

optimizer.step() train\_loss += loss.item()

print(f"Epoch {epoch + 1}/{epochs}, Loss: {train\_loss / len(trainloader.dataset)}")

*# Step 5: Visualize Reconstruction and Generated Samples*

def visualize\_reconstruction(model, num\_images=10): model.eval()

data\_iter = iter(testloader) images, \_ = data\_iter.next()

with torch.no\_grad():

recon\_images, \_, \_ = model(images.to(device))

*# Display original and reconstructed images*

fig, axes = plt.subplots(2, num\_images, figsize=(20, 4)) for i in range(num\_images):

axes[0, i].imshow(images[i].permute(1, 2, 0).cpu().numpy())

axes[0, i].axis('off')

134

--> 135

data\_iter = iter(testloader) images, \_ = data\_iter.next()

axes[1, i].imshow(recon\_images[i].permute(1, 2, 0).cpu().numpy())

axes[1, i].axis('off') plt.show()

visualize\_reconstruction(model)

Using device: cpu

Files already downloaded and verified Files already downloaded and verified Epoch 1/10, Loss: nan

Epoch 2/10, Loss: nan Epoch 3/10, Loss: nan Epoch 4/10, Loss: nan Epoch 5/10, Loss: nan Epoch 6/10, Loss: nan Epoch 7/10, Loss: nan Epoch 8/10, Loss: nan Epoch 9/10, Loss: nan Epoch 10/10, Loss: nan

AttributeError Traceback (most recent call last)

<ipython-input-9-4116f7968b8e> in <cell line: 0>()

147 plt.show()

148

--> 149 visualize\_reconstruction(model)

<ipython-input-9-4116f7968b8e> in visualize\_reconstruction(model, num\_images)

133 model.eval()

136

137 with torch.no\_grad():

AttributeError: '\_SingleProcessDataLoaderIter' object has no attribute 'next'

*# Step 1: Install and Import Necessary Libraries*

import torch

import torch.nn as nn import torch.optim as optim

from torch.utils.data import DataLoader import torchvision

import torchvision.transforms as transforms import matplotlib.pyplot as plt

import numpy as np

*# Check if CUDA is available*

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu") print("Using device:", device)

*# Step 2: Prepare the CIFAR-10 Dataset with Data Augmentation*

transform\_train = transforms.Compose([ transforms.RandomHorizontalFlip(), transforms.RandomRotation(10), transforms.ToTensor(),

transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)) *#*

*Normalize to [0, 1]*

])

transform\_test = transforms.Compose([ transforms.ToTensor(),

transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)) *#*

*Normalize to [0, 1]*

])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform\_train)

trainloader = DataLoader(trainset, batch\_size=64, shuffle=True)

testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform\_test)

testloader = DataLoader(testset, batch\_size=64, shuffle=False)

*# Step 3: Define the VAE Model with Regularization Techniques*

class VAE(nn.Module):

def init (self, latent\_dim=256, beta=4): super(VAE, self). init () self.latent\_dim = latent\_dim

self.beta = beta

*# Encoder*

self.enc1 = nn.Conv2d(3, 32, kernel\_size=3, stride=2, padding=1)

self.enc\_bn1 = nn.BatchNorm2d(32)

self.enc2 = nn.Conv2d(32, 64, kernel\_size=3, stride=2, padding=1)

self.enc\_bn2 = nn.BatchNorm2d(64)

self.enc3 = nn.Conv2d(64, 128, kernel\_size=3, stride=2, padding=1)

self.enc\_bn3 = nn.BatchNorm2d(128) self.fc1 = nn.Linear(128 \* 4 \* 4, 1024)

self.fc21 = nn.Linear(1024, latent\_dim) *# Mean of Z*

self.fc22 = nn.Linear(1024, latent\_dim) *# Log-variance of Z*

*# Decoder*

self.fc3 = nn.Linear(latent\_dim, 1024) self.fc4 = nn.Linear(1024, 128 \* 4 \* 4)

self.dec1 = nn.ConvTranspose2d(128, 64, kernel\_size=3, stride=2, padding=1, output\_padding=1)

self.dec\_bn1 = nn.BatchNorm2d(64)

self.dec2 = nn.ConvTranspose2d(64, 32, kernel\_size=3, stride=2, padding=1, output\_padding=1)

self.dec\_bn2 = nn.BatchNorm2d(32)

self.dec3 = nn.ConvTranspose2d(32, 3, kernel\_size=3, stride=2, padding=1, output\_padding=1)

self.dropout = nn.Dropout(p=0.2) *# Dropout with 20% rate*

def encode(self, x):

