Code for Adaptive Augmentations in Latent Space using Autoencoders

Nikhila Ramisetti

May 1, 2024

Introduction

This project introduces a novel approach to enhancing image generation by integrating adaptive augmentations within the latent space of Autoencoders (AEs). Traditional augmentation techniques, while useful, often struggle to meet the complex demands of image generation tasks, especially when faced with limited and homogeneous datasets. By embedding adaptive augmentations directly into the latent space of AEs, this project aims to overcome the limitations of conventional augmentation methods and improve model performance and scalability.

Installation and Usage Instructions

This section outlines installation instructions, dependencies, directory structure, and other essential subsections of the projects main notebook.

To use this project, follow these steps to set up your environment:

1. Clone the Repository: Clone this repository to your local machine.

```
git clone https://github.com/NikhilaRamisetti/Adaptive-Augmentations-in-Latent-Space-using-Autoencoders
```

2. Create Virtual Environment: Navigate to the project directory and create a virtual environment.

```
cd <project-directory>
python3 -m venv venv
```

3. Activate Virtual Environment

```
source venv/bin/activate
```

4. Install Dependencies

```
pip install -r requirements.txt
```

5. Code Directory Structure

   - README.md - The top-level README for developers.
   - data
     - input - Folder containing celebA raw data.
     - output - Folder to save .ckpts, generated images and plots.
       - checkpoints - Weights of model saved as .h5 files
       - generated_images - Folder containing generated images.
       - figures - Folder containing plots of training progress.
   - docs - A folder for documentation
   - notebooks
     - main_code.ipynb - The main code(containing the model).
     - evaluation.ipynb - The evaluation of the model.
   - requirements.txt - Required modules to be installed.
   - references - Data dictionaries, manuals, and all other explanatory materials.

Code listings

This section provides insights into the main code source file main_code.ipynb, which consists of the project's goals, methodologies, and expected outcomes.

In what follows, we list all the subsections distributed with this project's main code, providing an index for easy navigation:

Code Listings

Code Listing 1: Libraries and Modules

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import tensorflow as tf
import pickle
import os
import cv2
import time
import glob
#import tensorflow_probability as tfp

from sklearn.metrics import accuracy_score, precision_score, recall_score
from sklearn.model_selection import train_test_split
from tensorflow.keras import layers, Model, losses
from tensorflow.keras.layers import Layer, Input, Conv2D, Dense, Flatten, Reshape
    , Lambda, Dropout
from tensorflow.keras.layers import Conv2DTranspose, MaxPooling2D, UpSampling2D,
   LeakyReLU, BatchNormalization
from tensorflow.keras.activations import relu
from tensorflow.keras.models import Sequential, load_model
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping
from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

Code Listing 2: Data Loading and Preprocessing

```
# Load the Dataset
# Define the batch size
batch_size = 32
# Define the directory containing the CelebA dataset
celebA_dir = '../data/input/img_align_celeba'
num_files = 70000 # Define the desired number of files


# List all JPEG files in the directory
image_filenames = glob.glob(os.path.join(celebA_dir, '*.jpg'))

# Take only the first 'num_files' files
image_filenames = image_filenames[:num_files]

# Check if there are any JPEG files found
if not image_filenames:
    raise ValueError("No JPEG files found in the directory.")
```

```
# Split the dataset into training and testing sets
train_files, test_files = train_test_split(image_filenames, test_size=0.2,
    random_state=42)

# Print the number of images in each split
print("Number of train images:", len(train_files))
print("Number of test images:", len(test_files))


# Preprocess the dataset
def preprocess_images(image_filenames, target_size=(128, 128)):
    # Load, resize, and convert images to RGB
    images = [cv2.cvtColor(cv2.resize(cv2.imread(filename), target_size), cv2.
        COLOR_BGR2RGB) for filename in image_filenames if cv2.imread(filename) is
        not None]
    # Convert images to float32 and normalize
    preprocessed_images = np.array(images).astype('float32') / 255.0
    return preprocessed_images

target_size = (128,128)
# Load and preprocess test images
test_images = preprocess_images(test_files, target_size)

print(f"test_images:{test_images.shape}")
```

