Traffic Sign Image Generation & Detection for Autonomous Vehicles Using GANs

In this blog post, we will walk through the process of building a Deep Convolutional Generative Adversarial Network (DCGAN) for traffic sign generation and detection for autonomous vehicles. The code provided will load and preprocess the German Traffic Sign (GTSDB) dataset, create the generator and discriminator models, and train the DCGAN.

Importance of Traffic Sign Recognition for Autonomous Vehicles

Ensuring Road Safety with Traffic Sign Recognition

The recognition of traffic signs is a critical component in the development and safe operation of autonomous vehicles. It enables self-driving vehicles to comprehend and follow traffic regulations, protecting the safety of passengers, pedestrians, and other road users. By recognising and understanding traffic signals such as traffic lights, stop signs, and speed limit signs, autonomous vehicles can negotiate complex road situations. Autonomous vehicles that obey these signals can enhance traffic flow, minimise the likelihood of accidents, and contribute to a more efficient transportation system.

Staying within the Boundaries of the Law

Traffic signs are critical for implementing traffic laws and regulations. To comply with traffic laws, autonomous vehicles must be capable of recognising and interpreting these signs. This avoids fines, penalties, and accidents that may occur as a result of noncompliance.

Smarter Route Planning and Navigation

Routes, instructions, and upcoming exits are all provided through traffic signs. This data is used by autonomous vehicles to design the most efficient and safe path to their destination. Recognizing traffic signs such as "no entry" and "one-way" assists vehicles in avoiding restricted zones and wrong-way travel, resulting in smooth navigation.

Maintaining Efficient Traffic Flow

Autonomous vehicles can contribute to a smooth traffic flow by recognising traffic signs. For instance, by identifying a "yield" or "stop" sign, vehicles can give way to other road users, reducing the risk of traffic jams or accidents. Proper traffic sign recognition allows autonomous vehicles to drive in harmony with human drivers, improving overall traffic conditions.

Enhancing the Driving Experience

Passengers benefit from a more comfortable and convenient driving experience when autonomous vehicles effectively recognise and respond to traffic signs. A vehicle that recognizes a "school zone" sign and lowers its speed, for example, is less likely to surprise people or necessitate abrupt braking. This guarantees that passengers have a comfortable and enjoyable ride.

Generative Adversarial Networks(GANs)

GANs are a type of artificial neural network that was introduced in 2014 by Ian Goodfellow and his colleagues. They are known for their impressive capabilities. GANs are composed of two neural networks, namely a

generator and a discriminator. These networks engage in a process called adversarial training, where they compete against each other. The generator produces artificial data, whereas the discriminator has to differentiate between authentic and synthesised data. As time progresses, the networks enhance their performance, leading to the generator generating more authentic data.

Relevance to Traffic Sign Recognition

Traffic sign recognition plays a vital role in ensuring the safety and smooth operation of autonomous vehicles. Accurate recognition allows self-driving vehicles to adhere to traffic rules, navigate complex road environments, and avoid accidents. However, training an effective traffic sign recognition model requires a large and diverse dataset of traffic signs.

GANs can be instrumental in addressing this challenge by generating additional synthetic traffic sign images. These synthetic images can be used to augment existing datasets, improving the model's ability to recognize various traffic signs under different conditions, such as lighting, weather, and occlusion.

Moreover, GANs can be used to generate traffic signs in different styles, simulating the appearance of traffic signs in different countries or regions. This can help autonomous vehicles adapt to new environments more quickly and perform better in diverse settings.

Reinforcement Learning in the Wild

In the paper, “Virtual to Real Reinforcement Learning for Autonomous Driving”, the authors first educated it virtually to teach a deep reinforcement learning agent how to drive on a simulated road and obey traffic laws. Then, by fine-tuning the learnt agent on a smaller collection of real-world data, they applied it to a real-world driving scenario. Thanks to this transfer learning strategy, the agent may adapt to the physical environment while keeping the skills acquired in the virtual one. Given a picture from one modality, image translation attempts to anticipate an image in a different modality. The research studied the application of VAE-GAN in producing 3D voxel models, and the research presented a cascade GAN to create unrealistic images via structure and style.

Domain Shift in Image Classification Tasks

Because this work provides a framework for learning from synthetic data, which is an essential step towards addressing domain shift in autonomous driving scenarios where there is no training data available due to a lack of labelled images or videos depicting such systems, it can be applied to autonomous vehicle traffic sign and obstacle detection.

The paper “Learning from Synthetic Data: Addressing Domain Shift for Semantic Segmentation” presented a method based on Generative Adversarial Networks (GANs) to bring the embeddings closer together in the learned feature space. Although the paper focuses on semantic segmentation, the proposed methodology applies to a traffic sign and obstacle detection tasks. Furthermore, GANs can help deep learning models perform better by generating more realistic synthetic data, even when real-world training data is limited.

The issues of domain adaption and data augmentation in applications for autonomous vehicles can be significantly improved by GAN. Generally, GANs are hard to train on tasks involving realistic images of a larger scale. The Auxiliary Classifier GAN (AC-GAN) approach by Odena et al. is one promising approach to training stable generative models with the GAN framework.

Types of GANs

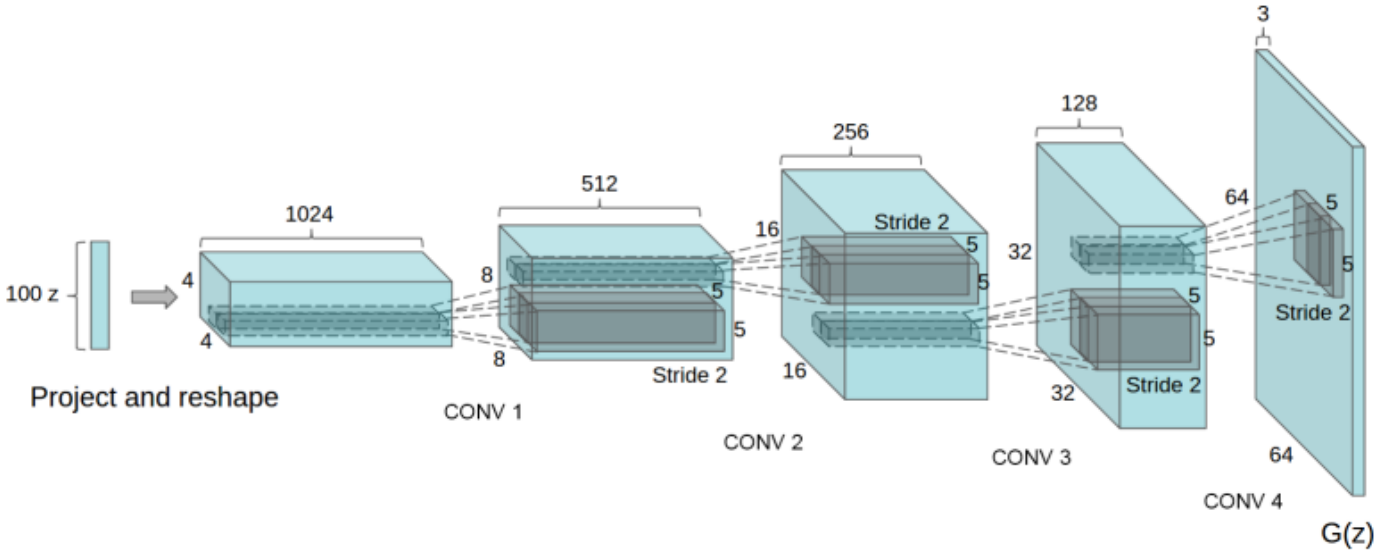
Since their inception, Generative Adversarial Networks (GANs) have rapidly evolved, resulting in a wide range of

GAN types and architectures.

Deep Convolutional Generative Adversarial Networks (DCGAN)

DCGAN is intended to address some problems encountered in previous GAN models, such as mode collapse (where the generator learns to produce a limited set of similar images) and training instability. During training, the generator network learns to generate realistic images to deceive the discriminator network, while the discriminator network learns to differentiate between real and generated images. For the sake of our blog post, We have used DCGAN as out main network for code implementation.

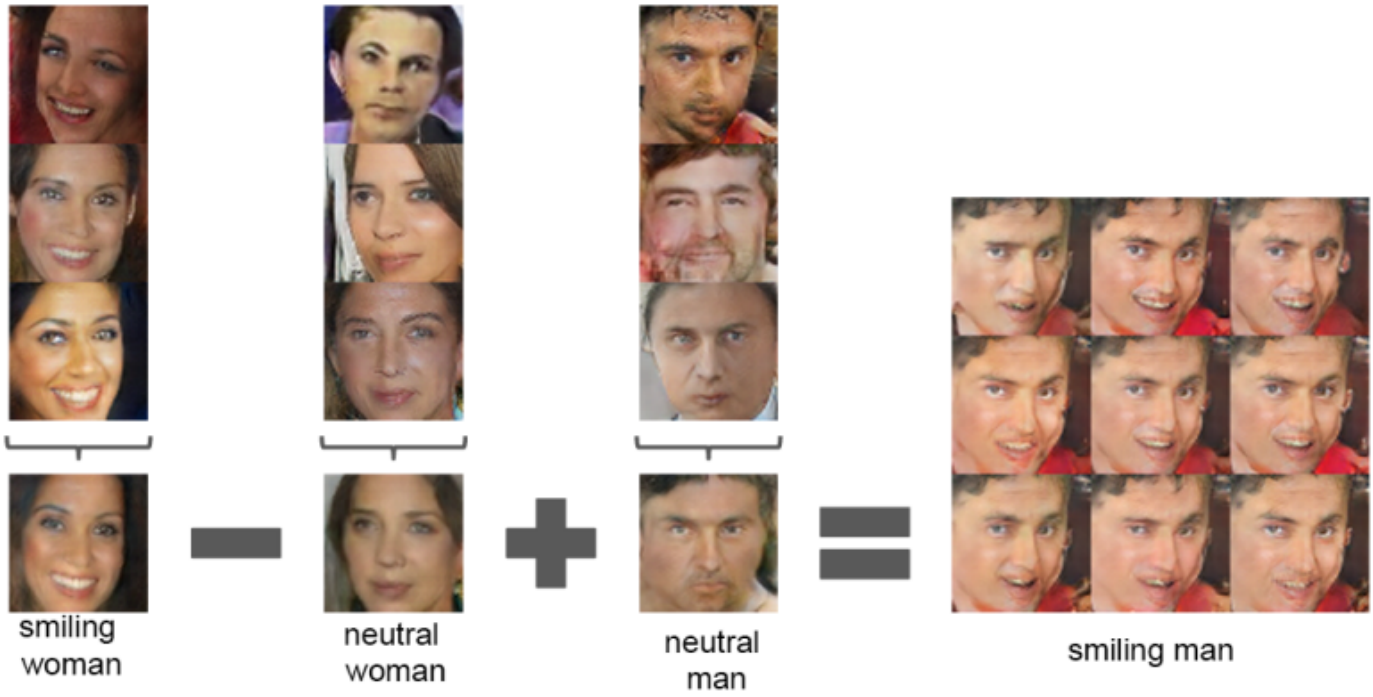
The numbers generated by the DCGAN are much clearer than those generated by the GAN.



