Implementation of Conditional GAN and StyleGAN

Table of Contents

ABSTRACT

In order to create handwritten digits with certain characteristics, this project focuses on the use of Conditional Generative Adversarial Networks (GANs) on the MNIST dataset. The stated issue is how to produce realistic and varied digit images based on characteristics such as digit class, stroke thickness, or tilt. The suggested method combines a generator and discriminator network to create digit pictures with required properties while also learning the underlying patterns of numbers in the dataset. This project explores the idea of combining styles while using the StyleGAN architecture and also producing realistic portraits. This project aims to create portraits from noise and combine two different portrait styles.

INTRODUCTION

In recent years, generative adversarial networks (GANs) have emerged as powerful tools for generating realistic and diverse images. GANs have revolutionized the field of Machine learning and have found applications in various domains such as art, design, entertainment, and data augmentation.

The project focuses on the application of Conditional Generative Adversarial Networks (GANs) and style mixing techniques for image generation tasks. The goal is to generate handwritten digits, portrait images and style mixing.

PROBLEM SPECIFICATION

The problem addressed in this project is the generation of images with specific attributes using Conditional GANs and style mixing techniques. Specifically, the project focuses on two domains: handwritten digit generation on the MNIST dataset and portrait image generation using StyleGAN.

The generation of images with specific attributes is important in various applications. For example, in the case of handwritten digits, being able to generate specific digits rather than generating random, for training machine learning models, and generating synthetic datasets.

Similarly to this, the ability to create different and realistic portrait photos with particular features can be used in industries like art, entertainment, and virtual reality. It gives designers, artists, and content producers a wider range of options for making distinctive and adaptable portraits.

BACKGROUND

In the field of generative models, several alternative approaches have been explored for image synthesis and manipulation. Here are some notable alternatives to the proposed approach:

Traditional GANs:

Traditional GANs, introduced by Goodfellow et al. in 2014, are a widely studied and utilized generative model. These models are made up of an adversarial training process between a discriminator network and a generator network. They have been successfully used in a variety of fields, including music, text, and image generation. Traditional GANs, on the other hand, lack explicit control over the generated samples, which restricts their applicability in situations where certain requirements or characteristics must be upheld.

Variational Autoencoders (VAEs):

Autoencoders and generative models are combined in variational autoencoders (VAEs). In order to control generation and interpolation between diffrent samples, VAEs learn a latent space representation of the input data.It is extensively used including picture production, anomaly detection, and data compression. Notable literature VAE study by Kingma and Welling (2013) and subsequent works on improving VAE training and sample quality.

PixelRNN and PixelCNN:

Pixel-by-pixel images are created by the autoregressive models PixelRNN and PixelCNN. They enable the creation of coherent, high-quality images by modelling the conditional probability distribution of each pixel given the pixels given the previously generated pixels. PixelRNN was introduced by van den Oord et al. (2016), while PixelCNN was proposed by van den Oord et al. (2016) and later improved upon in subsequent works. These models have been influential in capturing fine-grained details in generated images.

CONNECTION TO CS 713 ASSIGNMENT

The assignment of training a GAN on the MNIST dataset and implementing functionalities for training, saving, loading, and generating images directly related with the project's objective of implementing a Conditional GAN and StyleGAN. Completing the assignment provides a crucial component for implementing the Conditional GAN on MNIST, which is essential for generating controlled digit images. The functionalities developed for training and generating images can also be extended to the project's Style Mixing aspect, allowing for the creation of unique and artistic portraits through the blending of different styles. Therefore, the assignment serves as a fundamental building block for achieving the project's goals and enhancing the implementation of the Conditional GAN on MNIST and Style Mixing for Portraits using StyleGAN.Through this project, I aim to contribute to the advancement of GAN assignment and development while gaining insights into the cGANs and StyleGAN.

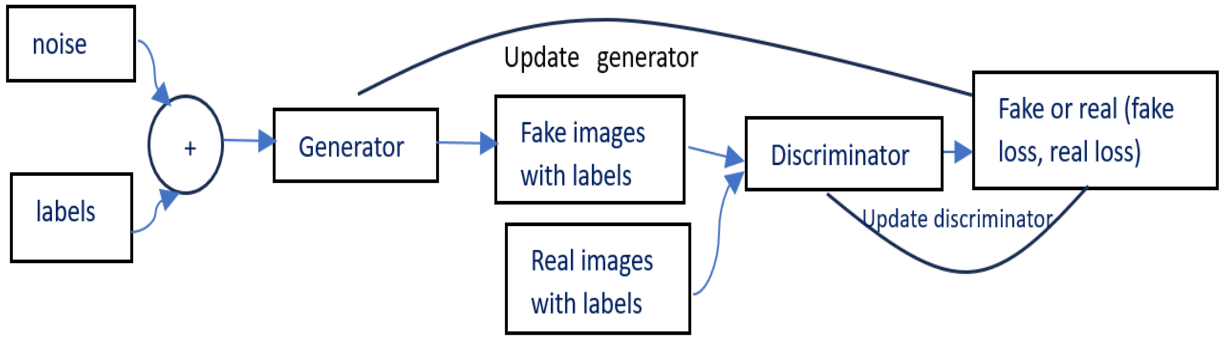
CONNECTION TO RESEARCH IDEA

Generative models, such as conditional GANs and StyleGAN, have gained significant attention in recent years for their ability to generate realistic and high-quality images.