Code Listing 3: VAE Model and Loss Functions

```
# VAE Model
# This model consists of Encoder and Decoder as two main components
class GaussianSampling(Layer):
    def call(self, inputs):
        means, logvar = inputs
        epsilon = tf.random.normal(shape=tf.shape(means), mean=0., stddev=1.)
        samples = means + tf.exp(0.5*logvar)*epsilon

        return samples

class DownConvBlock(Layer):
    count = 0
    def __init__(self, filters, kernel_size=(3,3), strides=1, padding='same'):
        super(DownConvBlock, self).__init__(name=f"DownConvBlock_{DownConvBlock.
            count}")
        DownConvBlock.count+=1
        self.forward = Sequential([Conv2D(filters, kernel_size, strides, padding)
            ])
        self.forward.add(BatchNormalization())
        self.forward.add(layers.LeakyReLU(0.2))

    def call(self, inputs):
        return self.forward(inputs)

class UpConvBlock(Layer):
    count = 0
    def __init__(self, filters, kernel_size=(3,3), padding='same'):
        super(UpConvBlock, self).__init__(name=f"UpConvBlock_{UpConvBlock.count}"
            )
        UpConvBlock.count += 1
        self.forward = Sequential([Conv2D(filters, kernel_size, 1, padding),])
        self.forward.add(layers.LeakyReLU(0.2))
        self.forward.add(UpSampling2D((2,2)))
```

```
    def call(self, inputs):
        return self.forward(inputs)

class Encoder(Layer):
    def __init__(self, z_dim, name='encoder'):
        super(Encoder, self).__init__(name=name)

        self.features_extract = Sequential([
            DownConvBlock(filters = 32, kernel_size=(3,3), strides=2),
            DownConvBlock(filters = 32, kernel_size=(3,3), strides=2),
            DownConvBlock(filters = 64, kernel_size=(3,3), strides=2),
            DownConvBlock(filters = 64, kernel_size=(3,3), strides=2),
            Flatten()])

        self.dense_mean = Dense(z_dim, name='mean')
        self.dense_logvar = Dense(z_dim, name='logvar')
        self.sampler = GaussianSampling()

    def call(self, inputs):
        x = self.features_extract(inputs)
        mean = self.dense_mean(x)
        logvar = self.dense_logvar(x)
        z = self.sampler([mean, logvar])
        return z, mean, logvar

class Decoder(Layer):
    def __init__(self, z_dim, name='decoder'):
        super(Decoder, self).__init__(name=name)

        self.forward = Sequential([
                        Dense(8*8*64, activation='relu'),
                        Reshape((8,8,64)),
                        UpConvBlock(filters=64, kernel_size=(3,3)),
                        UpConvBlock(filters=64, kernel_size=(3,3)),
                        UpConvBlock(filters=32, kernel_size=(3,3)),
                        UpConvBlock(filters=32, kernel_size=(3,3)),
                        Conv2D(filters=3, kernel_size=(3,3), strides=1, padding='
                            same', activation='sigmoid'),

        ])

    def call(self, inputs):
        return self.forward(inputs)


class VAE(Model):
    def __init__(self, z_dim, name='VAE'):
        super(VAE, self).__init__(name=name)
        self.encoder = Encoder(z_dim)
        self.decoder = Decoder(z_dim)
        self.mean = None
        self.logvar = None

    def call(self, inputs):
        z, self.mean, self.logvar = self.encoder(inputs)
        out = self.decoder(z)
        return out

def log_normal_pdf(sample, mean, logvar, raxis=1):
    log2pi = tf.math.log(2. * np.pi)
    return tf.reduce_sum(
```

```
        -.5 * ((sample - mean) ** 2. * tf.exp(-logvar) + logvar + log2pi),
        axis=raxis)

def compute_loss(model, x,z_augmented):
    z, mean, logvar = model.encoder(x)
    x_logit = model.decoder(z_augmented)

    # Reconstruction loss
    cross_ent = tf.nn.sigmoid_cross_entropy_with_logits(logits=x_logit, labels=x)
    logpx_z = -tf.reduce_sum(cross_ent, axis=[1, 2, 3])

    # KL divergence loss
    logpz = log_normal_pdf(z, 0., 0.)
    logqz_x = log_normal_pdf(z, mean, logvar)

    # Total loss
    loss = -tf.reduce_mean(logpx_z + logpz - logqz_x)

    return loss
```