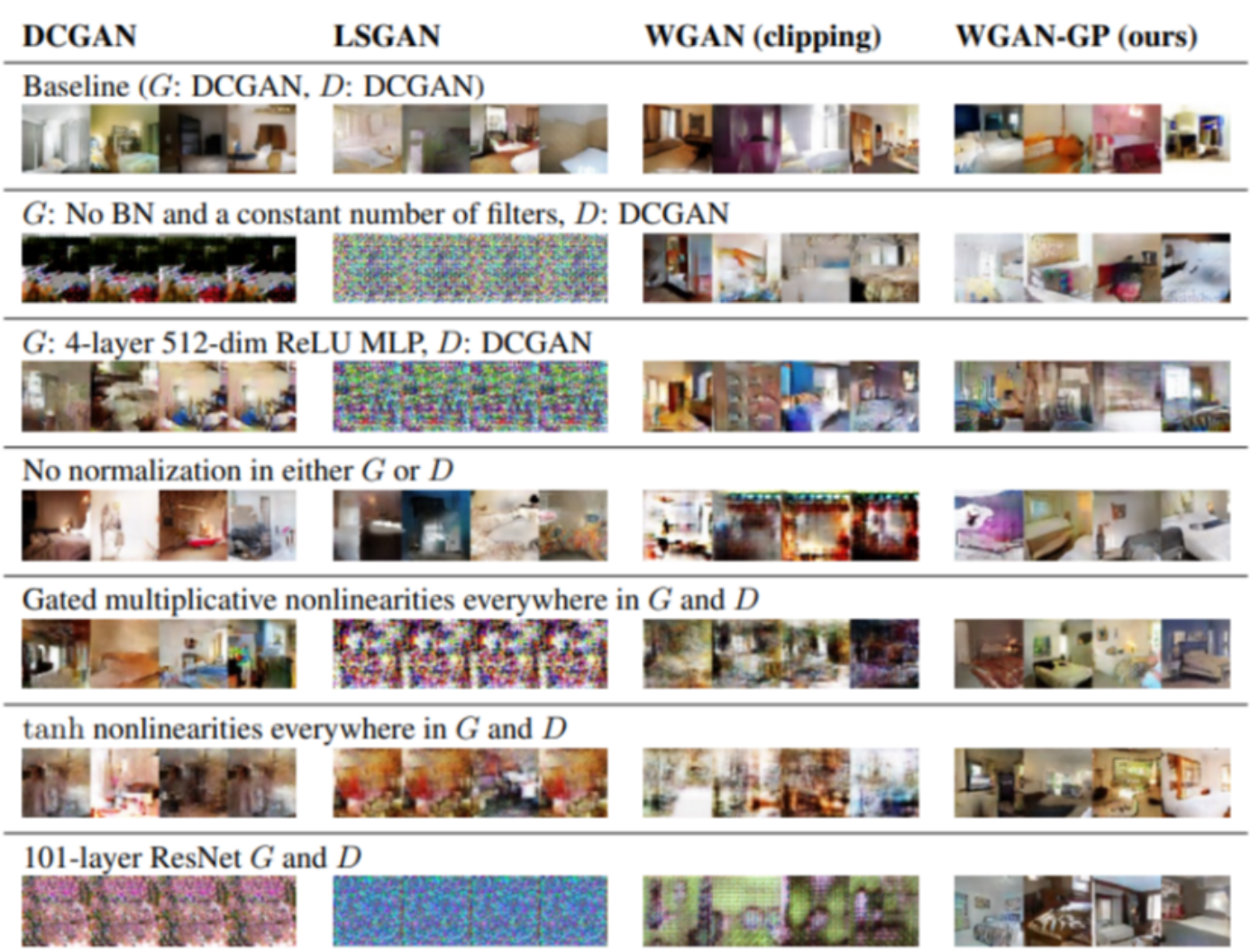
• Replacing any pooling layers with stride convolutions in the discriminator and fractional-strided convolutions in the generator.\ • To stabilise training, batch normalisation is used in both the generator and the discriminator.\ • Using a more convolutional architecture instead of fully connected hidden layers.\ • All layers in the generator use ReLU activation except the output, which uses the Tanh function.\ • For all layers, Leaky ReLU activation is used in the discriminator.

DCGAN Experiments

The above pictures are fake bedroom images created through the DCGAN model. These are fake bedroom images created by the generator, which was learned by turning five epoxies with a set of bed images, and they show high performance enough to be actual bedroom images.

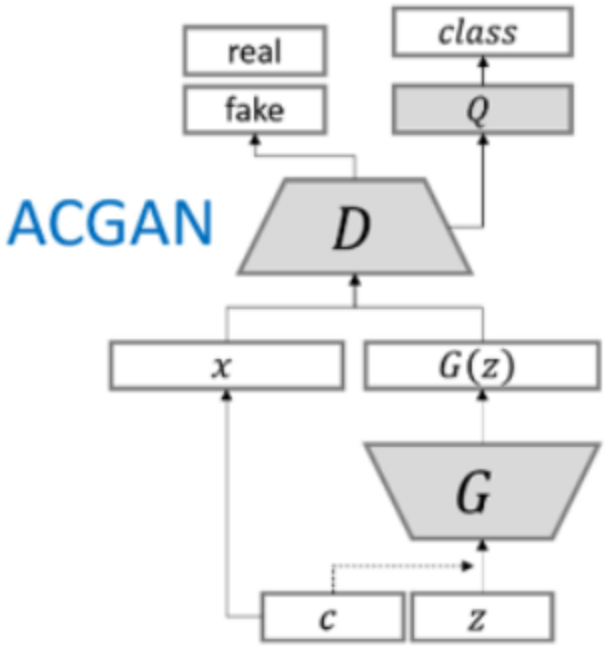


The 'smiling Woman' image datasets were removed from the 'neutral Woman' image datasets and the netural man' datasets was added again to obtain a fake image of 'smiling Man'.



Auxiliary Classifier Generative Adversarial Networks (AC-GAN)

AC-GAN is a conditional GAN extension that modifies the discriminator to predict the class label of a given image rather than receiving it as input. It stabilises the training process and enables the generation of large-high-quality images while learning a representation in the latent space independent of the class label. The AC-GAN model's structure allows for the classification of large datasets and the training of a generator and discriminator for each subset.



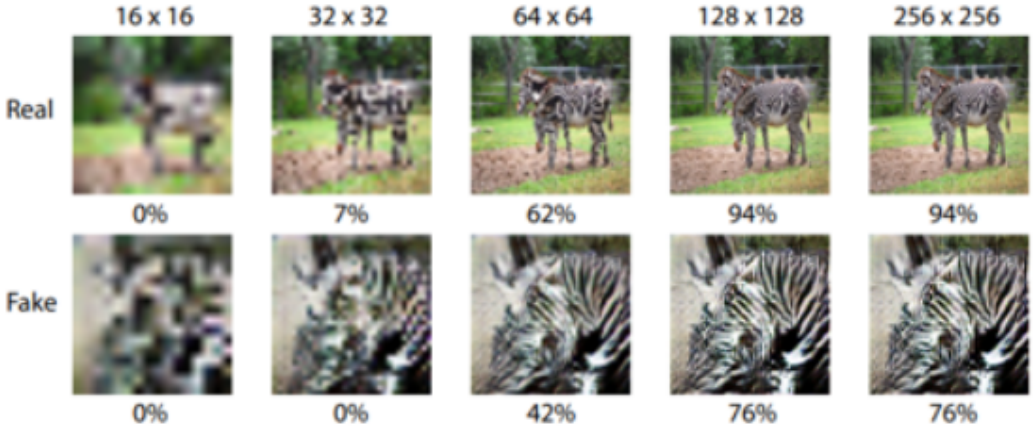
$$L_S = E[log\ p(S = real|X_{real})] + E[log\ p(S = fake|X_{fake})]$$

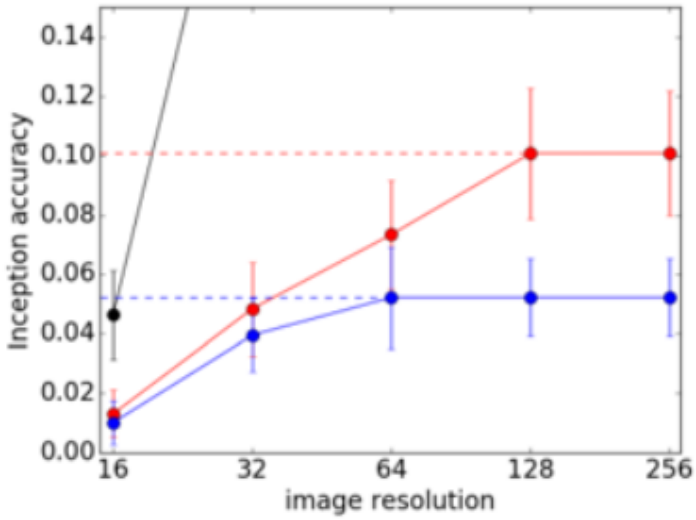
$$L_C = E[log\ p(C = c \quad |X_{real})] + E[log\ p(C = c \quad |X_{fake})]$$

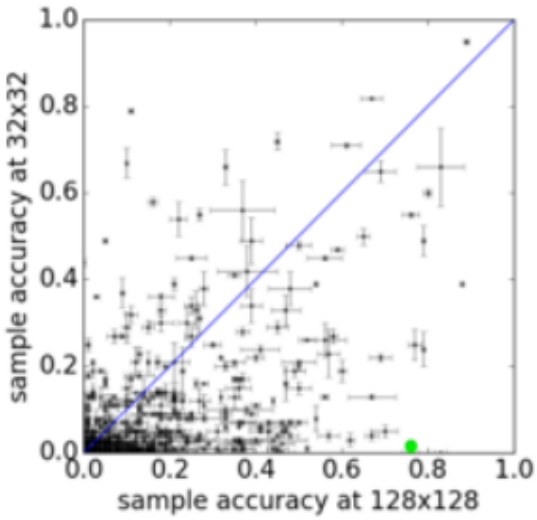
The AC-GAN structure allows dividing large datasets into subsets by class and training a generator and discriminator for each subset. Ls is the same as the objective function of the existing GAN. That is, it is related to determining real/fake. Lc is concerned with determining the class of the data. It's a little similar to what I saw on CGAN. D(discriminator) is trained to maximise Ls+Lc. G(generator) is trained to maximise Lc-Ls.

The AC-GAN framework can generate realistic images of traffic signs and obstacles for deep learning models. In addition, AC-GANs can generate diverse and representative synthetic data by conditioning the generated images on class labels, which can improve the performance of traffic sign and obstacle detection models, especially when real-world training data is scarce.

AC-GAN Experiments







Creating images with high resolution improves discriminability. AC-GANs can generate ImageNet samples that are globally coherent.

Implementation of Bidirectional Generative Adversarial Networks (BiGAN)

In their current state, GANs cannot learn inverse mapping, which involves projecting data back into the latent space. However, an unsupervised feature learning framework, BiGAN, can learn this inverse mapping and make it suitable for supplemental supervised discrimination tasks. Inverse mapping means projecting the image back into latent space.

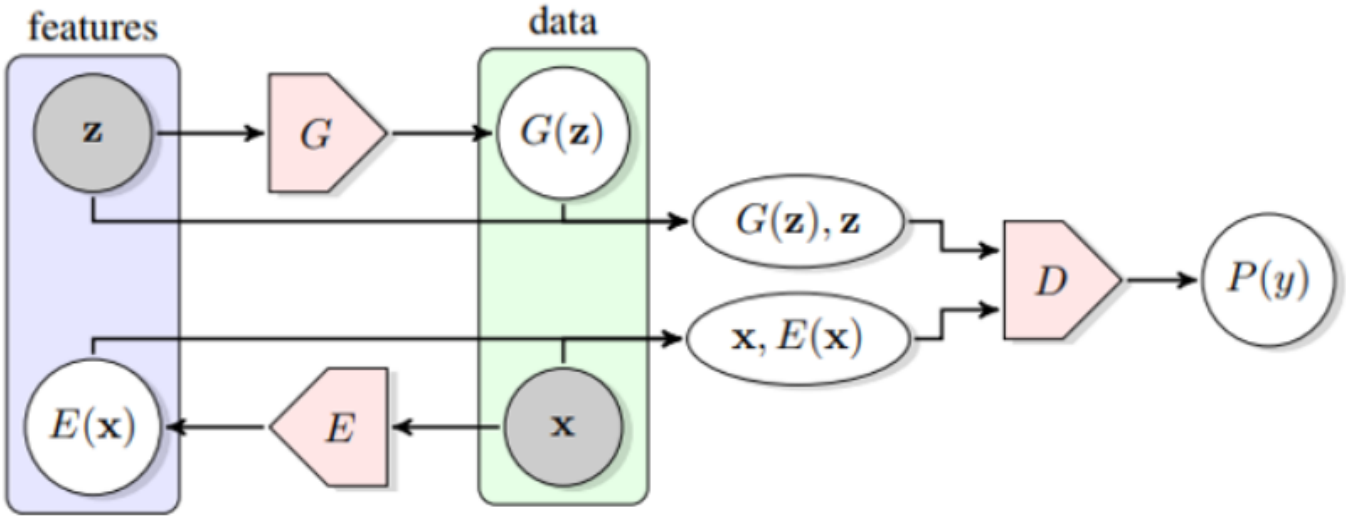


Figure 1: The structure of Bidirectional Generative Adversarial Networks (BiGAN).

G: Generator
E(x) : Encoder
D: Discriminator discriminating in data space(x versus G(z)) and data and latent space(tuples(x, E(x)) versus (G(z),z).
Z: Generator input

To completely deceive discriminators, encoders and generators must learn the complete opposite mapping.

BiGAN Training Objective Function

$$\min_G \max_D V(G,D)$$

$$= \mathrm{E}_{x\sim p_{data}(x)}[\log D(x,E(x))] + \mathrm{E}_{x\sim p_z(z)}[\log(1-D(G(z),z))]$$

Utilising the same alternating gradient-based optimisation, we optimise this minimax goal.

Background

Different Traffic Sign Recognition Techniques

Traditional Image Processing Techniques

These methods involve processing and analysing photos with algorithms that do not require data learning. Some common traditional image processing techniques for traffic sign recognition include:

• Colour-based segmentation: This method identifies traffic signs in an image by filtering out specific colours (e.g., red, yellow, or green for traffic lights) and extracting relevant regions.

• Shape-based detection: After colour segmentation, shape-based detection algorithms, such as Hough transform or contour analysis, can be used to identify traffic signs based on their geometric properties (e.g., circular or rectangular shapes).