The project aims to use the MNIST dataset to create a Conditional Generative Adversarial Network (cGAN). The cGAN produce digit pictures that are controlled and conditioned on particular attributes such as class labels or attribute vectors. This is consistent with the research plan investigating conditional generation and its uses. The proposal also includes the idea of employing StyleGAN for style mixing. The ability to combine many styles or characteristics to create wholly original and beautiful portraits is known as style mixing. The research investigates the creation of varied and aesthetically pleasing portrait photographs with various styles by using StyleGAN's architecture and methodologies.

APPROACH/METHOD(s)

Conditional GAN

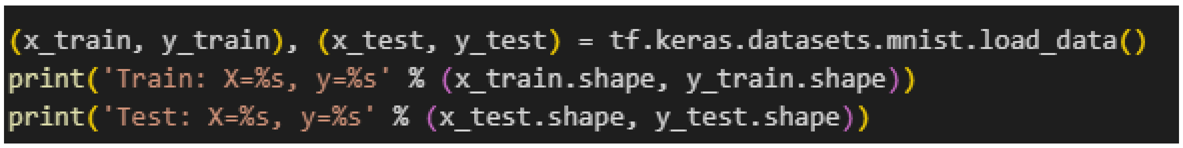


The technique used by Conditional Generative Adversarial Networks (cGAN) involves developing a generative model (the generator) to generate images that are conditional on particular input conditions or labels. The classic GAN architecture is extended by the cGAN framework by adding labels to control the generation of images. Labels, text descriptions, or other variables that control the generator and discriminator.

The cGAN consists of two main components: the generator and the discriminator.

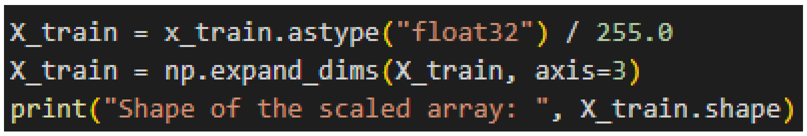
The use of LeakyReLU activation function in this context helps the generator model to introduce non-linearity and capture more complex patterns and structures in the generated images. It allows the model to handle different levels of activations, including negative values, which can be beneficial for generating diverse and realistic images.

Load the MINST data set from Keras.



```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
print('Train: X=%s, y=%s' % (x_train.shape, y_train.shape))
print('Test: X=%s, y=%s' % (x_test.shape, y_test.shape))
```

Normalize it as the pixel value range from o to 255. If we divide the image with 255 it will get normalized.



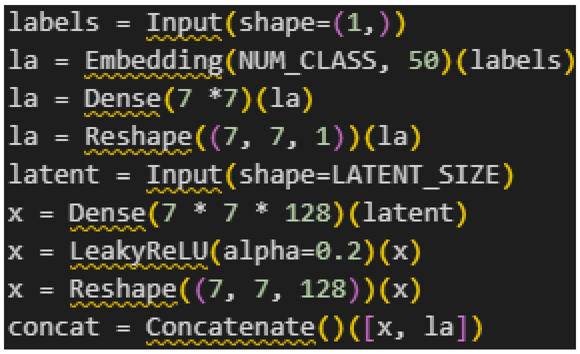
```
X_train = x_train.astype("float32") / 255.0
X_train = np.expand_dims(X_train, axis=3)
print("Shape of the scaled array: ", X_train.shape)
```

Generator

Inputs

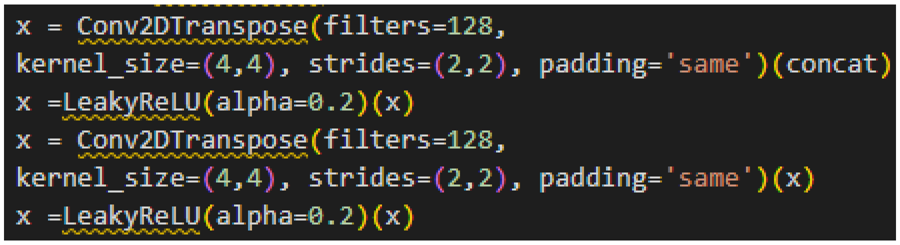
Labels: It is creating an embedding layer with 10(0 to 9 digits) as the input dimension and 50 as the output dimension of the embedding vectors for capturing more complex relationships. The embedded labels are then passed through a dense layer to reshape them into a compatible shape for concatenation.The Dense(7 * 7) layer is a fully connected layer that takes the input tensor la and applies a linear transformation to produce a tensor of shape (7 * 7,). The reshapes the tensor la into a 3-dimensional tensor with a shape of (7, 7, 1).