x = torch.relu(self.enc\_bn1(self.enc1(x))) x = torch.relu(self.enc\_bn2(self.enc2(x))) x = torch.relu(self.enc\_bn3(self.enc3(x))) x = x.view(-1, 128 \* 4 \* 4)

x = torch.relu(self.fc1(x))

return self.fc21(x), self.fc22(x) *# mean and log-variance*

def reparameterize(self, mu, logvar): std = torch.exp(0.5\*logvar)

eps = torch.randn\_like(std) return mu + eps \* std

def decode(self, z):

z = torch.relu(self.fc3(z))

z = torch.relu(self.fc4(z)).view(-1, 128, 4, 4) z = torch.relu(self.dec\_bn1(self.dec1(z)))

z = torch.relu(self.dec\_bn2(self.dec2(z))) return torch.sigmoid(self.dec3(z))

def forward(self, x):

mu, logvar = self.encode(x)

z = self.reparameterize(mu, logvar) recon\_x = self.decode(z)

*# Clamp the output to ensure it's between 0 and 1*

recon\_x = torch.clamp(recon\_x, 0., 1.) return recon\_x, mu, logvar

def loss\_function(self, recon\_x, x, mu, logvar):

*# Binary Cross-Entropy (BCE) loss with logits, directly using sigmoid inside BCEWithLogitsLoss*

BCE =

nn.functional.binary\_cross\_entropy\_with\_logits(recon\_x.view(-1,

3\*32\*32), x.view(-1, 3\*32\*32), reduction='sum')

*# KL Divergence loss (Modified for Beta-VAE)*

MSE = torch.sum(mu\*\*2 + torch.exp(logvar) - logvar - 1) \* -0.5 return BCE + self.beta \* MSE

*# Step 4: Training the VAE Model with L2 Regularization and Hyperparameter Tuning*

latent\_dim = 256

beta = 4 *# Beta-VAE parameter (adjust as needed)*

lr = 1e-3 epochs = 10

*# Initialize model and optimizer*

model = VAE(latent\_dim=latent\_dim, beta=beta).to(device)

optimizer = optim.Adam(model.parameters(), lr=lr, weight\_decay=1e-5)

*# Weight decay for L2 regularization*

*# Training loop*

for epoch in range(epochs): model.train() train\_loss = 0

for data in trainloader: imgs, \_ = data

imgs = imgs.to(device)

optimizer.zero\_grad()

recon\_imgs, mu, logvar = model(imgs)

loss = model.loss\_function(recon\_imgs, imgs, mu, logvar) loss.backward()

optimizer.step() train\_loss += loss.item()

print(f"Epoch {epoch + 1}/{epochs}, Loss: {train\_loss / len(trainloader.dataset)}")

*# Step 5: Visualize Reconstruction and Generated Samples*

def visualize\_reconstruction(model, num\_images=10): model.eval()

data\_iter = iter(testloader) *# Create an iterator from the test loader*

images, \_ = next(data\_iter) *# Fetch the next batch of images*

with torch.no\_grad():

recon\_images, \_, \_ = model(images.to(device))

*# Display original and reconstructed images*

fig, axes = plt.subplots(2, num\_images, figsize=(20, 4))

for i in range(num\_images):

axes[0, i].imshow(images[i].permute(1, 2, 0).cpu().numpy()) axes[0, i].axis('off')

axes[1, i].imshow(recon\_images[i].permute(1, 2, 0).cpu().numpy())

axes[1, i].axis('off') plt.show()

visualize\_reconstruction(model)

Using device: cuda

Downloading [https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to](http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gzto)

./data/cifar-10-python.tar.gz

100%|██████████| 170M/170M [00:13<00:00, 12.3MB/s]

Extracting ./data/cifar-10-python.tar.gz to ./data Files already downloaded and verified

Epoch 1/10, Loss: nan Epoch 2/10, Loss: nan Epoch 3/10, Loss: nan Epoch 4/10, Loss: nan Epoch 5/10, Loss: nan Epoch 6/10, Loss: nan Epoch 7/10, Loss: nan Epoch 8/10, Loss: nan Epoch 9/10, Loss: nan

WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers). Got range [-0.8980392..1.0].

WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers). Got range [-1.0..0.9372549].

WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers). Got range [-0.9764706..0.9764706].

WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers). Got range [-0.7411765..0.8352941].

WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers). Got range [-0.9529412..0.7254902].

WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers). Got range [-1.0..0.654902].

WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers). Got range [-1.0..1.0].

WARNING:matplotlib.image:Clipping input data to the valid range for

imshow with RGB data ([0..1] for floats or [0..255] for integers). Got range [-0.92941177..1.0].

WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers). Got range [-0.8745098..0.96862745].

WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers). Got range [-0.84313726..0.96862745].

Epoch 10/10, Loss: nan