• Feature extraction and matching: Techniques like Scale-Invariant Feature Transform (SIFT) or Speeded-Up

Robust Features (SURF) can be used to extract distinctive features from traffic signs, which can then be matched against a database of known traffic sign features to recognize them.

Machine Learning-based Techniques

Training a model to recognise traffic signs using machine learning approaches entails learning patterns and features from labelled data. Some common machine learning-based techniques for traffic sign recognition include:

• Support Vector Machines (SVM): SVM is a supervised learning algorithm that may be used to recognise traffic signs by training a classifier to distinguish between distinct signal classes based on their attributes.

• Decision Trees and Random Forests: By recursively partitioning the feature space into regions associated with different signal kinds, these tree-based approaches can be utilised to categorise traffic signs.

• Deep Learning: Deep learning techniques, particularly Convolutional Neural Networks (CNNs), have shown outstanding performance in traffic sign recognition tasks. CNNs can learn hierarchical characteristics from raw picture data, making them ideal for dealing with fluctuations in illumination, weather, and traffic sign types.

GAN Architecture and Hyper-parameters

Learning Rate

The learning rate is a crucial hyperparameter that influences the optimizer's step size while training. Selecting a suitable learning rate is crucial for ensuring the stability and convergence of the training procedure. Frequently employed learning rates typically fall within the range of 1E-4 to 1E-2 and may vary based on the optimizer and architecture.

Batch Size

The batch size refers to the quantity of samples that are employed for every update of the network weights. Increasing the batch size may enhance the stability of the training process; however, it necessitates a larger memory allocation. Batch sizes commonly vary between 16 and 512.

Generator architecture

The architecture of the generator network impacts the quality and diversity of generated samples. Choices such as the number of layers, the number of units per layer, and the type of activation functions can all affect performance.

Discriminator architecture

The discriminator network's architecture should be designed to effectively distinguish between real and generated samples. Similar to the generator, choices such as the number of layers, units per layer, and activation functions can impact performance.The discriminator network's architecture should be designed to effectively distinguish between real and generated samples. Similar to the generator, choices such as the number of layers, units per layer, and activation functions can impact performance.

Latent space dimension

The dimensionality of the latent space (input noise vector) can influence the quality and diversity of the generated samples. Typical latent space dimensions range from 100 to 512.

Loss Function

A loss function, also known as a cost function, is a mathematical function used in machine learning models to quantify the disparity between the predicted output and the actual output (or target). The metric evaluates the model's training error, aiding in optimisation. Frequently used loss functions comprise mean squared error (MSE) for regression assignments and cross-entropy loss for classification assignments.

Gradient

The gradient is a mathematical vector that shows how each of a function's input variables affects its partial derivatives.Gradients are used in machine learning to ascertain the magnitude and direction of modifications to the model's parameters, such as weights and biases, that will reduce the loss function. The gradient indicates the direction of maximum increase in the function, while its opposite direction indicates the direction of maximum decrease.

Optimizer

An optimizer is a computational method that modifies the model's parameters, such as weights and biases, to reduce the loss function during the training process. Gradient descent is a fundamental optimisation algorithm that involves updating the parameters through the subtraction of the gradient multiplied by a learning rate. Advanced optimizers like stochastic gradient descent (SGD), momentum, RMSprop, and Adam utilise techniques such as momentum and adaptive learning rates to enhance convergence and decrease training time.

Epoch

An epoch refers to a full cycle through a dataset while training a machine learning model. During each epoch, the model looks at the whole dataset once and makes changes to its parameters based on the loss.The count of epochs is a hyperparameter that decides the number of iterations the model will perform on the complete dataset. Increasing the number of epochs can improve the performance of the model, but it may also increase the risk of overfitting and require more training time.

Dataset Collection and preprocessing (GTSDB, GTSRB)

Of the two mentioned datasets here, I would like to mention that GTSDB dataset was used for Keras-GAN and Tensorflow but if you would like to implement PyTorch, it already comes with the GTSRB dataset. The reasoning behind using GTSDB was the easy access we had to the dataset from the internet and a ‘LETS GO’ attitude. This is a brief explanation of these datasets:

GTSDB (German Traffic Sign Detection Benchmark)

The GTSDB dataset comprises 900 images that have more than 1,200 annotated traffic signs. The dataset has been created to facilitate object detection tasks in the fields of computer vision and machine learning research. The GTSDB dataset differs from GTSRB in that it concentrates on both detection and recognition of traffic signs in images, while GTSRB only focuses on classification. The dataset comprises diverse traffic signs captured in different lighting conditions, scales, rotations, and occlusions.

GTSRB (German Traffic Sign Recognition Benchmark)

The GTSRB collection includes over 50,000 real-world traffic sign photos broken down into 43 classes. The dataset is used in machine learning and computer vision research for multi-class, single-image classification tasks. The photos are not guaranteed to be the same scale or rotation, and they vary in size. For developing and evaluating traffic sign recognition algorithms, many people use the GTSRB dataset.

Implementation Details

Implementing DCGAN &CNN Based On Modified GTSDB Dataset To Generate & Detect Traffic Signs

The whole working code with thier respective explanations along with ouput is provided below.

Importing Necessary Libraries To Implement DCGAN

```
os         : provides a way to interact with the operating system (e.g., read files
and directories)
numpy      : A library for numerical computing in Python
pandas     : A library for data manipulation and analysis
PIL        : A library for working with images
tensorflow : A popular open-source machine learning library
sklearn    : A library for machine learning and statistical modeling
matplotlib : A library for data visualization
```

In [2]:
```
import os
import numpy as np
import pandas as pd
from PIL import Image
import tensorflow as tf
from tensorflow.keras import layers
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
```

Preprocessing Image Data & Loading Datase

pd.read_csv(csv_path) reads in a CSV file using the pandas library and creates a DataFrame object called df.

images and **labels** will be populated with image data and corresponding labels, respectively. The **for** loop iterates through each row in the df DataFrame.

os.path.join(data_dir, row['Path']) creates the file path for the image file by joining the "data_dir" directory path with the "image path" in the Path column of the CSV file.

Image.open(img_path) opens the image file using the PIL library. **img.resize((32, 32))** resizes the image to a 32x32 pixel size.\ **np.array(img)** converts the PIL image object to a numpy array.\ **images.append(img_array)** appends the numpy array to the images list.\ **labels.append(row['ClassId'])** appends the class ID (label) to the labels list.

After the loop, the images and labels lists are converted to numpy arrays using **np.array()**.

The image data is normalized by dividing the pixel values by 127.5 and subtracting 1 to ensure the values are between -1 and 1.

In [54]:
```
def load_data(data_dir, csv_path):
    df = pd.read_csv(csv_path)
    images = []
    labels = []
    for idx, row in df.iterrows():
        img_path = os.path.join(data_dir, row['Path'])
        img = Image.open(img_path)
        img = img.resize((32, 32))
        img_array = np.array(img)
        images.append(img_array)
        labels.append(row['ClassId'])
    images = np.array(images)
    labels = np.array(labels)
    return images, labels
```

```
data_dir = "C:/Users/beneg/Downloads/GTSDB"
train_csv_path = os.path.join(data_dir, "Train_New.csv")
test_csv_path = os.path.join(data_dir, "Test.csv")

x_train, y_train = load_data(data_dir, train_csv_path)
x_test, y_test = load_data(data_dir, test_csv_path)

x_train = (x_train.astype("float32") - 127.5) / 127.5
x_test = (x_test.astype("float32") - 127.5) / 127.5
```

Defining The DCGAN Model by Building Generator & Discriminator

This code sets up the basic structure for a GAN model, including the generator, discriminator, and the full DCGAN model, and prepares the models for training using Adam optimizer and binary_crossentropy loss functions.

build_generator defines the generator model architecture, which takes in a **latent_dim** input and generates a 32x32 RGB image. The model includes several layers such as Dense, Reshape, and Conv2DTranspose layers to gradually upsample the input into a 32x32 image.

build_discriminator defines the discriminator model architecture, which takes in a 32x32 RGB image and classifies it as real or fake. The model includes several Conv2D and Dense layers to downsample the input and produce a binary classification output.

build_dcgan combines the generator and discriminator models into a single model, where the generator is trained to fool the discriminator by generating realistic images that are classified as real by the discriminator.

latent_dim is set to 100, representing the dimensionality of the input to the generator model.\ **input_shape** is set to (32, 32, 3), representing the shape of the input image to the discriminator model. **opt_gen** and **opt_disc** are defined as **Adam optimizers** with a learning rate of **0.0002** and a **beta_1** value of 0.5, for use in training the generator and discriminator models, respectively.

discriminator is compiled using binary cross-entropy loss and the opt_disc optimizer, and its accuracy is also tracked as a metric during training.\ **dcgan** is compiled using binary cross-entropy loss and the opt_gen optimizer. The discriminator is not compiled because its weights are frozen when training the full GAN model.

In [55]:
```
def build_generator(latent_dim):
    model = tf.keras.Sequential([
        layers.Dense(128 * 8 * 8, activation="relu", input_dim=latent_dim),
        layers.Reshape((8, 8, 128)),
        layers.UpSampling2D(),
        layers.Conv2D(128, kernel_size=3, padding="same", activation="relu"),
        layers.UpSampling2D(),
        layers.Conv2D(64, kernel_size=3, padding="same", activation="relu"),
        layers.Conv2D(3, kernel_size=3, padding="same", activation="tanh")
    ])
    return model

def build_discriminator(input_shape):
    model = tf.keras.Sequential([
        layers.Conv2D(64, kernel_size=3, strides=2, padding="same", input_shape=input_shap
        layers.LeakyReLU(alpha=0.2),
        layers.Dropout(0.3),
        layers.Conv2D(128, kernel_size=3, strides=2, padding="same"),
        layers.LeakyReLU(alpha=0.2),
        layers.Dropout(0.3),
        layers.Flatten(),
        layers.Dense(1, activation="sigmoid")
```

```
    ])
    return model

def build_dcgan(generator, discriminator):
    discriminator.trainable = False
    model = tf.keras.Sequential([generator, discriminator])
    return model

latent_dim = 100
input_shape = (32, 32, 3)
generator = build_generator(latent_dim)
discriminator = build_discriminator(input_shape)
dcgan = build_dcgan(generator, discriminator)

opt_gen = tf.keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.5)
opt_disc = tf.keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.5)

discriminator.compile(loss="binary_crossentropy", optimizer=opt_disc, metrics=["accuracy"]
dcgan.compile(loss="binary_crossentropy", optimizer=opt_gen)
```

Training & Saving The Generator & Discriminator models

Defining the Generator and Discriminator Networks

The DCGAN is built by combining two neural networks, namely the **generator** and the **discriminator**. The generator takes random noise as input and produces a new image, while the discriminator is a binary classifier that takes an image as input and returns a probability of it being real (1) or fake (0).

The first section of the code defines these networks and combines them to build the DCGAN.

The batch size is set to 64.\ The function sample_images is defined to generate and save sample images from the generator network. The current epoch number, generator, discriminator, latent dimension, and the path to save the generated images are provided as input. A grid of 25 images (5 rows and 5 columns) are generated by predicting the images from random noise. The generated images are saved as a PNG file based on the epoch number.

Setting the Hyperparameters and Parameters

The second section of the code defines the hyperparameters and parameters for training the DCGAN. It sets the number of epochs, batch size, and the directory to save the generated images. It also creates two sets of labels for the discriminator: one set with all ones for real images, and another set with all zeros for fake images.

The number of epochs is set to 100, and the number of steps per epoch is calculated based on the number of images in the training set and the batch size.\ The path to save the generated images is set to "C:/Users/beneg/Downloads/GTSDB/generated_images".\ The **real** and **fake** labels for the **discriminator** are defined as arrays of ones and zeros, respectively, with the shape **(batch_size, 1)**.\ It returns a probability of the images being **real (1) or fake (0)**.

Training the DCGAN

The third section of the code trains the DCGAN. For each epoch, it trains the discriminator first by taking a batch of real images and a batch of fake images generated by the generator, and computes the loss and accuracy. Then, it trains the generator by taking a batch of random noise and computing the loss. Finally, it saves the generated images periodically and saves the final generator and discriminator models.

The training loop runs for the specified number of epochs.\ For each epoch, it loops over the steps in the current epoch.\ A batch of real images are selecteda at random from the training set and generates a batch of fake images from random noise using the generator network.\ The discriminator is trained on the real and fake

images separately and calculates the loss as the average of the losses.\ The generator is trained using random noise and calculates the loss.

Generating Sample Images

The function sample_images is called to generate and save sample images from the generator network.

The current epoch number, step number, discriminator loss, discriminator accuracy, and generator loss is printed every 50 steps.\ A grid of generated images is saved every 10 epochs.

The generator models are saved in file named **generator_final_set.h5** and discriminator models are saved in file named **discriminator.h5**, respectively.