Latent vector: The latent vector serves as a random noise input to the generator. A noise vector is used to transform the random noise into a more structured representation. The alpha parameter in leaky relu controls the slope of the negative region of the activation function. Then reshape into a specific dimension. The latent input and the label representation are concatenated along the channel axis to incorporate the label information into the generator with the dimensions of (7,7,1)+(7,7,129) = (7,7,129).



```
labels = Input(shape=(1,))
la = Embedding(NUM_CLASS, 50)(labels)
la = Dense(7 *7)(la)
la = Reshape((7, 7, 1))(la)
latent = Input(shape=LATENT_SIZE)
x = Dense(7 * 7 * 128)(latent)
x = LeakyReLU(alpha=0.2)(x)
x = Reshape((7, 7, 128))(x)
concat = Concatenate()([x, la])
```

Upsampling the noise to form image: The Transposed convolution performs the operation of up-sampling the input concerted information by applying a 2D kernel of size 4x4 and a stride of 2x2. The filters=128 parameter specifies the number of output filters or feature maps for this convolutional layer. The padding ensures that the spatial dimensions of the input and output feature maps remain the same. It is passed through the leakyrelu activation.



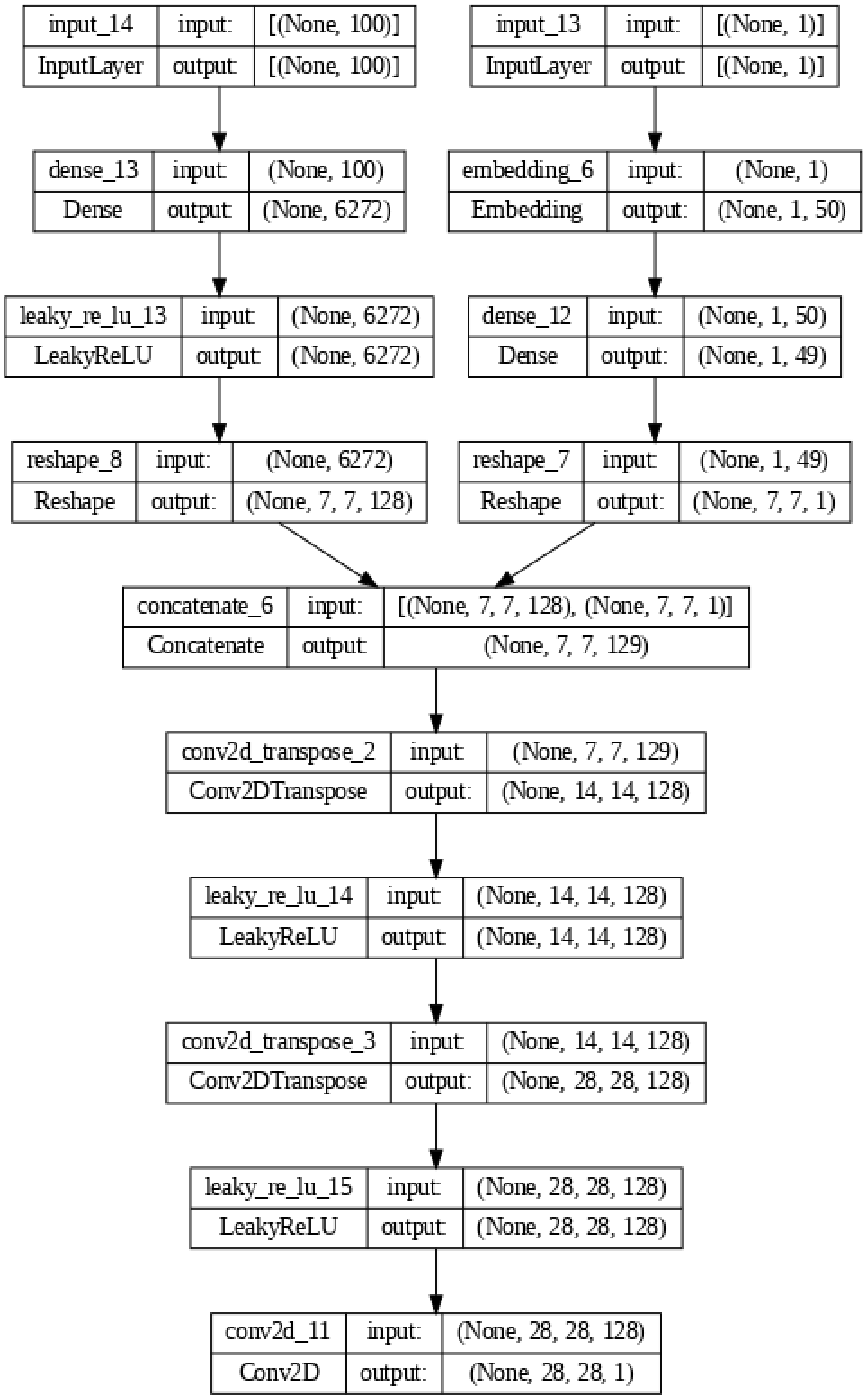
```
x = Conv2DTranspose(filters=128,
kernel_size=(4,4), strides=(2,2), padding='same')(concat)
x =LeakyReLU(alpha=0.2)(x)
x = Conv2DTranspose(filters=128,
kernel_size=(4,4), strides=(2,2), padding='same')(x)
x =LeakyReLU(alpha=0.2)(x)
```

Output :

The final convolutional layer in the generator model applies a 2D convolution with a kernel of size 7x7 and a single output filter. The padding argument ensures that the spatial dimensions of the input and output feature maps remain the same. The activation function used is the sigmoid function. which maps the output values between 0 and 1, representing the generated image's pixel intensities.And filter one as one dimensal output gry scale.