Code Listing 4: Individual Augmentation Functions

```
# This code subsection consists of the three different augmentations we apply to
    the latent vectors.
def cutout_augmentation(z, mask_size=8):
    batch_size, latent_dim = z.shape
    cutout_z = tf.identity(z)  # Create a copy of z to preserve the original
        values
    mask_indices = tf.random.uniform((batch_size, mask_size), minval=0, maxval=
        latent_dim, dtype=tf.int32)  # Generate random indices
    # Create mask tensor
    mask = tf.one_hot(mask_indices, depth=latent_dim, dtype=z.dtype)
    mask = tf.reduce_sum(mask, axis=1)
    # Apply mask to the latent vectors
    cutout_z = z * (1 - mask)
    return cutout_z

def mixup_augmentation(z1, z2, mixup_alpha):
    # Cast tensors to float32 to ensure consistency
    z1 = tf.cast(z1, tf.float32)
    z2 = tf.cast(z2, tf.float32)
    mixed_z = mixup_alpha * z1 + (1 - mixup_alpha) * z2  # Mix latent vectors
        based on coefficients
    return mixed_z

def gaussian_noise_augmentation(z, noise_std):
    # Generate Gaussian noise with the same shape as z
    noise = tf.random.normal(shape=z.shape, mean=0.0, stddev=noise_std)
    # Add the noise to the latent vectors
    noisy_z = z + noise
    return noisy_z
```

Code Listing 5: Composed Augmentation Functions

```
# Define latent space augmentations functions in a composed framework
def latent_space_augmentations(z, validation_loss_category, noise_std,
    cutout_mask_size, mixup_alpha):
    if validation_loss_category == 'low':
        # Apply gaussian noise augmentation
        z_augmented = gaussian_noise_augmentation(z, noise_std)
    elif validation_loss_category == 'medium':
```

```
        # Apply cutout augmentation
        z_augmented = cutout_augmentation(z, cutout_mask_size)
    elif validation_loss_category == 'high':
        # Apply mixup augmentation
        z_augmented = mixup_augmentation(z, tf.reverse(z, axis=[0]), mixup_alpha)
    else:
        raise ValueError("Invalid validation loss category.")

    return z_augmented
```