In [58]:

```
# First Section

batch_size = 64

def sample_images(epoch, generator, discriminator, latent_dim, save_path):
    r, c = 5, 5
    noise = np.random.normal(0, 1, (r * c, latent_dim))
    gen_imgs = generator.predict(noise)
    gen_imgs = 0.5 * gen_imgs + 0.5
    fig, axs = plt.subplots(r, c)
    cnt = 0
    for i in range(r):
        for j in range(c):
            axs[i, j].imshow(gen_imgs[cnt, :, :, :])
            axs[i, j].axis("off")
            cnt += 1
    fig.savefig(os.path.join(save_path, f"epoch_{epoch}.png"))
    plt.close()

# Second Section

epochs = 100
steps_per_epoch = x_train.shape[0] // batch_size
save_path = "C:/Users/beneg/Downloads/GTSDB/generated_images"
os.makedirs(save_path, exist_ok=True)

real_labels = np.ones((batch_size, 1))
fake_labels = np.zeros((batch_size, 1))

for epoch in range(epochs):
    for step in range(steps_per_epoch):
        # Train the discriminator
        idx = np.random.randint(0, x_train.shape[0], batch_size)
        real_images = x_train[idx]
        noise = np.random.normal(0, 1, (batch_size, latent_dim))
        fake_images = generator.predict(noise)

        d_loss_real = discriminator.train_on_batch(real_images, real_labels)
        d_loss_fake = discriminator.train_on_batch(fake_images, fake_labels)
        d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

# Third Section

        # Train the generator
        noise = np.random.normal(0, 1, (batch_size, latent_dim))
        g_loss = dcgan.train_on_batch(noise, real_labels)

        # Print progress
        if step % 50 == 0:
            print(f"Epoch {epoch}, Step {step}, D-Loss: {d_loss[0]}, D-Acc: {100 * d_loss
```

```python
    # Save generated images periodically
    if epoch % 10 == 0:
        sample_images(epoch, generator, discriminator, latent_dim, save_path)

# Save the final generator model
generator.save("generator_final_set.h5")
discriminator.save("discriminator.h5")
```

```
2/2 [==============================] - 0s 44ms/step
Epoch 0, Step 0, D-Loss: 1.461111694574356, D-Acc: 42.1875, G-Loss: 0.10914776474237442
2/2 [==============================] - 0s 48ms/step
2/2 [==============================] - 0s 63ms/step
1/1 [==============================] - 0s 47ms/step
2/2 [==============================] - 0s 53ms/step
Epoch 1, Step 0, D-Loss: 1.4748227894306183, D-Acc: 46.09375, G-Loss: 0.11077520996332169
2/2 [==============================] - 0s 29ms/step
2/2 [==============================] - 0s 47ms/step
2/2 [==============================] - 0s 39ms/step
Epoch 2, Step 0, D-Loss: 1.4538122713565826, D-Acc: 47.65625, G-Loss: 0.10524711012840271
2/2 [==============================] - 0s 56ms/step
2/2 [==============================] - 0s 47ms/step
2/2 [==============================] - 0s 43ms/step
Epoch 3, Step 0, D-Loss: 1.4492749571800232, D-Acc: 48.4375, G-Loss: 0.10745572298765182
2/2 [==============================] - 0s 48ms/step
2/2 [==============================] - 0s 58ms/step
2/2 [==============================] - 0s 37ms/step
Epoch 4, Step 0, D-Loss: 1.4788402020931244, D-Acc: 46.09375, G-Loss: 0.10882503539323807
2/2 [==============================] - 0s 32ms/step
2/2 [==============================] - 0s 31ms/step
2/2 [==============================] - 0s 31ms/step
Epoch 5, Step 0, D-Loss: 1.467747539281845, D-Acc: 47.65625, G-Loss: 0.10813499987125397
2/2 [==============================] - 0s 33ms/step
2/2 [==============================] - 0s 48ms/step
2/2 [==============================] - 0s 47ms/step
Epoch 6, Step 0, D-Loss: 1.4816865622997284, D-Acc: 45.3125, G-Loss: 0.10535633563995361
2/2 [==============================] - 0s 31ms/step
2/2 [==============================] - 0s 63ms/step
2/2 [==============================] - 0s 35ms/step
Epoch 7, Step 0, D-Loss: 1.4711078506946564, D-Acc: 46.875, G-Loss: 0.10157033801078796
2/2 [==============================] - 0s 35ms/step
2/2 [==============================] - 0s 57ms/step
2/2 [==============================] - 0s 33ms/step
Epoch 8, Step 0, D-Loss: 1.4801795482635498, D-Acc: 46.09375, G-Loss: 0.10533356666564941
2/2 [==============================] - 0s 37ms/step
2/2 [==============================] - 0s 32ms/step
2/2 [==============================] - 0s 47ms/step
Epoch 9, Step 0, D-Loss: 1.4770026803016663, D-Acc: 47.65625, G-Loss: 0.09989471733570099
2/2 [==============================] - 0s 50ms/step
2/2 [==============================] - 0s 39ms/step
2/2 [==============================] - 0s 47ms/step
Epoch 10, Step 0, D-Loss: 1.483162522315979, D-Acc: 47.65625, G-Loss: 0.10117088258266449
2/2 [==============================] - 0s 38ms/step
2/2 [==============================] - 0s 56ms/step
1/1 [==============================] - 0s 53ms/step
2/2 [==============================] - 0s 47ms/step
Epoch 11, Step 0, D-Loss: 1.4774686694145203, D-Acc: 44.53125, G-Loss: 0.09963217377662659
2/2 [==============================] - 0s 33ms/step
2/2 [==============================] - 0s 39ms/step
2/2 [==============================] - 0s 58ms/step
Epoch 12, Step 0, D-Loss: 1.4966202974319458, D-Acc: 45.3125, G-Loss: 0.09849157929420471
2/2 [==============================] - 0s 38ms/step
2/2 [==============================] - 0s 52ms/step
2/2 [==============================] - 0s 34ms/step
Epoch 13, Step 0, D-Loss: 1.5060155391693115, D-Acc: 42.96875, G-Loss: 0.10190156102180481
```

```
2/2 [==============================] - 0s 54ms/step
2/2 [==============================] - 0s 30ms/step
2/2 [==============================] - 0s 44ms/step
Epoch 14, Step 0, D-Loss: 1.5247148275375366, D-Acc: 46.875, G-Loss: 0.09873279184103012
2/2 [==============================] - 0s 36ms/step
2/2 [==============================] - 0s 49ms/step
2/2 [==============================] - 0s 43ms/step
Epoch 15, Step 0, D-Loss: 1.4832774996757507, D-Acc: 46.09375, G-Loss: 0.09747262299060822
2/2 [==============================] - 0s 60ms/step
2/2 [==============================] - 0s 36ms/step
2/2 [==============================] - 0s 39ms/step
Epoch 16, Step 0, D-Loss: 1.5174309313297272, D-Acc: 46.875, G-Loss: 0.09902487695217133
2/2 [==============================] - 0s 48ms/step
2/2 [==============================] - 0s 42ms/step
2/2 [==============================] - 0s 31ms/step
Epoch 17, Step 0, D-Loss: 1.489420771598816, D-Acc: 46.09375, G-Loss: 0.09886634349822998
2/2 [==============================] - 0s 39ms/step
2/2 [==============================] - 0s 53ms/step
2/2 [==============================] - 0s 42ms/step
Epoch 18, Step 0, D-Loss: 1.5022391080856323, D-Acc: 46.09375, G-Loss: 0.09703516215085983
2/2 [==============================] - 0s 38ms/step
2/2 [==============================] - 0s 26ms/step
2/2 [==============================] - 0s 36ms/step
Epoch 19, Step 0, D-Loss: 1.5323908030986786, D-Acc: 48.4375, G-Loss: 0.09705422818660736
2/2 [==============================] - 0s 42ms/step
2/2 [==============================] - 0s 37ms/step
2/2 [==============================] - 0s 48ms/step
Epoch 20, Step 0, D-Loss: 1.5196599662303925, D-Acc: 49.21875, G-Loss: 0.09296995401382446
2/2 [==============================] - 0s 32ms/step
2/2 [==============================] - 0s 32ms/step
1/1 [==============================] - 0s 54ms/step
2/2 [==============================] - 0s 49ms/step
Epoch 21, Step 0, D-Loss: 1.509431153535843, D-Acc: 45.3125, G-Loss: 0.0959690660238266
2/2 [==============================] - 0s 44ms/step
2/2 [==============================] - 0s 38ms/step
2/2 [==============================] - 0s 61ms/step
Epoch 22, Step 0, D-Loss: 1.5196900367736816, D-Acc: 47.65625, G-Loss: 0.09034629166126251
2/2 [==============================] - 0s 41ms/step
2/2 [==============================] - 0s 52ms/step
2/2 [==============================] - 0s 33ms/step
Epoch 23, Step 0, D-Loss: 1.513130784034729, D-Acc: 46.09375, G-Loss: 0.0934382975101471
2/2 [==============================] - 0s 32ms/step
2/2 [==============================] - 0s 64ms/step
2/2 [==============================] - 0s 38ms/step
Epoch 24, Step 0, D-Loss: 1.5376593172550201, D-Acc: 46.09375, G-Loss: 0.09580685198307037
2/2 [==============================] - 0s 45ms/step
2/2 [==============================] - 0s 37ms/step
2/2 [==============================] - 0s 54ms/step
Epoch 25, Step 0, D-Loss: 1.545970857143402, D-Acc: 46.09375, G-Loss: 0.09495623409748077
2/2 [==============================] - 0s 47ms/step
2/2 [==============================] - 0s 63ms/step
2/2 [==============================] - 0s 32ms/step
Epoch 26, Step 0, D-Loss: 1.5400753021240234, D-Acc: 48.4375, G-Loss: 0.08871212601661682
2/2 [==============================] - 0s 70ms/step
2/2 [==============================] - 0s 32ms/step
2/2 [==============================] - 0s 48ms/step
Epoch 27, Step 0, D-Loss: 1.5389150977134705, D-Acc: 49.21875, G-Loss: 0.08688881248235703
2/2 [==============================] - 0s 50ms/step
2/2 [==============================] - 0s 59ms/step
2/2 [==============================] - 0s 34ms/step
Epoch 28, Step 0, D-Loss: 1.556357204914093, D-Acc: 48.4375, G-Loss: 0.08841417729854584
2/2 [==============================] - 0s 43ms/step
2/2 [==============================] - 0s 45ms/step
2/2 [==============================] - 0s 52ms/step
Epoch 29, Step 0, D-Loss: 1.5415984690189362, D-Acc: 47.65625, G-Loss: 0.09251536428928375
2/2 [==============================] - 0s 31ms/step
```

```
2/2 [==============================] - 0s 61ms/step
2/2 [==============================] - 0s 32ms/step
Epoch 30, Step 0, D-Loss: 1.543577253818512, D-Acc: 46.09375, G-Loss: 0.08652600646018982
2/2 [==============================] - 0s 39ms/step
2/2 [==============================] - 0s 59ms/step
1/1 [==============================] - 0s 73ms/step
2/2 [==============================] - 0s 43ms/step
Epoch 31, Step 0, D-Loss: 1.54795041680336, D-Acc: 47.65625, G-Loss: 0.09020714461803436
2/2 [==============================] - 0s 48ms/step
2/2 [==============================] - 0s 44ms/step
2/2 [==============================] - 0s 55ms/step
Epoch 32, Step 0, D-Loss: 1.5288953185081482, D-Acc: 47.65625, G-Loss: 0.09089905023574829
2/2 [==============================] - 0s 45ms/step
2/2 [==============================] - 0s 47ms/step
2/2 [==============================] - 0s 50ms/step
Epoch 33, Step 0, D-Loss: 1.545252799987793, D-Acc: 46.09375, G-Loss: 0.08806487917900085
2/2 [==============================] - 0s 49ms/step
2/2 [==============================] - 0s 54ms/step
2/2 [==============================] - 0s 55ms/step
Epoch 34, Step 0, D-Loss: 1.5501168370246887, D-Acc: 46.875, G-Loss: 0.0888870358467102
2/2 [==============================] - 0s 47ms/step
2/2 [==============================] - 0s 54ms/step
2/2 [==============================] - 0s 49ms/step
Epoch 35, Step 0, D-Loss: 1.5607050061225891, D-Acc: 42.96875, G-Loss: 0.0864979475736618
2/2 [==============================] - 0s 50ms/step
2/2 [==============================] - 0s 50ms/step
2/2 [==============================] - 0s 53ms/step
Epoch 36, Step 0, D-Loss: 1.5813909769058228, D-Acc: 45.3125, G-Loss: 0.08336884528398514
2/2 [==============================] - 0s 42ms/step
2/2 [==============================] - 0s 47ms/step
2/2 [==============================] - 0s 43ms/step
Epoch 37, Step 0, D-Loss: 1.5748992562294006, D-Acc: 47.65625, G-Loss: 0.08490622788667679
2/2 [==============================] - 0s 41ms/step
2/2 [==============================] - 0s 35ms/step
2/2 [==============================] - 0s 47ms/step
Epoch 38, Step 0, D-Loss: 1.5898439586162567, D-Acc: 47.65625, G-Loss: 0.08591754734516144
2/2 [==============================] - 0s 35ms/step
2/2 [==============================] - 0s 49ms/step
2/2 [==============================] - 0s 39ms/step
Epoch 39, Step 0, D-Loss: 1.5639400482177734, D-Acc: 46.875, G-Loss: 0.08568787574768066
2/2 [==============================] - 0s 45ms/step
2/2 [==============================] - 0s 53ms/step
2/2 [==============================] - 0s 49ms/step
Epoch 40, Step 0, D-Loss: 1.5606472492218018, D-Acc: 46.875, G-Loss: 0.08307512104511261
2/2 [==============================] - 0s 45ms/step
2/2 [==============================] - 0s 49ms/step
1/1 [==============================] - 0s 59ms/step
2/2 [==============================] - 0s 46ms/step
Epoch 41, Step 0, D-Loss: 1.585960566997528, D-Acc: 45.3125, G-Loss: 0.08248283714056015
2/2 [==============================] - 0s 38ms/step
2/2 [==============================] - 0s 40ms/step
2/2 [==============================] - 0s 40ms/step
Epoch 42, Step 0, D-Loss: 1.5780255198478699, D-Acc: 46.09375, G-Loss: 0.08523035794496536
2/2 [==============================] - 0s 50ms/step
2/2 [==============================] - 0s 39ms/step
2/2 [==============================] - 0s 43ms/step
Epoch 43, Step 0, D-Loss: 1.5831877291202545, D-Acc: 44.53125, G-Loss: 0.08670960366725922
2/2 [==============================] - 0s 49ms/step
2/2 [==============================] - 0s 55ms/step
2/2 [==============================] - 0s 38ms/step
Epoch 44, Step 0, D-Loss: 1.5836209058761597, D-Acc: 48.4375, G-Loss: 0.08174121379852295
2/2 [==============================] - 0s 46ms/step
2/2 [==============================] - 0s 42ms/step
2/2 [==============================] - 0s 55ms/step
Epoch 45, Step 0, D-Loss: 1.5696801543235779, D-Acc: 47.65625, G-Loss: 0.08194457739591599
2/2 [==============================] - 0s 55ms/step
```

```
2/2 [==============================] - 0s 38ms/step
2/2 [==============================] - 0s 50ms/step
Epoch 46, Step 0, D-Loss: 1.5984586477279663, D-Acc: 43.75, G-Loss: 0.0832623764872551
2/2 [==============================] - 0s 38ms/step
2/2 [==============================] - 0s 32ms/step
2/2 [==============================] - 0s 35ms/step
Epoch 47, Step 0, D-Loss: 1.5853493809700012, D-Acc: 46.875, G-Loss: 0.0820540338754654
2/2 [==============================] - 0s 41ms/step
2/2 [==============================] - 0s 40ms/step
2/2 [==============================] - 0s 47ms/step
Epoch 48, Step 0, D-Loss: 1.5825019478797913, D-Acc: 47.65625, G-Loss: 0.07994864881038666
2/2 [==============================] - 0s 33ms/step
2/2 [==============================] - 0s 51ms/step
2/2 [==============================] - 0s 42ms/step
Epoch 49, Step 0, D-Loss: 1.5919575691223145, D-Acc: 44.53125, G-Loss: 0.08076215535402298
2/2 [==============================] - 0s 36ms/step
2/2 [==============================] - 0s 39ms/step
2/2 [==============================] - 0s 39ms/step
Epoch 50, Step 0, D-Loss: 1.5952409505844116, D-Acc: 49.21875, G-Loss: 0.08289815485477448
2/2 [==============================] - 0s 57ms/step
2/2 [==============================] - 0s 43ms/step
1/1 [==============================] - 0s 61ms/step
2/2 [==============================] - 0s 53ms/step
Epoch 51, Step 0, D-Loss: 1.5955896973609924, D-Acc: 46.875, G-Loss: 0.07652326673269272
2/2 [==============================] - 0s 38ms/step
2/2 [==============================] - 0s 51ms/step
2/2 [==============================] - 0s 50ms/step
Epoch 52, Step 0, D-Loss: 1.6081750988960266, D-Acc: 46.875, G-Loss: 0.07906067371368408
2/2 [==============================] - 0s 41ms/step
2/2 [==============================] - 0s 55ms/step
2/2 [==============================] - 0s 39ms/step
Epoch 53, Step 0, D-Loss: 1.6118325889110565, D-Acc: 45.3125, G-Loss: 0.08084765076637268
2/2 [==============================] - 0s 53ms/step
2/2 [==============================] - 0s 53ms/step
2/2 [==============================] - 0s 30ms/step
Epoch 54, Step 0, D-Loss: 1.5975553393363953, D-Acc: 46.875, G-Loss: 0.08242578059434891
2/2 [==============================] - 0s 37ms/step
2/2 [==============================] - 0s 48ms/step
2/2 [==============================] - 0s 63ms/step
Epoch 55, Step 0, D-Loss: 1.5920605659484863, D-Acc: 44.53125, G-Loss: 0.08209642022848129
2/2 [==============================] - 0s 53ms/step
2/2 [==============================] - 0s 48ms/step
2/2 [==============================] - 0s 37ms/step
Epoch 56, Step 0, D-Loss: 1.6162050366401672, D-Acc: 46.875, G-Loss: 0.07638239860534668
2/2 [==============================] - 0s 39ms/step
2/2 [==============================] - 0s 55ms/step
2/2 [==============================] - 0s 34ms/step
Epoch 57, Step 0, D-Loss: 1.607642412185669, D-Acc: 44.53125, G-Loss: 0.07525159418582916
2/2 [==============================] - 0s 58ms/step
2/2 [==============================] - 0s 42ms/step
2/2 [==============================] - 0s 42ms/step
Epoch 58, Step 0, D-Loss: 1.6085178852081299, D-Acc: 42.96875, G-Loss: 0.07484909147024155
2/2 [==============================] - 0s 55ms/step
2/2 [==============================] - 0s 40ms/step
2/2 [==============================] - 0s 55ms/step
Epoch 59, Step 0, D-Loss: 1.6176687479019165, D-Acc: 48.4375, G-Loss: 0.0763208419084549
2/2 [==============================] - 0s 31ms/step
2/2 [==============================] - 0s 44ms/step
2/2 [==============================] - 0s 47ms/step
Epoch 60, Step 0, D-Loss: 1.6188184320926666, D-Acc: 46.875, G-Loss: 0.0791281908750534
2/2 [==============================] - 0s 63ms/step
2/2 [==============================] - 0s 25ms/step
1/1 [==============================] - 0s 47ms/step
2/2 [==============================] - 0s 39ms/step
Epoch 61, Step 0, D-Loss: 1.6194016337394714, D-Acc: 46.875, G-Loss: 0.0789446160197258
2/2 [==============================] - 0s 36ms/step
```