Flow chart:

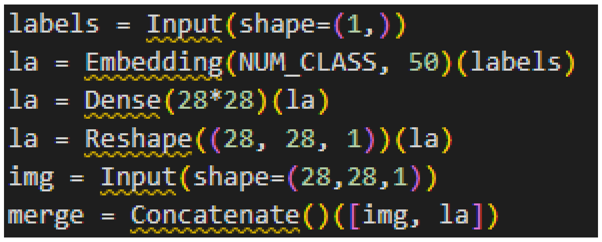
The concatenated image of (7,7,7,129) is sent through transpose convolution layer for upsampling which gives (14,14,128) and sent to give activation for nolinery and to avoid dead neurons with slope of 0.2. This further passed to transpose convolution and activation will result in (28,28,128). The final final convluion layer downs samples the image to 28,28,1 as gray scale imgae.


input_14 | input: | [(None, 100)]
InputLayer | output: | [(None, 100)]
input_13 | input: | [(None, 1)]
InputLayer | output: | [(None, 1)]
dense_13 | input: | (None, 100)
Dense | output: | (None, 6272)
embedding_6 | input: | (None, 1)
Embedding | output: | (None, 1, 50)
leaky_re_lu_13 | input: | (None, 6272)
LeakyReLU | output: | (None, 6272)
dense_12 | input: | (None, 1, 50)
Dense | output: | (None, 1, 49)
reshape_8 | input: | (None, 6272)
Reshape | output: | (None, 7, 7, 128)
reshape_7 | input: | (None, 1, 49)
Reshape | output: | (None, 7, 7, 1)
concatenate_6 | input: | [(None, 7, 7, 128), (None, 7, 7, 1)]
Concatenate | output: | (None, 7, 7, 129)
conv2d_transpose_2 | input: | (None, 7, 7, 129)
Conv2DTranspose | output: | (None, 14, 14, 128)
leaky_re_lu_14 | input: | (None, 14, 14, 128)
LeakyReLU | output: | (None, 14, 14, 128)
conv2d_transpose_3 | input: | (None, 14, 14, 128)
Conv2DTranspose | output: | (None, 28, 28, 128)
leaky_re_lu_15 | input: | (None, 28, 28, 128)
LeakyReLU | output: | (None, 28, 28, 128)
conv2d_11 | input: | (None, 28, 28, 128)
Conv2D | output: | (None, 28, 28, 1)

Discriminator

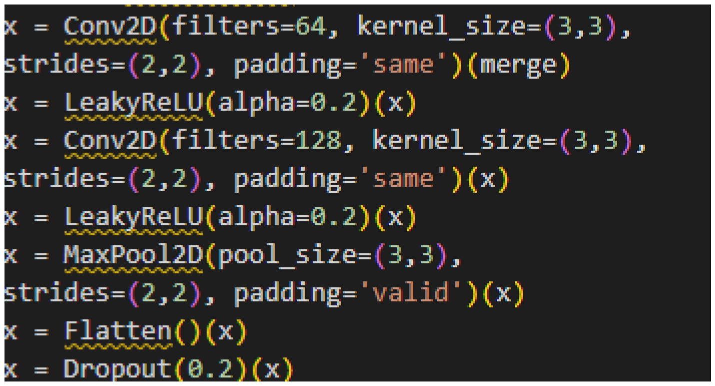
Inputs

The input layer is defined for the label, which represents the class of the image it passed to an embedding layer to convert the label into a dense representation. It maps the label to a continuous vector space, allowing the network to learn meaningful representations for different labels. The dense layer is applied to reshape the embedding into the same shape as the image (28x28) to enable concatenation and the input layer is defined for the image, which represents the 28x28 grayscale image.



```
labels = Input(shape=(1,))
la = Embedding(NUM_CLASS, 50)(labels)
la = Dense(28*28)(la)
la = Reshape((28, 28, 1))(la)
img = Input(shape=(28,28,1))
merge = Concatenate()([img, la])
```

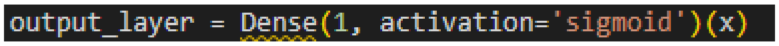
Downsampling the image to evaluate it as fake or not: The convolution layer performs the operation of Downsampling the input concerted information by applying a 2D kernel of size 3*3 and a stride of 2x2. The filters=128 parameter specifies the number of output filters or feature maps for this convolutional layer. The padding ensures that the spatial dimensions of the input and output feature maps remain the same. It is passed through the leakyrelu activation. The second Conv2D layer has 128 filters, the same kernel and stride size, and another LeakyReLU activation. A MaxPooling2D layer is used to downsample the feature maps by selecting the maximum value within a 3x3 window, with a stride of (2,2). Then it is Flatten to a 1D vector and the Dropout function is applied to prevent overfitting by randomly setting a fraction of inputs to 0 during training.



```
x = Conv2D(filters=64, kernel_size=(3,3),
strides=(2,2), padding='same')(merge)
x = LeakyReLU(alpha=0.2)(x)
x = Conv2D(filters=128, kernel_size=(3,3),
strides=(2,2), padding='same')(x)
x = LeakyReLU(alpha=0.2)(x)
x = MaxPool2D(pool_size=(3,3),
strides=(2,2), padding='valid')(x)
x = Flatten()(x)
x = Dropout(0.2)(x)
```