Code Listing 6: Incorporating Adaptivity to Augmentation Framework

```
# This code subsection consists of incorporating the adaptivity to the
    augmentation framework
# The adaptivity is implemented in three parts:
##    1. Validation Loss Category Determination
##    2. Validation Loss Computation
##    3. Hyperparameter Optimization

# 1. Validation Loss Category Determination
# Determining validation loss category

def determine_augmentation_technique(validation_losses, last_n_losses=3):
    # Consider the trend of last N validation losses
    trend = np.diff(validation_losses[-last_n_losses:])
    print(f"trend:{trend}")
    # Calculate the average validation loss
    avg_loss = np.mean(validation_losses[-last_n_losses:])
    print(f"avg_loss:{avg_loss}")
    # If the validation loss is decreasing and below a certain threshold, use
        cutout
    if np.all(trend < 0) and avg_loss < 0.05:
        return 'low'

    # If the validation loss is decreasing and below a certain threshold, use
        mixup
    elif np.all(trend < 0) and avg_loss > 0.05:
        return 'medium'
    elif np.all(trend < 0) and avg_loss > 0.15:
        return 'high'
    # If the validation loss is stable or slightly increasing, use random
        flipping
    else:
        print("Valiation loss is stable and slightly increasing")
        return 'high'

# 2. Validation Loss Computation
# Mean Squared Error (MSE)
def compute_mse(images1, images2):
    if images1.shape != images2.shape:
        raise ValueError("Shapes of input images must be the same.")
    return np.mean((images1 - images2)**2)

# Function to compute validation loss
def compute_validation_loss(model, batch_images, z_augmented):
    decoded_imgs = model.decoder(z_augmented)
    val_loss_tf = compute_mse(batch_images, decoded_imgs.numpy())
    return val_loss_tf

def hyp_validation_loss(model, validation_images, noise_std, cutout_mask_size,
    mixup_alpha):
```

```
    batch_size = 32  # Define batch size for validation
    num_batches = len(validation_images) // batch_size  # Calculate the number of
        batches
    gaussian_losses =[]
    # Initialize lists to store losses for each batch
    cutout_losses = []
    mixup_losses = []

    # Iterate over batches
    for i in range(num_batches):
        # Get batch of validation images
        batch_images = validation_images[i * batch_size: (i + 1) * batch_size]
        z_test, encoded_imgs_mean, encoded_imgs_logvar = model.encoder(
            batch_images)
        z_gauss_augmented = latent_space_augmentations(z_test,'low', noise_std,
            cutout_mask_size, mixup_alpha)
        gauss_loss = compute_validation_loss(model, batch_images,
            z_gauss_augmented)
        # Calculate validation loss for cutout augmentation
        z_cutout_augmented = latent_space_augmentations(z_test,'medium',
            noise_std, cutout_mask_size, mixup_alpha)
        cutout_loss = compute_validation_loss(model, batch_images,
            z_cutout_augmented)
        # Calculate validation loss for mixup augmentation
        z_mixup_augmented = latent_space_augmentations(z_test, 'high', noise_std,
            cutout_mask_size, mixup_alpha)
        mixup_loss = compute_validation_loss(model, batch_images,
            z_mixup_augmented)
        gaussian_losses.append(gauss_loss)
        cutout_losses.append(cutout_loss)
        mixup_losses.append(mixup_loss)

    # Take the average of losses across all batches
    avg_gauss_loss = np.mean(gaussian_losses)
    avg_cutout_loss = np.mean(cutout_losses)
    avg_mixup_loss = np.mean(mixup_losses)

    # Choose the minimum loss between cutout and mixup augmentation
    validation_loss = min(avg_gauss_loss, avg_cutout_loss, avg_mixup_loss)

    # Convert validation loss to TensorFlow tensor
    val_loss_tf = tf.constant(validation_loss, dtype=tf.float32)

    return val_loss_tf

#3. Hyperparameter Optimization
# Hyperparameter Optimization
def update_parameters(model, validation_images):
    # Define search space for hyperparameters
    gauss_noises = np.linspace(0.1,0.5,5)
    cutout_mask_sizes = np.arange(1, 21)  # Assuming maximum mask size of 20
    mixup_alphas = np.linspace(0.1, 0.9, 9)  # Assuming mixup alpha values from
        0.1 to 0.9

    # Evaluate MSE for the new hyperparameters
    mse_scores = []

    for _ in range(5):  # Evaluate MSE for 5 different random samples
        # Calculate validation loss using the new hyperparameters
        # Randomly sample new hyperparameters
        new_gaussian_noise = np.random.choice(gauss_noises)
```

```
        new_cutout_mask_size = np.random.choice(cutout_mask_sizes)
        new_mixup_alpha = np.random.choice(mixup_alphas)
        updated_validation_loss = hyp_validation_loss(model, validation_images,
            new_gaussian_noise, new_cutout_mask_size, new_mixup_alpha)
        mse_scores.append((new_gaussian_noise, new_cutout_mask_size,
            new_mixup_alpha, updated_validation_loss))


    # Find the hyperparameters with the lowest MSE score
    best_hyperparameters = min(mse_scores, key=lambda x: x[3])

    # Retrieve the best hyperparameters
    best_gaussian_noise, best_cutout_mask_size, best_mixup_alpha, best_loss =
        best_hyperparameters

    print(f"Lowest loss:{best_loss}")

    # Here, let's just update parameters based on random search
    updated_gaussian_noise = tf.Variable(best_gaussian_noise, trainable=False,
        dtype=tf.float32)
    updated_cutout_mask_size = tf.Variable(best_cutout_mask_size, trainable=False
        , dtype=tf.int32)
    updated_mixup_alpha = tf.Variable(best_mixup_alpha, trainable=False, dtype=tf
        .float32)

    return best_loss, updated_gaussian_noise, updated_cutout_mask_size,
        updated_mixup_alpha
```