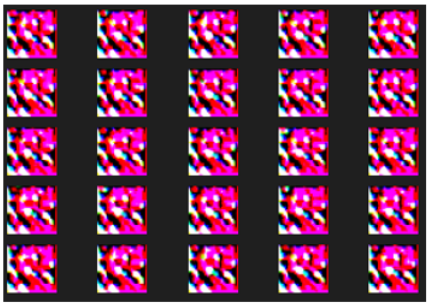
```
2/2 [==============================] - 0s 47ms/step
2/2 [==============================] - 0s 45ms/step
Epoch 62, Step 0, D-Loss: 1.6301100850105286, D-Acc: 45.3125, G-Loss: 0.0741746723651886
2/2 [==============================] - 0s 36ms/step
2/2 [==============================] - 0s 73ms/step
2/2 [==============================] - 0s 36ms/step
Epoch 63, Step 0, D-Loss: 1.6291669011116028, D-Acc: 45.3125, G-Loss: 0.07468157261610031
2/2 [==============================] - 0s 37ms/step
2/2 [==============================] - 0s 63ms/step
2/2 [==============================] - 0s 39ms/step
Epoch 64, Step 0, D-Loss: 1.6287148296833038, D-Acc: 46.875, G-Loss: 0.07735884934663773
2/2 [==============================] - 0s 55ms/step
2/2 [==============================] - 0s 35ms/step
2/2 [==============================] - 0s 59ms/step
Epoch 65, Step 0, D-Loss: 1.6308998465538025, D-Acc: 46.875, G-Loss: 0.07221978902816772
2/2 [==============================] - 0s 37ms/step
2/2 [==============================] - 0s 49ms/step
2/2 [==============================] - 0s 53ms/step
Epoch 66, Step 0, D-Loss: 1.6498736143112183, D-Acc: 44.53125, G-Loss: 0.070715993642807
2/2 [==============================] - 0s 44ms/step
2/2 [==============================] - 0s 49ms/step
2/2 [==============================] - 0s 51ms/step
Epoch 67, Step 0, D-Loss: 1.6352354288101196, D-Acc: 46.09375, G-Loss: 0.07180492579936981
2/2 [==============================] - 0s 58ms/step
2/2 [==============================] - 0s 32ms/step
2/2 [==============================] - 0s 47ms/step
Epoch 68, Step 0, D-Loss: 1.6578554511070251, D-Acc: 45.3125, G-Loss: 0.07664079964160919
2/2 [==============================] - 0s 38ms/step
2/2 [==============================] - 0s 38ms/step
2/2 [==============================] - 0s 47ms/step
Epoch 69, Step 0, D-Loss: 1.6481069922447205, D-Acc: 44.53125, G-Loss: 0.0745805948972702
2/2 [==============================] - 0s 33ms/step
2/2 [==============================] - 0s 45ms/step
2/2 [==============================] - 0s 36ms/step
Epoch 70, Step 0, D-Loss: 1.6473522186279297, D-Acc: 48.4375, G-Loss: 0.07246960699558258
2/2 [==============================] - 0s 57ms/step
2/2 [==============================] - 0s 41ms/step
1/1 [==============================] - 0s 49ms/step
2/2 [==============================] - 0s 47ms/step
Epoch 71, Step 0, D-Loss: 1.6353341042995453, D-Acc: 46.875, G-Loss: 0.07464936375617981
2/2 [==============================] - 0s 47ms/step
2/2 [==============================] - 0s 40ms/step
2/2 [==============================] - 0s 32ms/step
Epoch 72, Step 0, D-Loss: 1.6666560769081116, D-Acc: 47.65625, G-Loss: 0.07353644073009491
2/2 [==============================] - 0s 63ms/step
2/2 [==============================] - 0s 27ms/step
2/2 [==============================] - 0s 40ms/step
Epoch 73, Step 0, D-Loss: 1.648057758808136, D-Acc: 48.4375, G-Loss: 0.07171425968408585
2/2 [==============================] - 0s 59ms/step
2/2 [==============================] - 0s 49ms/step
2/2 [==============================] - 0s 45ms/step
Epoch 74, Step 0, D-Loss: 1.640342652797699, D-Acc: 47.65625, G-Loss: 0.07314617931842804
2/2 [==============================] - 0s 57ms/step
2/2 [==============================] - 0s 39ms/step
2/2 [==============================] - 0s 49ms/step
Epoch 75, Step 0, D-Loss: 1.6570880115032196, D-Acc: 45.3125, G-Loss: 0.07579794526100159
2/2 [==============================] - 0s 42ms/step
2/2 [==============================] - 0s 34ms/step
2/2 [==============================] - 0s 59ms/step
Epoch 76, Step 0, D-Loss: 1.6564058661460876, D-Acc: 47.65625, G-Loss: 0.07145845890045166
2/2 [==============================] - 0s 39ms/step
2/2 [==============================] - 0s 30ms/step
2/2 [==============================] - 0s 46ms/step
Epoch 77, Step 0, D-Loss: 1.668128788471222, D-Acc: 47.65625, G-Loss: 0.06986162066459656
2/2 [==============================] - 0s 43ms/step
2/2 [==============================] - 0s 64ms/step
```

```
2/2 [==============================] - 0s 23ms/step
Epoch 78, Step 0, D-Loss: 1.6807662844657898, D-Acc: 45.3125, G-Loss: 0.0703335627913475
2/2 [==============================] - 0s 58ms/step
2/2 [==============================] - 0s 36ms/step
2/2 [==============================] - 0s 64ms/step
Epoch 79, Step 0, D-Loss: 1.6650253534317017, D-Acc: 47.65625, G-Loss: 0.06960901618003845
2/2 [==============================] - 0s 43ms/step
2/2 [==============================] - 0s 44ms/step
2/2 [==============================] - 0s 48ms/step
Epoch 80, Step 0, D-Loss: 1.6608455181121826, D-Acc: 46.875, G-Loss: 0.0689220130443573
2/2 [==============================] - 0s 63ms/step
2/2 [==============================] - 0s 38ms/step
1/1 [==============================] - 0s 59ms/step
2/2 [==============================] - 0s 47ms/step
Epoch 81, Step 0, D-Loss: 1.651425838470459, D-Acc: 46.875, G-Loss: 0.07155314087867737
2/2 [==============================] - 0s 43ms/step
2/2 [==============================] - 0s 37ms/step
2/2 [==============================] - 0s 41ms/step
Epoch 82, Step 0, D-Loss: 1.672002911567688, D-Acc: 48.4375, G-Loss: 0.06854095309972763
2/2 [==============================] - 0s 63ms/step
2/2 [==============================] - 0s 32ms/step
2/2 [==============================] - 0s 40ms/step
Epoch 83, Step 0, D-Loss: 1.6837009191513062, D-Acc: 46.875, G-Loss: 0.07091417908668518
2/2 [==============================] - 0s 48ms/step
2/2 [==============================] - 0s 38ms/step
2/2 [==============================] - 0s 40ms/step
Epoch 84, Step 0, D-Loss: 1.6769118010997772, D-Acc: 43.75, G-Loss: 0.06949591636657715
2/2 [==============================] - 0s 36ms/step
2/2 [==============================] - 0s 49ms/step
2/2 [==============================] - 0s 31ms/step
Epoch 85, Step 0, D-Loss: 1.6793779134750366, D-Acc: 46.875, G-Loss: 0.06719396263360977
2/2 [==============================] - 0s 47ms/step
2/2 [==============================] - 0s 28ms/step
2/2 [==============================] - 0s 57ms/step
Epoch 86, Step 0, D-Loss: 1.6664071679115295, D-Acc: 48.4375, G-Loss: 0.07121209055185318
2/2 [==============================] - 0s 37ms/step
2/2 [==============================] - 0s 41ms/step
2/2 [==============================] - 0s 37ms/step
Epoch 87, Step 0, D-Loss: 1.6702519357204437, D-Acc: 43.75, G-Loss: 0.06960982829332352
2/2 [==============================] - 0s 50ms/step
2/2 [==============================] - 0s 39ms/step
2/2 [==============================] - 0s 57ms/step
Epoch 88, Step 0, D-Loss: 1.69154292345047, D-Acc: 46.875, G-Loss: 0.06744537502527237
2/2 [==============================] - 0s 49ms/step
2/2 [==============================] - 0s 43ms/step
2/2 [==============================] - 0s 36ms/step
Epoch 89, Step 0, D-Loss: 1.684645414352417, D-Acc: 46.09375, G-Loss: 0.06648287177085876
2/2 [==============================] - 0s 57ms/step
2/2 [==============================] - 0s 53ms/step
2/2 [==============================] - 0s 44ms/step
Epoch 90, Step 0, D-Loss: 1.6848950386047363, D-Acc: 46.09375, G-Loss: 0.0642966628074646
2/2 [==============================] - 0s 47ms/step
2/2 [==============================] - 0s 49ms/step
1/1 [==============================] - 0s 51ms/step
2/2 [==============================] - 0s 47ms/step
Epoch 91, Step 0, D-Loss: 1.6722689867019653, D-Acc: 47.65625, G-Loss: 0.06848204880952835
2/2 [==============================] - 0s 39ms/step
2/2 [==============================] - 0s 36ms/step
2/2 [==============================] - 0s 47ms/step
Epoch 92, Step 0, D-Loss: 1.6989425718784332, D-Acc: 46.09375, G-Loss: 0.0647907480597496
2/2 [==============================] - 0s 43ms/step
2/2 [==============================] - 0s 56ms/step
2/2 [==============================] - 0s 48ms/step
Epoch 93, Step 0, D-Loss: 1.7077288925647736, D-Acc: 46.09375, G-Loss: 0.07219605147838593
2/2 [==============================] - 0s 47ms/step
2/2 [==============================] - 0s 33ms/step
```

