**Data Augmentation for New Data Generation**

from sklearn.model\_selection import train\_test\_split

from sklearn.utils import shuffle

from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Create an ImageDataGenerator with desired augmentation options

datagen = ImageDataGenerator(

rotation\_range=20,

width\_shift\_range=0.1,

height\_shift\_range=0.1,

zoom\_range=0.2,

horizontal\_flip=True

)

X\_train= np.array(X\_train).reshape(-1,28,28,3)

# Fit the ImageDataGenerator on the training data

datagen.fit(X\_train)

# Generate augmented data

augmented\_data = datagen.flow(X\_train, Y\_train, batch\_size=len(X\_train), shuffle=False)

# Extract augmented samples and labels

augmented\_samples, augmented\_labels = next(augmented\_data)

# Concatenate original and augmented data

balanced\_x = np.concatenate((X\_train, augmented\_samples))

balanced\_y = np.concatenate((Y\_train, augmented\_labels))

print(balanced\_x.shape, balanced\_y.shape)

**Random Over Sampling**

# To overcome class imbalace

oversample = RandomOverSampler()

balanced\_x,balanced\_y = oversample.fit\_resample(X\_train,Y\_train)

print(balanced\_x.shape,balanced\_y.shape)

**ADASYN (**Adaptive Synthetic Sampling**) Over Sampling for Data Balancing**

import numpy as np

from imblearn.over\_sampling import ADASYN

# Assuming you have your feature data 'X' and corresponding labels 'y'

# X: numpy array of shape (n\_samples, n\_features)

# y: numpy array of shape (n\_samples,)

# Instantiate the ADASYN oversampler

adasyn = ADASYN(random\_state=42)

# Resample the data using ADASYN

balanced\_x, balanced\_y = adasyn.fit\_resample(X\_train, Y\_train)

# Check the class distribution after resampling

unique, counts = np.unique(balanced\_y, return\_counts=True)

print(dict(zip(unique, counts)))

**Adversarial Noise Code**

import tensorflow as tf

def generate\_adversarial\_noise(X\_test, Y\_test, model, epsilon):

# Create a directory for the adversarial images

if not os.path.exists(f"test\_ADN\_{epsilon}"):

os.mkdir(f"test\_ADN\_{epsilon}")

# Create a copy of the input image

X\_test\_adn = X\_test.copy()

# Set the input tensor to be trainable

X\_test\_adn = tf.Variable(X\_test\_adn, dtype=tf.float32)

# Create a one-hot encoding of the target label

Y\_test\_onehot = tf.one\_hot(Y\_test, depth=7)

# Calculate the gradients of the loss with respect to the input

with tf.GradientTape() as tape:

prediction = model(X\_test\_adn)

loss = tf.keras.losses.categorical\_crossentropy(Y\_test\_onehot, prediction)

gradients = tape.gradient(loss, X\_test\_adn)

# Get the sign of the gradients

signed\_gradients = tf.sign(gradients)

# Add the noise to the image

X\_test\_adn.assign\_add(epsilon \* signed\_gradients)

# Clip the image to the valid pixel range

X\_test\_adn = tf.clip\_by\_value(X\_test\_adn, 0, 255)

# Convert the tensor back to a numpy array

X\_test\_adn = X\_test\_adn.numpy()

# Loop through each image in the test set

for i in range(len(X\_test\_adn)):

# Get the pixel values of the image

pixels = X\_test\_adn[i]

label = Y\_test.iloc[i]

# Save the noisy image to the new directory

cv2.imwrite(f"test\_ADN\_{epsilon}/{i}\_{label}.png", pixels)

return X\_test\_adn

**Translation Noise code**

from PIL import Image

import numpy as np

def add\_translation\_noise(image, max\_translation):

width, height = image.size

# Generate random translation values

translation\_x = np.random.randint(-max\_translation, max\_translation + 1)

translation\_y = np.random.randint(-max\_translation, max\_translation + 1)

# Create the translation matrix

translation\_matrix = (1, 0, translation\_x, 0, 1, translation\_y)

# Apply the translation to the image

translated\_image = image.transform((width, height), Image.AFFINE, translation\_matrix)

return translated\_image

# Example usage

image = Image.open('/kaggle/input/image-test/test.jpg') # Load your image here

# image = image.resize((28, 28)) # Resize the image to 28x28 if needed

max\_translation = 5 # Maximum translation in pixels

noisy\_image = add\_translation\_noise(image, max\_translation)

# Save the original and noisy images

image.save('original\_image.jpg')

noisy\_image.save('noisy\_image.jpg')

**Scalling Noise code**

from PIL import Image

import numpy as np

import os

def add\_scaling\_noise(image, min\_scale, max\_scale):

width, height = image.size

# Generate a random scaling factor

scaling\_factor = np.random.uniform(min\_scale, max\_scale)

# Calculate the new dimensions

new\_width = int(width \* scaling\_factor)

new\_height = int(height \* scaling\_factor)

# Resize the image using the new dimensions

scaled\_image = image.resize((new\_width, new\_height))

return scaled\_image

# Example usage for a test dataset

dataset\_path = 'path/to/your/dataset' # Path to the directory containing your test dataset

output\_path = 'path/to/save/noisy/dataset' # Path to save the noisy dataset

min\_scale = 0.5 # Minimum scaling factor (e.g., 50% of the original size)

max\_scale = 1.5 # Maximum scaling factor (e.g., 150% of the original size)

# Iterate over the images in the dataset directory

for filename in os.listdir(dataset\_path):

if filename.endswith('.jpg') or filename.endswith('.png'):

image\_path = os.path.join(dataset\_path, filename)

image = Image.open(image\_path)

# Apply scaling noise

noisy\_image = add\_scaling\_noise(image, min\_scale, max\_scale)

# Save the noisy image to the output directory

noisy\_image\_path = os.path.join(output\_path, filename)

noisy\_image.save(noisy\_image\_path)