Code Listing 7: Instantiation and Training the Model

```
# This code subsection consists of the Training setup and Instantiation

# Instantiate the model and loss function
z_dim = 64
model = VAE(z_dim)

# Define optimizer
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-4)

# Define training step
@tf.function
def train_step(model, images, optimizer):
    with tf.GradientTape() as tape:
        z, mean, logvar = model.encoder(images)
        loss = compute_loss(model,images,z)
        #print(f"training loss type : {type(loss)}")
    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))
    return loss

# Define training step with adaptive augmentations and learnable parameters
@tf.function
def train_step_with_augmentations_learnable(model, images, optimizer,
    cutout_mask_size, mixup_alpha,validation_loss_category):
    with tf.GradientTape() as tape:

        z, mean, logvar = model.encoder(images)
        # calculate loss
        z_augmented = latent_space_augmentations(z, validation_loss_category,
            cutout_mask_size, mixup_alpha)
        loss = compute_loss(model,images,z_augmented)
```

```
    # Backpropagation
    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))

    return loss


# Define a function to create batches of image filenames
def batch_generator(image_filenames, batch_size):
    for i in range(0, len(image_filenames), batch_size):
        yield image_filenames[i:i+batch_size]


def vector_generation_images_batch(images_batch):

  z, mean, logvar = model.encoder(images_batch)
  print(type(z))
  z_numpy_array = z.numpy()
  print(type(z_numpy_array))
  return z_numpy_array

epochs = 100
batch_size = 32
random_vector_for_generation = np.random.normal(loc=0, scale=1, size=(16, z_dim))

# Define adaptive augmentation parameters as trainable variables
gaussian_noise = tf.Variable(initial_value=0.1, trainable= False, dtype=tf.
    float32, name='gaussian_noise')
cutout_mask_size = tf.Variable(initial_value=10, trainable= False, dtype=tf.int32
    , name='cutout_mask_size')
mixup_alpha = tf.Variable(initial_value=0.2, trainable= False, dtype=tf.float32,
    name='mixup_alpha')
# Plot the evolution of adaptive augmentation parameters
loss_history = []
gaussian_noises = []
cutout_mask_sizes = []
mixup_alphas = []
validation_losses = []

updated_gaussian_noise = gaussian_noise
updated_cutout_mask_size = cutout_mask_size
updated_mixup_alpha = mixup_alpha
validation_images = test_images
validation_loss_category = 'low'
# Training loop with adaptive augmentations and learnable parameters
for epoch in range(epochs):
    print(f"Epoch: {epoch}")
    start_time = time.time()
    epoch_loss_avg = tf.keras.metrics.Mean()
    # Iterate over training batches
    for batch_files in batch_generator(train_files, batch_size):
        # Preprocess images
        batch_images = preprocess_images(batch_files, target_size)

        # Use regular training step for the first epoch
        if epoch == 0:
            loss = train_step(model, batch_images, optimizer)
        else:
            loss = train_step_with_augmentations_learnable(model, batch_images,
                optimizer, updated_gaussian_noise, updated_cutout_mask_size,
                updated_mixup_alpha,validation_loss_category)
```

```
        epoch_loss_avg.update_state(loss)
    # Update parameters based on validation loss
    validation_loss, updated_gaussian_noise, updated_cutout_mask_size,
        updated_mixup_alpha = update_parameters(model, validation_images)
    validation_loss_category = determine_augmentation_technique(validation_losses
        )
    print(f"epoch, validation_loss_category: {epoch},{validation_loss_category}")
    gaussian_noises.append(updated_gaussian_noise.numpy())
    cutout_mask_sizes.append(updated_cutout_mask_size.numpy())
    mixup_alphas.append(updated_mixup_alpha.numpy())
    validation_losses.append(validation_loss)
    loss_history.append(epoch_loss_avg.result())
    elapsed_time = time.time() - start_time
    if epoch % 20 == 0:
        print(f"Epoch {epoch + 1}, Loss: {epoch_loss_avg.result()}, Time: {
            elapsed_time:.2f} sec")
        model.save_weights('../data/output/checkpoints/model_at_epoch_{:04d}.h5'.
            format(epoch))
        generate_images_random(model, epoch, random_vector_for_generation)
        plot_hyperparameters(gaussian_noises, cutout_mask_sizes,mixup_alphas,
            validation_losses,loss_history)
```