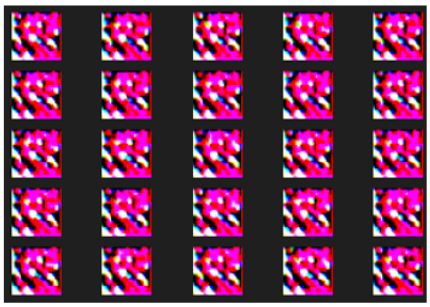
```
2/2 [==============================] - 0s 63ms/step
Epoch 94, Step 0, D-Loss: 1.6963713467121124, D-Acc: 48.4375, G-Loss: 0.06623832881450653
2/2 [==============================] - 0s 32ms/step
2/2 [==============================] - 0s 63ms/step
2/2 [==============================] - 0s 41ms/step
Epoch 95, Step 0, D-Loss: 1.675630658864975, D-Acc: 46.09375, G-Loss: 0.0667354017496109
2/2 [==============================] - 0s 37ms/step
2/2 [==============================] - 0s 41ms/step
2/2 [==============================] - 0s 47ms/step
Epoch 96, Step 0, D-Loss: 1.6762495040893555, D-Acc: 46.875, G-Loss: 0.06800433993339539
2/2 [==============================] - 0s 38ms/step
2/2 [==============================] - 0s 43ms/step
2/2 [==============================] - 0s 49ms/step
Epoch 97, Step 0, D-Loss: 1.6995777487754822, D-Acc: 46.875, G-Loss: 0.06468817591667175
2/2 [==============================] - 0s 52ms/step
2/2 [==============================] - 0s 40ms/step
2/2 [==============================] - 0s 63ms/step
Epoch 98, Step 0, D-Loss: 1.6887358129024506, D-Acc: 46.09375, G-Loss: 0.06443732231855392
2/2 [==============================] - 0s 40ms/step
2/2 [==============================] - 0s 63ms/step
2/2 [==============================] - 0s 42ms/step
Epoch 99, Step 0, D-Loss: 1.6910340189933777, D-Acc: 48.4375, G-Loss: 0.06537196040153503
2/2 [==============================] - 0s 53ms/step
2/2 [==============================] - 0s 36ms/step
WARNING:tensorflow:Compiled the loaded model, but the compiled metrics have yet to be buil
t. `model.compile_metrics` will be empty until you train or evaluate the model.
```

epoch_0

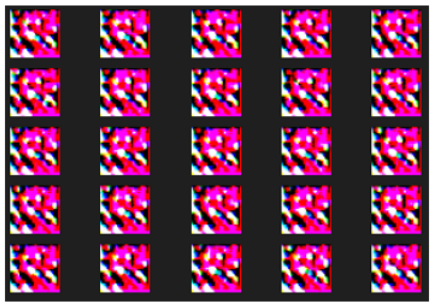
epoch_10



epoch_20

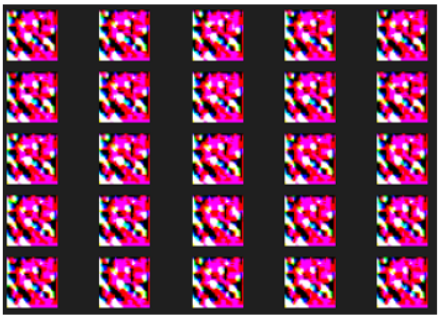


epoch_30

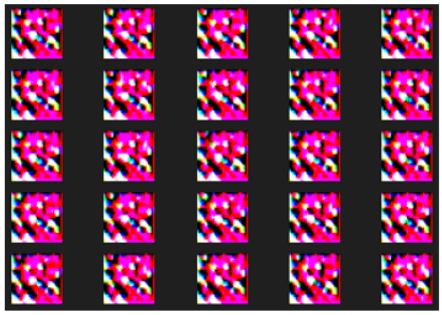


epoch_40

epoch_50

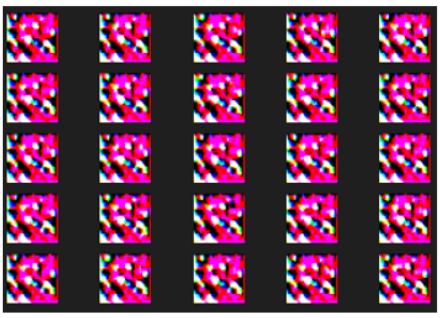


epoch_60



epoch_70

epoch_80



epoch_90

In [59]:
```
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping
```

Evaluation Of Generated Architechture Using CNN

The build_classification_model function takes two arguments: **input_shape**, provied during the building of Generator & Discriminator, and num_classes, which is the number of classes in the classification problem.\ This code is used in conjunction with the generator network to evaluate the quality of the generated images.

The model consists of several layers, as desribed below.

1. **Convolutional layer:** The first layer in our model is a 2D convolutional layer with 32 filters and a 3x3 kernel size. This layer applies a set of filters to the input image to extract features, which are then used in subsequent layers to classify the image.
2. **Max pooling layer:** The second layer in our model is a 2D max pooling layer with a 2x2 pool size. This layer downsamples the feature maps outputted by the previous convolutional layer, reducing their size.
3. **Convolutional layer:** The third layer in our model is another 2D convolutional layer with 64 filters and a 3x3 kernel size. This layer further extracts features from the output of the previous layer.
4. **Max pooling layer:** The fourth layer in our model is another 2D max pooling layer with a 2x2 pool size. This layer further downsamples the feature maps outputted by the previous convolutional layer.
5. **Dropout layer:** The fifth layer in our model is a dropout layer with a rate of 0.25. This layer randomly sets 25% of the output values of the previous layer to 0 during training to reduce overfitting.
6. **Flattening layer:** The sixth layer in our model is a flattening layer, which flattens the output of the previous layer into a 1D array.
7. **Dense layer:** The seventh layer in our model is a dense layer with 128 units and a ReLU activation function. This layer applies a set of weights to the flattened output of the previous layer to produce a set of activations.
8. **Dropout layer:** The eighth layer in our model is another dropout layer with a rate of 0.5. This layer further reduces overfitting.

9. **Dense layer:** The ninth and final layer in our model is a dense layer with num_classes units and a softmax activation function. This layer produces a probability distribution over the possible classes based on the activations outputted by the previous layer.

In [60]:
```
def build_classification_model(input_shape, num_classes):
    model = tf.keras.Sequential([
        layers.Conv2D(32, kernel_size=3, activation="relu", input_shape=input_shape),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(64, kernel_size=3, activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Dropout(0.25),
        layers.Flatten(),
        layers.Dense(128, activation="relu"),
        layers.Dropout(0.5),
        layers.Dense(num_classes, activation="softmax")
    ])
    return model
```

In [61]:
```
num_classes = np.unique(y_train).shape[0]
classification_model = build_classification_model(input_shape, num_classes)
classification_model.compile(loss="binary_crossentropy", optimizer="adam", metrics=["accu
```

The training process for the traffic sign classification model is being set up and executed with the help of callbacks. **Callbacks** in TensorFlow Keras are functions that can be applied at specific stages of the training process, such as at the start or end of an epoch, batch, or training session. They track the model's performance, saving the model, or even adjusting the learning rate during training.

ModelCheckpoint: This callback saves the model at specific intervals during training. In this code below,, it is configured to save the model with the best validation accuracy achieved so far.\ The saved model file is named **"traffic_sign_classification.h5"**.\ **The save_best_only=True** argument ensures that only the best model, in terms of validation accuracy, is saved.\ **verbose=1** provides updates when a new best model is saved.

EarlyStopping: This callback stops the training process if the model's performance on the validation set does not improve after a specified number of consecutive epochs.\ In this code, the **patience** parameter is set to **10**, If the validation accuracy does not improve for 10 consecutive epochs, the training process will be stopped. \ The **verbose=1** argument provides updates when the training is stopped due to early stopping.

The classification_model.fit() function is called to train the model on the training data (x_train, y_train) for 100 epochs with a batch size specified. \ **The validation_split=0.2** This argument indicates that 20% of the training data will be used for validation during training.

The callbacks, model_checkpoint and early_stopping, are passed in a list to the **callbacks** parameter.