Output

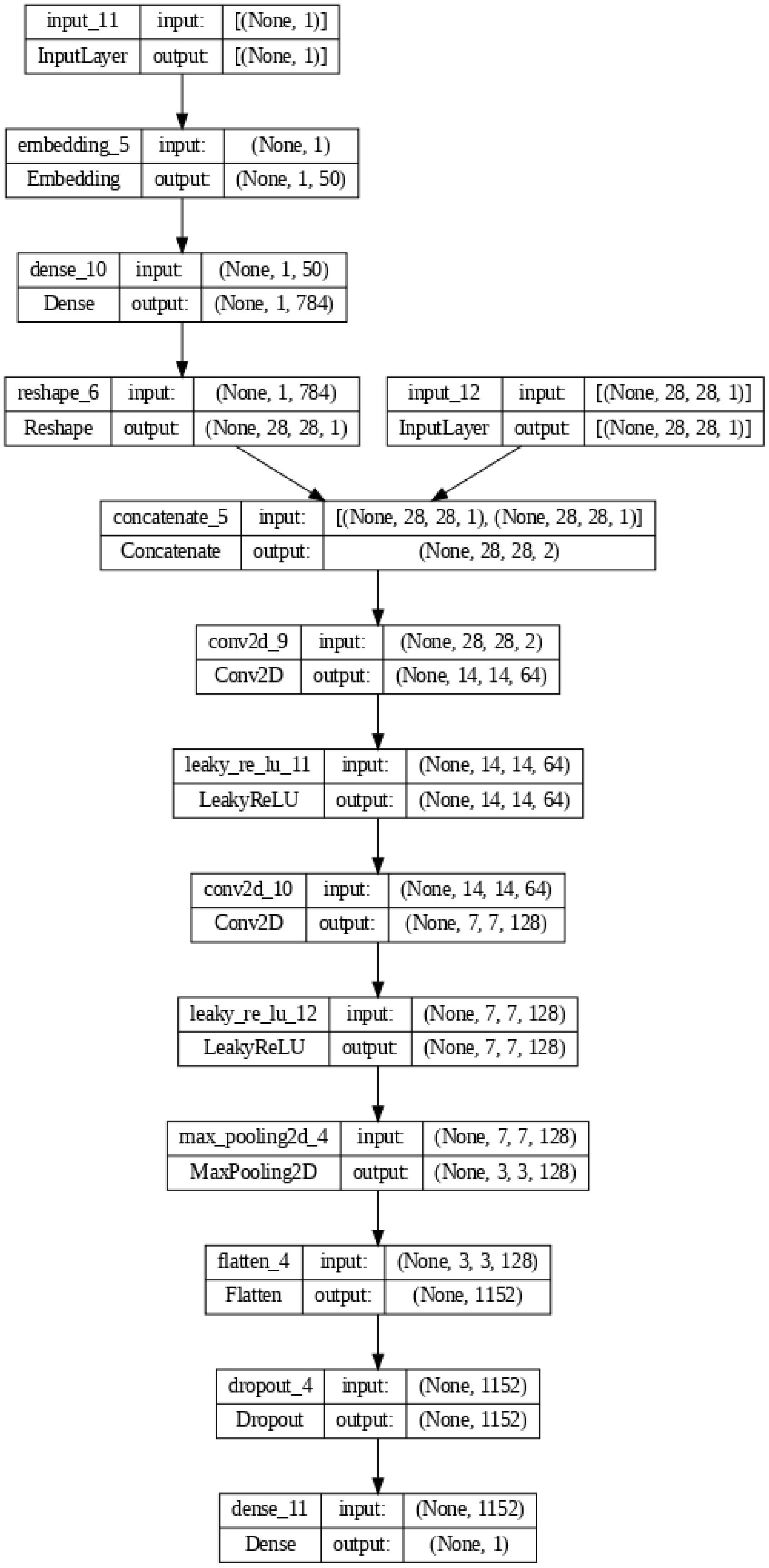
The output layer is a dense layer with a single unit and sigmoid activation function, indicating image is real or fake.



```
output_layer = Dense(1, activation='sigmoid')(x)
```

Flow chart:

The concatenated image of (28,28,2) is sent through the convolution layer for downsampling which gives (14,14,128) and is sent to give activation for nolinearty and to avoid dead neurons with a slope of 0.2. This further passed to convolution and activation will result in (7,7,128). The max polling downsamples the image to (3,3,128) to extract more features. Then it is flattened to a 1D vector of shape(None,1152) and the Dropout function is applied to prevent overfitting. The final dense with a single unit and sigmoid activation function, indicating the image is real or fake.