In [62]:
```
model_checkpoint = ModelCheckpoint("traffic_sign_classification_set.h5", save_best_only=Tr
early_stopping = EarlyStopping(patience=10, verbose=1)

history = classification_model.fit(x_train, y_train, batch_size=batch_size, epochs=10, val
```

```
Epoch 1/10
2/3 [===================>.....] - ETA: 0s - loss: 0.3939 - accuracy: 0.0000e+00
Epoch 1: val_loss improved from inf to 0.00291, saving model to traffic_sign_classificatio
n_set.h5
3/3 [==============================] - 1s 168ms/step - loss: 0.3093 - accuracy: 0.0000e+00
- val_loss: 0.0029 - val_accuracy: 0.0000e+00
Epoch 2/10
3/3 [==============================] - ETA: 0s - loss: 0.0035 - accuracy: 0.0000e+00
Epoch 2: val_loss improved from 0.00291 to 0.00000, saving model to traffic_sign_classific
```

```
ation_set.h5
3/3 [==============================] - 0s 92ms/step - loss: 0.0035 - accuracy: 0.0000e+00
- val_loss: 4.0581e-06 - val_accuracy: 0.0000e+00
Epoch 3/10
3/3 [==============================] - ETA: 0s - loss: 7.9142e-05 - accuracy: 0.0000e+00
Epoch 3: val_loss improved from 0.00000 to 0.00000, saving model to traffic_sign_classific
ation_set.h5
3/3 [==============================] - 0s 87ms/step - loss: 7.9142e-05 - accuracy: 0.0000e
+00 - val_loss: 1.3002e-08 - val_accuracy: 0.0000e+00
Epoch 4/10
3/3 [==============================] - ETA: 0s - loss: 1.5754e-06 - accuracy: 0.0000e+00
Epoch 4: val_loss improved from 0.00000 to 0.00000, saving model to traffic_sign_classific
ation_set.h5
3/3 [==============================] - 0s 86ms/step - loss: 1.5754e-06 - accuracy: 0.0000e
+00 - val_loss: 1.2963e-10 - val_accuracy: 0.0000e+00
Epoch 5/10
3/3 [==============================] - ETA: 0s - loss: 1.9688e-07 - accuracy: 0.0000e+00
Epoch 5: val_loss improved from 0.00000 to 0.00000, saving model to traffic_sign_classific
ation_set.h5
3/3 [==============================] - 0s 72ms/step - loss: 1.9688e-07 - accuracy: 0.0000e
+00 - val_loss: 3.3321e-12 - val_accuracy: 0.0000e+00
Epoch 6/10
3/3 [==============================] - ETA: 0s - loss: 2.6543e-08 - accuracy: 0.0000e+00
Epoch 6: val_loss improved from 0.00000 to 0.00000, saving model to traffic_sign_classific
ation_set.h5
3/3 [==============================] - 0s 69ms/step - loss: 2.6543e-08 - accuracy: 0.0000e
+00 - val_loss: 1.8131e-13 - val_accuracy: 0.0000e+00
Epoch 7/10
3/3 [==============================] - ETA: 0s - loss: 2.7596e-09 - accuracy: 0.0000e+00
Epoch 7: val_loss improved from 0.00000 to 0.00000, saving model to traffic_sign_classific
ation_set.h5
3/3 [==============================] - 0s 76ms/step - loss: 2.7596e-09 - accuracy: 0.0000e
+00 - val_loss: 1.8156e-14 - val_accuracy: 0.0000e+00
Epoch 8/10
3/3 [==============================] - ETA: 0s - loss: 9.7435e-10 - accuracy: 0.0000e+00
Epoch 8: val_loss improved from 0.00000 to 0.00000, saving model to traffic_sign_classific
ation_set.h5
3/3 [==============================] - 0s 72ms/step - loss: 9.7435e-10 - accuracy: 0.0000e
+00 - val_loss: 2.9845e-15 - val_accuracy: 0.0000e+00
Epoch 9/10
3/3 [==============================] - ETA: 0s - loss: 1.6193e-10 - accuracy: 0.0000e+00
Epoch 9: val_loss improved from 0.00000 to 0.00000, saving model to traffic_sign_classific
ation_set.h5
3/3 [==============================] - 0s 73ms/step - loss: 1.6193e-10 - accuracy: 0.0000e
+00 - val_loss: 7.3126e-16 - val_accuracy: 0.0000e+00
Epoch 10/10
3/3 [==============================] - ETA: 0s - loss: 4.4442e-11 - accuracy: 0.0000e+00
Epoch 10: val_loss improved from 0.00000 to 0.00000, saving model to traffic_sign_classifi
cation_set.h5
3/3 [==============================] - 0s 71ms/step - loss: 4.4442e-11 - accuracy: 0.0000e
+00 - val_loss: 2.4634e-16 - val_accuracy: 0.0000e+00
```

To print the test loss and test accuracy of the trained model

In [63]:

```
# y_test = np.argmax(y_test, axis=1)
test_loss, test_accuracy = classification_model.evaluate(x_test, y_test)
print(f"Test loss: {test_loss}, Test accuracy: {test_accuracy}")

# test_loss, test_accuracy = classification_model.evaluate(x_test, y_test)
# print(f"Test loss: {test_loss}, Test accuracy: {test_accuracy}")
```

```
395/395 [==============================] - 3s 6ms/step - loss: 712.3958 - accuracy: 0.0570
Test loss: 712.395751953125, Test accuracy: 0.05700712651014328
```

To generate images using a pre-trained DCGAN generator model

Load the pre-trained generator model: The generator model is loaded using tf.keras.models.load_model() and stored in the generator variable.

Set the number of images to generate.

Set the latent dimension

Generate random noise as input to the generator: Random noise is generated using NumPy's np.random.normal() function, with a mean of 0 and a standard deviation of 1. The shape of the noise tensor is (num_images, latent_dim).

Generate images using the generator: The generated images are obtained by passing the noise tensor through the generator model using the predict() method.

Rescale the images: The images are rescaled from the range of [-1, 1] to [0, 1] by multiplying by 0.5 and adding 0.5.

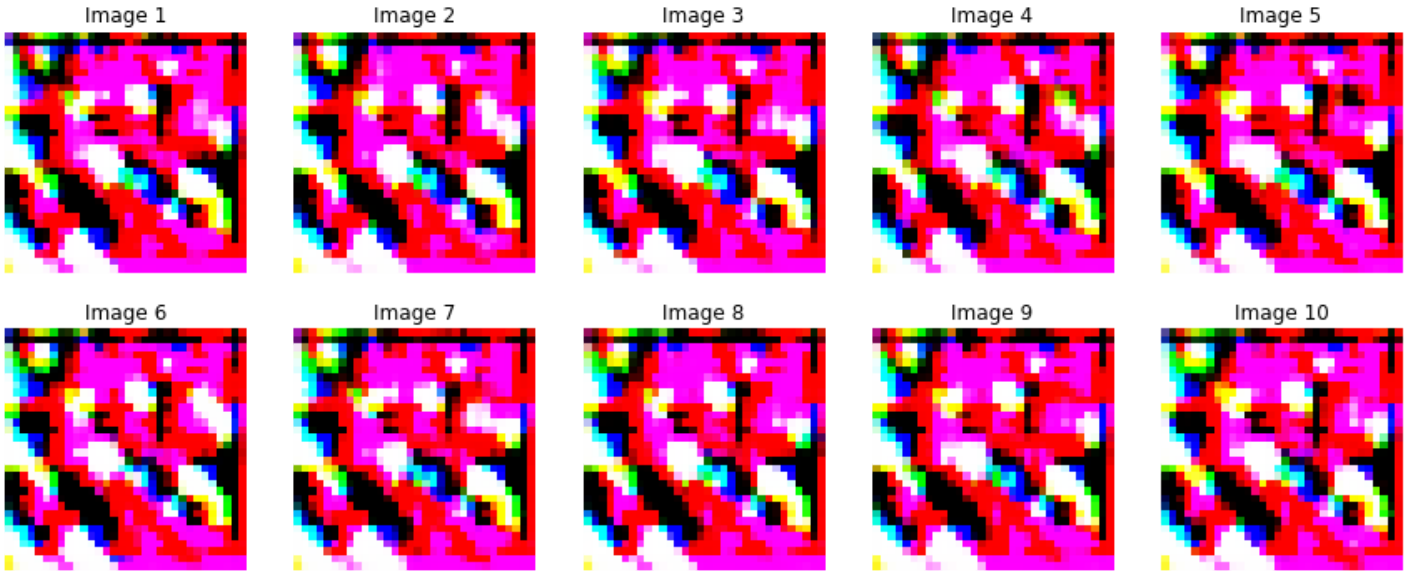
Plot and display the generated images: The generated images are plotted and displayed using Matplotlib. The plt.subplots() function is set to create a 2x5 grid of subplots, The generated images are displayed using the imshow() method.

The **plt.show()** function displays the generated images.

In [64]:

```
# Load the trained generator model
generator = tf.keras.models.load_model("generator_final_set.h5")

#Set the number of images you want to generate
num_images = 50

# Set the latent dimension (same as used during training)
latent_dim = 100

# Generate random noise as input to the generator
noise = np.random.normal(0, 1, (num_images, latent_dim))

# Generate images using the generator
generated_images = generator.predict(noise)

# Rescale the images from the range [-1, 1] to [0, 1]
generated_images = (generated_images * 0.5) + 0.5

# Plot and display the generated images
fig, axes = plt.subplots(2, 5, figsize=(15, 6))

for i, ax in enumerate(axes.flat):
    ax.imshow(generated_images[i])
    ax.axis("off")
    ax.set_title(f"Image {i+1}")

plt.show()
```

```
WARNING:tensorflow:No training configuration found in the save file, so the model was *not
* compiled. Compile it manually.
2/2 [==============================] - 0s 37ms/step
```



In [65]:

```
from tensorflow.keras.models import load_model
```

Validating Traffic Signs using a Trained Discriminator Model

The code uses a trained discriminator to classify an input image as either a real or fake traffic sign. It pre-processes the input image by resizing, normalizing, and expanding its dimensions.

The discriminator predicts whether the input image is a valid traffic sign or not.

If the image is classified as a real traffic sign, it prints **"The test image is classified as: Traffic sign"**.

If it is classified as a fake traffic sign, it prints **"The test image is classified as: Non-traffic sign"**.

For The following Real Traffic Sign that was given as input,

The code successfully detects a real traffic sign and produced the output\ **"The test image is classified as: Traffic sign"**

The function **classify_traffic_sign()** takes an image path and a pre-trained discriminator model as its inputs.

the Image.open() method Loads the image. **The resize()** method resizes the image to 32x32 pixels. Normalize the image by dividing its pixel values by 127.5 and subtracting 1. **NumPy's np.expand_dims()** Expands the dimensions of the image to match the input format of the discriminator model. **The predict() method** Predicts the class of the image using of the discriminator model.

The classify_traffic_sign preprocesses the image, feeds it into the discriminator, and returns the following classification result, **"Traffic sign"** if the discriminator predicts it to be a traffic sign, or **"Non-traffic sign"** if it predicts it to be something else.

The image path used to test is **"C:/Users/beneg/Downloads/GTSDB/Test/test12611.jpg"**

In [71]:
```
def classify_traffic_sign(image_path, discriminator):
    # Preprocess the image (resize, normalize, and expand dimensions)
    image = Image.open(image_path)
    resized_image = image.resize((32, 32))
    normalized_image = (np.array(resized_image) / 127.5) - 1
    expanded_image = np.expand_dims(normalized_image, axis=0)

    # Use the discriminator to predict the class
    prediction = discriminator.predict(expanded_image)
    return "Traffic sign" if prediction > 0.5 else "Non-traffic sign"

# Load the saved discriminator model
discriminator_saved_path = "discriminator.h5"
loaded_discriminator = load_model(discriminator_saved_path)

test_image_path = "C:/Users/beneg/Downloads/GTSDB/Test/test12611.jpg"
classification = classify_traffic_sign(test_image_path, loaded_discriminator)
print(f"The test image is classified as: {classification}")
```

```
1/1 [==============================] - 0s 74ms/step
The test image is classified as: Traffic sign
```

Evaluation of the Results

Comparing GAN approach with traditional techniques

Accuracy

GANs can help train recognition algorithms to manage differences in lighting, weather, and traffic sign types by creating diverse and realistic traffic sign images.

Traditional Techniques

Because of their reliance on manual feature extraction and sensitivity to variations in lighting and weather conditions, these approaches may have lesser accuracy in complicated circumstances.

Processing Time

GAN-based recognition systems may necessitate substantial computer resources for training, resulting in prolonged processing times. However, depending on the specific implementation, their inference time may be equivalent to or faster than traditional techniques once trained.\ Traditional Techniques: These approaches have faster training times but can be slower in inference, particularly when dealing with complex feature extraction and matching tasks.

Robustness

GAN-based recognition systems are more robust in general because they can learn features from raw visual data and adapt to fluctuations in lighting, weather, and traffic sign types.\ Traditional Techniques: Traditional approaches have a limited robustness since they frequently rely on handcrafted features and may struggle to adjust to changes in environmental conditions or different traffic sign kinds.

Real-World Testing & Validation

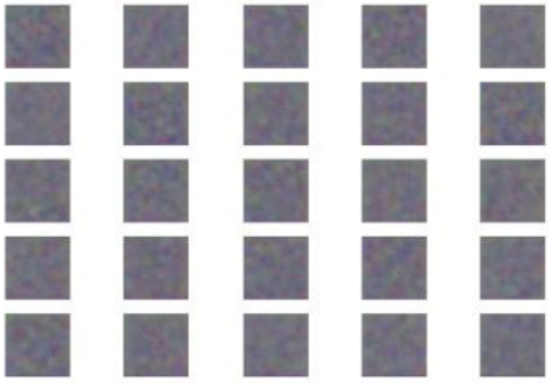
Simulation Results

In this section, we present a visual comparison of the generated images during the training process of our DCGAN model. This was accomplished by utilising both the adjusted GTSDB dataset and the complete GTSDB dataset.