input_11 | InputLayer | input: [(None, 1)] | output: [(None, 1)]
embedding_5 | Embedding | input: (None, 1) | output: (None, 1, 50)
dense_10 | Dense | input: (None, 1, 50) | output: (None, 1, 784)
reshape_6 | Reshape | input: (None, 1, 784) | output: (None, 28, 28, 1)
input_12 | InputLayer | input: [(None, 28, 28, 1)] | output: [(None, 28, 28, 1)]
concatenate_5 | Concatenate | input: [(None, 28, 28, 1), (None, 28, 28, 1)] | output: (None, 28, 28, 2)
conv2d_9 | Conv2D | input: (None, 28, 28, 2) | output: (None, 14, 14, 64)
leaky_re_lu_11 | LeakyReLU | input: (None, 14, 14, 64) | output: (None, 14, 14, 64)
conv2d_10 | Conv2D | input: (None, 14, 14, 64) | output: (None, 7, 7, 128)
leaky_re_lu_12 | LeakyReLU | input: (None, 7, 7, 128) | output: (None, 7, 7, 128)
max_pooling2d_4 | MaxPooling2D | input: (None, 7, 7, 128) | output: (None, 3, 3, 128)
flatten_4 | Flatten | input: (None, 3, 3, 128) | output: (None, 1152)
dropout_4 | Dropout | input: (None, 1152) | output: (None, 1152)
dense_11 | Dense | input: (None, 1152) | output: (None, 1)

Conditonal GAN:

It takes the generator's input (latent vector and label) and passes them through the generator and discriminator networks to produce the GAN's output.

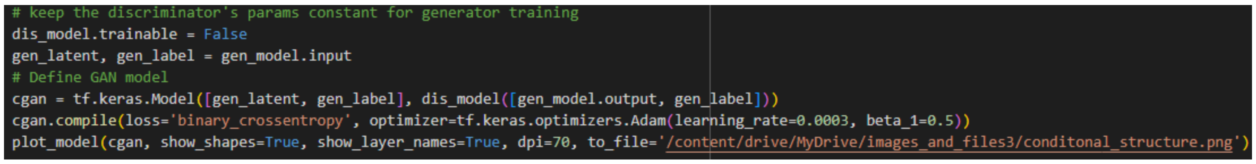
the binary cross-entropy loss function is used as the objective function for the GAN training, and the Adam optimizer is chosen with a learning rate of 0.0003 and a beta_1 value of 0.5.

The conditional GAN is trained on the generated noise vectors and fake labels. The target label for the generator is set as one, indicating that the generated samples are real.

The discriminator is then trained on this subset of real samples, aiming to correctly classify them as real and give real loss.

The discriminator is then trained on these generated fake samples along with their corresponding random class labels, aiming to correctly classify them as fake and give fake loss.

Using LeakyReLU activation function in this context helps the generator model introduce non-linearity and capture more complex patterns and structures in the generated images. It allows the model to handle different levels of activations, including negative values, which can be beneficial for generating diverse and realistic images



```
# keep the discriminator's params constant for generator training
dis_model.trainable = False
gen_latent, gen_label = gen_model.input
# Define GAN model
cgan = tf.keras.Model([gen_latent, gen_label], dis_model([gen_model.output, gen_label]))
cgan.compile(loss='binary_crossentropy', optimizer=tf.keras.optimizers.Adam(learning_rate=0.0003, beta_1=0.5))
plot_model(cgan, show_shapes=True, show_layer_names=True, dpi=70, to_file='/content/drive/MyDrive/images_and_files3/conditonal_structure.png')
```

Adam optimizes: it is used because It combines the advantages of two other popular optimization algorithms, AdaGrad and RMSprop, to provide adaptive learning rates and momentum.

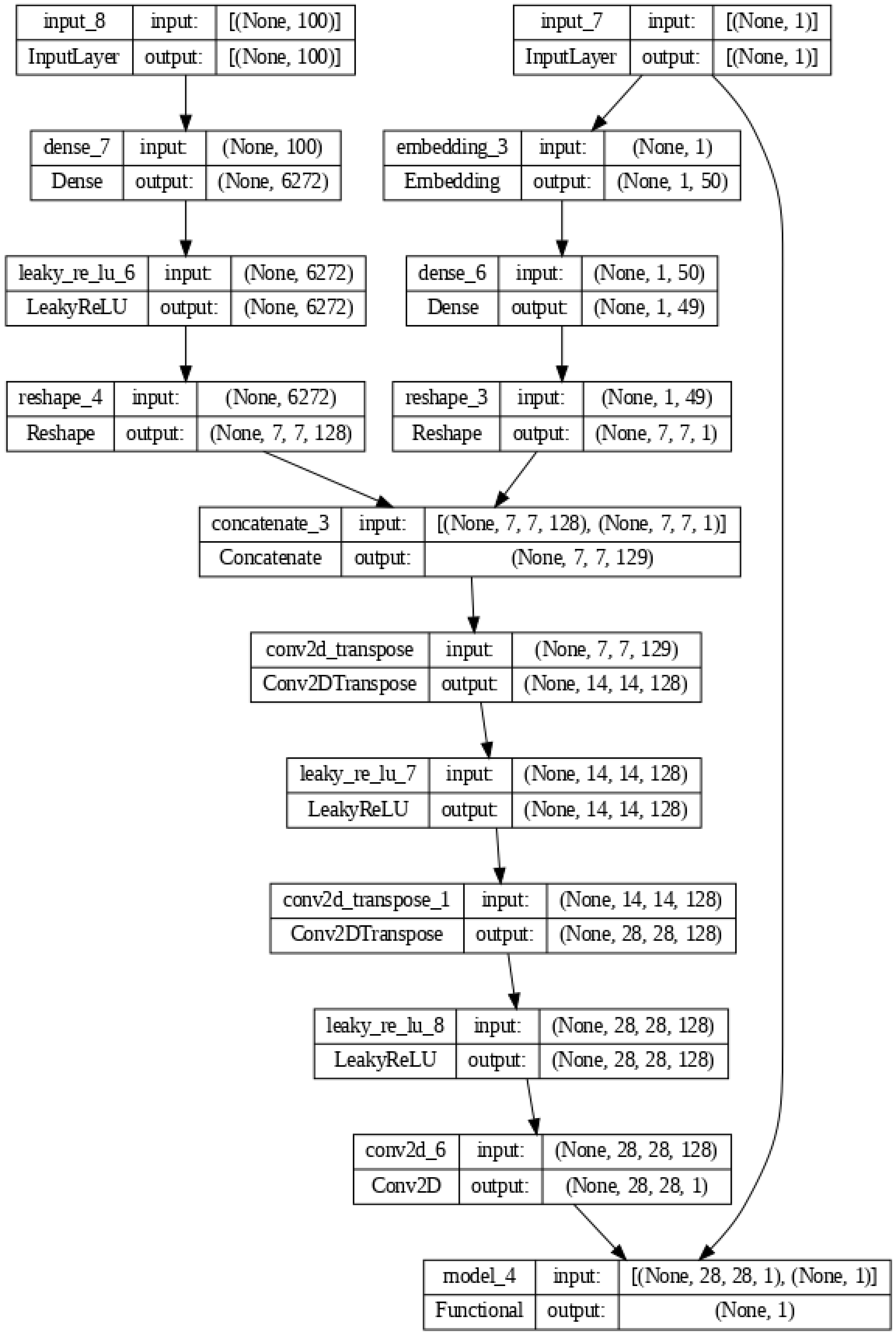
Binary cross-entropy loss: To measure the dissimilarity between the predicted probabilities and the true binary labels

Mathematically, the binary cross-entropy loss is computed as follows:

$$\text{loss} = -[(y\_true * \log(y\_pred)) + ((1 - y\_true) * \log(1 - y\_pred))]$$

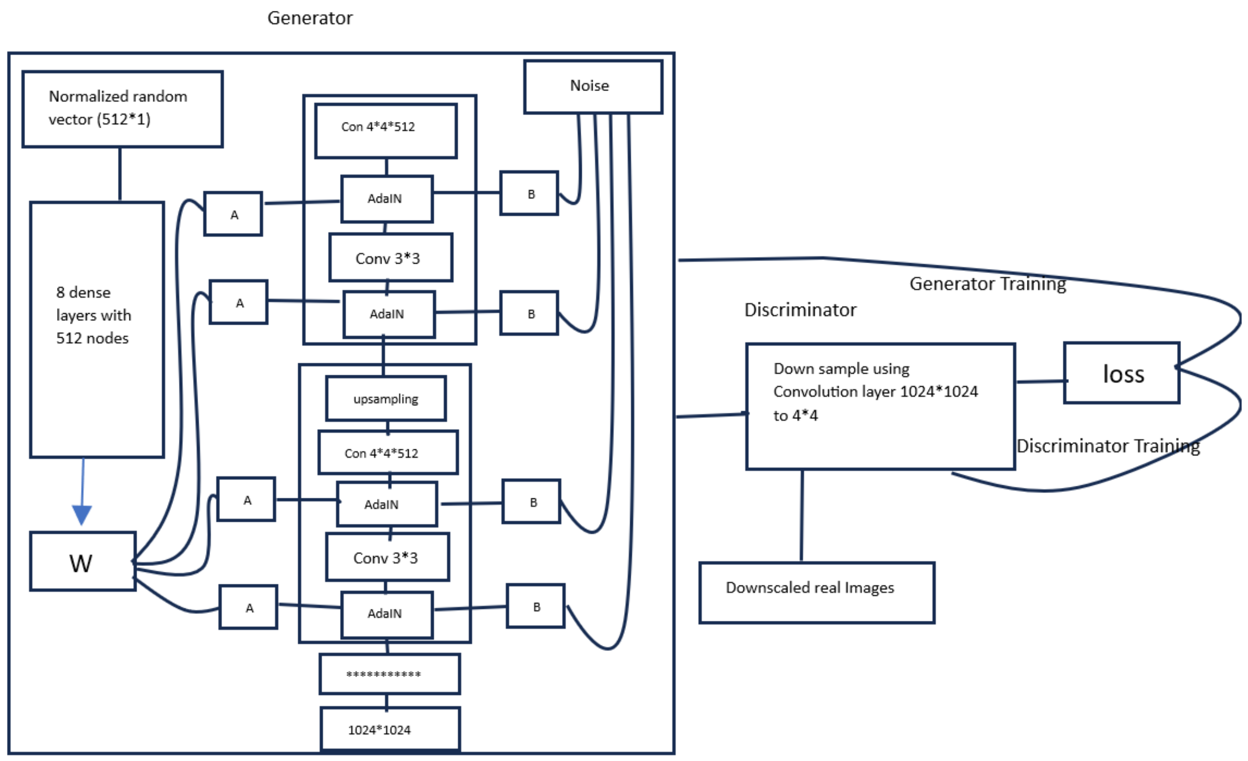
where: y_true represents the true binary label (0 or 1).