It was observed that training the model on a smaller subset of the dataset going through 100 epochs, consisting of only 600 files, resulted in generated images that were more easily distinguishable by the naked eye. This indicates that the model was able to capture the essential characteristics of the traffic signs more effectively.

We decided on using a modified dataset to train the model on. Simultaneously, we were training the model on the GTSDB dataset of 39000+ files on another system to ensure the success of our code. The differences between the outputs of the two datasets are interesting enough to warrant a separate study on how the code functions. It took approximately an hour for the modified dataset compared to the 35+ plus hours for the whole dataset.

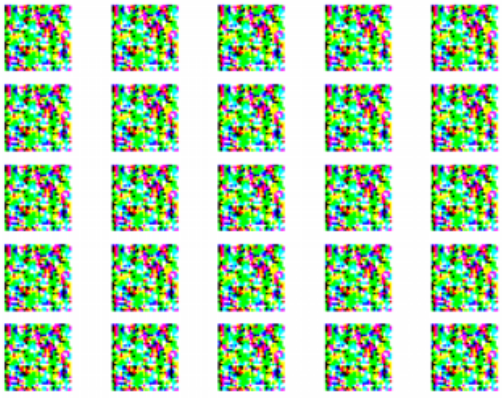
Here, we showcase the progression of the generated images from epoch_0 to epoch_90 using the modified dataset. These images demonstrate how the model's understanding of traffic sign features and structures improved as the training progressed. The images show a gradual improvement in clarity and sharpness.



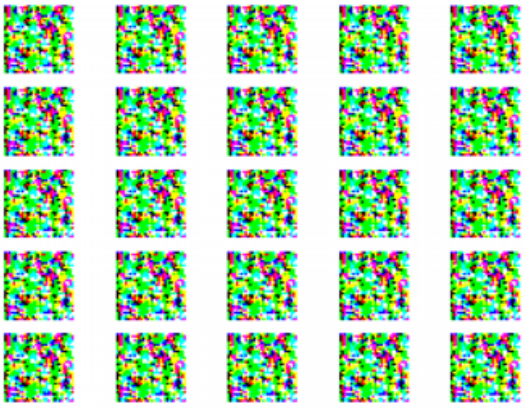
epoch_0

epoch_90

These are the images generated from GTSDB dataset consisting of 39000 files:



epoch_0



epoch_90

In this study, we observed that the generated images from the entire GTSDB dataset were not as distinguishable as those from the modified dataset. There could be several reasons for this outcome, and we would like to discuss a few possible explanations.

Firstly, the number of training epochs might have been too low to generate significant results for synthesizing realistic traffic sign images. It is possible that running the model for a higher number of epochs, such as 1000 or more, could yield better results. However, due to limited computational resources and time constraints, we were unable to test this hypothesis during our experiment.

Secondly, the size and complexity of the entire GTSDB dataset might have contributed to the less distinguishable generated images. A larger dataset presents a greater diversity of images for the model to learn from, which could require a longer training time or more sophisticated model architecture to capture the intricate patterns and features of the traffic signs.

Challenges and Limitations

In this blog post, we share our experience implementing a traffic sign recognition system using the DCGAN model. We faced several challenges during the process, which we will discuss along with the solutions we found.

Landing on DCGAN as the main model to be used

Our initial challenge was choosing the right model for our project. After reviewing various generative models and considering factors such as stability and performance, we settled on the DCGAN model. Its architecture, combined with other techniques like batch normalisation and using ReLU and LeakyReLU activation functions, made it a suitable choice for our task.

Finding the dataset to train the model on

The next challenge was finding an appropriate dataset for training our DCGAN model. After some research, we decided to use the GTSDB (German Traffic Sign Detection Benchmark) dataset, which consists of annotated images of traffic signs. This dataset provided a good balance of complexity and variety, making it an ideal choice for our task.

The various errors we encountered and how we solved them

During the implementation of our DCGAN model, we faced several errors, such as incorrect input shapes, mismatched dimensions, and improper use of certain functions. To address these issues, we carefully reviewed our code, consulted relevant documentation, and sought advice from online forums and resources. Through iterative debugging, we were able to resolve these errors and improve our model's performance.

The computational time taken for the code to compile and execute

Training a DCGAN model can be computationally expensive, especially when using large datasets and complex architectures. In our case, we faced long compilation and execution times, which slowed down our progress due to primarily working on Jupyter notebook, which does not provide stronger computing power unless we do multiprocessing. To address this issue, we optimised our code by using techniques like early stopping, reducing the number of training epochs, and adjusting the batch size.

Modifying the dataset

Since the computational time was a significant issue, we had to take steps to ensure the success of our code due to the time constraints. At first, we decided on using a modified dataset to train the model on. We took a single folder consisting of 600 files from the GTSDB dataset and trained the model to receive an output. We ran the code for 100 epochs and found a satisfactory result. Simultaneously, we were training the model on the entire GTSDB dataset of 39000+ files on another system to ensure the success of our code. The differences between the outputs of the two datasets are interesting enough to warrant a separate study on how the code functions. We will provide the outputs for both the code implementations, but we are showing the code based on the partial dataset due to the word constraints.

Better performance in different weather conditions

Traffic sign recognition can be challenging due to changes in lighting and weather. GANs can generate images with various lighting and weather conditions, which helps train the recognition model to work well in diverse environments.

Handling various traffic sign types

Traffic signs can vary in design and shape. GANs can create synthetic images of many traffic sign types, helping the recognition model learn to recognize them effectively.

In short, GANs improve traffic sign recognition systems by helping them work well in different conditions and recognize various signal types, making them essential for self-driving cars.

Training data requirements and potential biases

Large and diverse datasets: Machine learning-based traffic sign recognition techniques, especially deep learning methods, require large and diverse datasets for training to achieve high performance and robustness.

Data labelling & Potential biases

Accurate labelling of traffic sign images is essential for supervised learning methods. However, manual labelling can be time-consuming and prone to errors. Training data might be biased towards certain traffic sign types, lighting conditions, or weather situations, leading to poor performance when encountering underrepresented scenarios.

In conclusion, despite the challenges we faced, our implementation of a DCGAN model for traffic sign recognition was a valuable learning experience. It allowed us to explore the potential of generative models in the context of autonomous vehicle safety and navigation while overcoming various technical obstacles along the way.

Limitations of GANs in handling edge cases and unexpected scenarios

Rare and unseen scenarios

GANs may struggle to create and handle traffic sign images from unusual or unknown scenarios because their success is strongly dependent on the diversity and quality of the training data. GANs may not accurately depict specific edge cases in generated images if the training dataset lacks examples of these situations.

Mode collapse

In some cases, GANs can suffer from a phenomenon known as mode collapse, in which the generator produces only a limited number of samples rather than capturing the entire range of the data distribution. This problem may limit GANs' ability to generate diverse traffic sign images, limiting their effectiveness in dealing with edge cases and unexpected scenarios.

Sensitivity to hyperparameters and architecture choices

The performance of GANs can be affected by the hyperparameters and network architecture used.Poor-quality generated images may result from suboptimal configurations, limiting the system's ability to handle difficult situations.

Uncertainty quantification

When confronted with edge cases or unexpected scenarios, GAN-based traffic sign recognition systems may not provide a reliable measure of uncertainty, making it difficult to determine the system's confidence in its predictions.

Generalization

While GANs can improve the robustness of a recognition system, they may struggle to generalise to entirely new scenarios, especially when the differences between training data and real-world situations are significant.

To address these limitations, researchers can investigate strategies such as incorporating more diverse and representative training data, employing advanced GAN architectures, and combining GANs with other techniques to improve the traffic sign recognition system's ability to handle edge cases and unexpected scenarios.

Future Work and Applications

As we move forward with our result after our code implementations, we have found several areas that could be improved upon and expand the capabilities of our model. The following are the aspects we could work on for the future.

Large-scale and diverse datasets

Training the model on more extensive and diverse datasets that include varying lighting conditions, weather, and road environments can further enhance the model's ability to recognize traffic signs in various real-world situations.

Multi-task learning

Exploring the possibility of training the DCGAN model to perform multiple tasks simultaneously, such as traffic sign recognition, lane detection, and vehicle detection, could lead to more efficient and comprehensive autonomous driving systems.

Improved transfer learning

GAN-generated data can help bridge the domain gap between different traffic sign datasets, making it easier to apply pre-trained models to new regions or countries with minimal additional training. This can significantly reduce the time and effort required to deploy autonomous vehicles in new locations.

Real-time adaptation

By incorporating GANs into the onboard systems of autonomous vehicles, it may be possible to enable real-time adaptation of traffic sign detection and classification algorithms. This can help the vehicle dynamically adjust its performance to better handle changing road conditions and traffic sign variations.

Multi-modal integration

GANs can be employed to fuse information from multiple sensors, such as cameras, LiDAR, and radar, to improve traffic sign recognition. This multi-modal approach can help increase the reliability and accuracy of traffic sign detection and classification systems in autonomous vehicles.

Adversarial robustness

GANs can be used to generate adversarial examples, which are images intentionally designed to deceive traffic sign recognition algorithms. By training models on these adversarial examples, researchers can develop more robust traffic sign detection and classification systems that are resilient to potential attacks or manipulations.

Conclusion

Traffic sign recognition is a critical component in the development of autonomous vehicles and intelligent transportation systems. The use of deep learning techniques, such as the DCGAN model, has shown promising results in accurately detecting and classifying traffic signs, ensuring safety and compliance with traffic rules.

Furthermore, transfer learning can provide an efficient and effective approach to leveraging pre-trained models, reducing the need for extensive training data and computational resources.

This blog post has explored the process of implementing a DCGAN model for traffic sign detection, discussing challenges encountered and the steps taken to overcome them. The use of the GTSDB dataset has enabled us to train and evaluate the model, showcasing its potential for real-world applications.

In this blog, we have extensively explained the significance of Traffic sign detection for autonomous vehicles, the Role of GANs in Traffic sign detection, GANs architecture, different types of GAN, Hyperparameters for GANs, the adaption of GTSDB dataset, implementation of working code based on DCGAN for generating and detecting traffic signs, evaluation and comparison of results, discussed the challenges and limitations faced during the execution and compiling phase and have provided insights into the future work which explores other possibilities with GAN that would make it more efficient in image generation.

Our exploration of the DCGAN model for traffic sign recognition demonstrates the potential of generative models in enhancing autonomous vehicle safety and navigation. Despite the challenges we faced during implementation, we were able to overcome them by carefully reviewing our code, optimizing our model, and leveraging available resources. Our experience serves as an example of how deep learning techniques can be applied to real-world problems, leading to innovative solutions that can improve traffic flow, reduce the risk of accidents, and contribute to a more efficient transportation system. We hope our findings inspire further research and development in the field of autonomous vehicle technology and contribute to the ongoing advancements in the area of deep learning for traffic sign recognition.

References

Donahue J, Krähenbühl P, Darrell T, 2017. Adversarial feature learning. arXiv:1605.09782v7 [cs.LG]

Goodfellow I, Pouget-Abadie J, Mirza M, Xu B, Warde-Farley D, Ozair S, Courville A, Bengio Y, 2014. Generative Adversarial Nets. In Advances in Neural Information Processing Systems

Huang W, Huang M, Zhang Y, 2018. Detection of Traffic Signs Based on Combination of GAN and Faster-RCNN. Journal of Physics: Conference Series, 1069(1). https://doi.org/10.1088/1742-6596/1069/1/012159

Hulse LM, Xie H, Galea ER, 2018. Perceptions of autonomous vehicles: Relationships with road users, risk, gender and age.

Odena A, Olah C, Shlens J, 2017. Conditional Image Synthesis with Auxiliary Classifier GANs. arXiv:1610.09585v4 [stat.ML]

Pan X, You Y, Wang Z, Lu C, 2017. Virtual to Real Reinforcement Learning for Autonomous Driving. In British Machine Vision Conference (BMVC), London, UK

Radford A, Metz L, Chintala S, 2016. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. arXiv:1511.06434v2 [cs.LG]

Sankaranarayanan S, Balaji Y, Jain A, Lim SN, Chellappa R, 2018. Learning from Synthetic Data: Addressing Domain Shift for Semantic Segmentation. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)

Shanmugavel AB, Ellappan V, Mahendran A, Subramanian M, Lakshmanan R, Mazzara M, 2023. A Novel Ensemble Based Reduced Overfitting Model with Convolutional Neural Network for Traffic Sign Recognition System. Electronics (Switzerland), 12(4). https://doi.org/10.3390/electronics12040926

Links for GTSDB dataset -

https://www.kaggle.com/datasets/safabouguezzi/german-traffic-sign-detection-benchmark-gtsdb

http://benchmark.ini.rub.de/?section=gtsdb&subsection=dataset