y_pred represents the predicted probability by the discriminator.

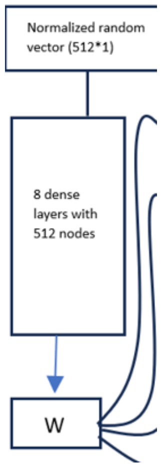

input_8 | InputLayer | input: [(None, 100)] | output: [(None, 100)]
input_7 | InputLayer | input: [(None, 1)] | output: [(None, 1)]
dense_7 | Dense | input: (None, 100) | output: (None, 6272)
embedding_3 | Embedding | input: (None, 1) | output: (None, 1, 50)
leaky_re_lu_6 | LeakyReLU | input: (None, 6272) | output: (None, 6272)
dense_6 | Dense | input: (None, 1, 50) | output: (None, 1, 49)
reshape_4 | Reshape | input: (None, 6272) | output: (None, 7, 7, 128)
reshape_3 | Reshape | input: (None, 1, 49) | output: (None, 7, 7, 1)
concatenate_3 | Concatenate | input: [(None, 7, 7, 128), (None, 7, 7, 1)] | output: (None, 7, 7, 129)
conv2d_transpose | Conv2DTranspose | input: (None, 7, 7, 129) | output: (None, 14, 14, 128)
leaky_re_lu_7 | LeakyReLU | input: (None, 14, 14, 128) | output: (None, 14, 14, 128)
conv2d_transpose_1 | Conv2DTranspose | input: (None, 14, 14, 128) | output: (None, 28, 28, 128)
leaky_re_lu_8 | LeakyReLU | input: (None, 28, 28, 128) | output: (None, 28, 28, 128)
conv2d_6 | Conv2D | input: (None, 28, 28, 128) | output: (None, 28, 28, 1)
model_4 | Functional | input: [(None, 28, 28, 1), (None, 1)] | output: (None, 1)

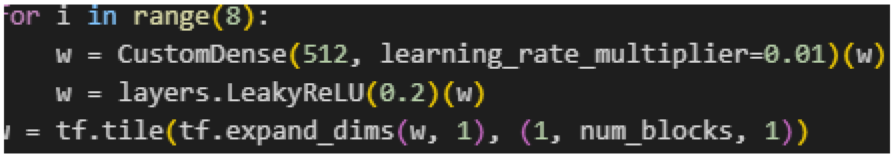
StyleGAN

Load dataset from Kaggle. Then resize the image with the nearest neighborhood interpolation



The Mapping function defines a mapping network that takes a latent vector z as input shape of batch and latent size 512, applies 8 layers of equalized dense units with LeakyReLU activations, and produces an intermediate latent representation w. The purpose of this network is to transform the input latent vector into a higher-level latent space that can be further utilized in the generation process

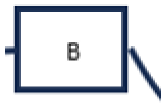

Normalized random vector (512*1)
8 dense layers with 512 nodes
W

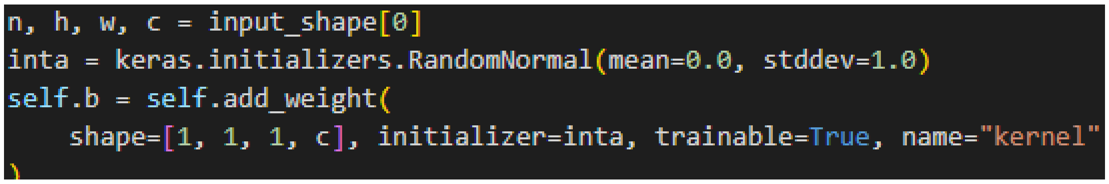


```
or i in range(8):
   w = CustomDense(512, learning_rate_multiplier=0.01)(w)
   w = layers.LeakyReLU(0.2)(w)
 = tf.tile(tf.expand_dims(w, 1), (1, num_blocks, 1))
```

The custom nose layer neural network. where the standard devotion of all weights falls from 0 to 1 which helps to initialize the weights in a way that allows for effective learning during the training process and varies with the bias based on channels.

output = input + bias * noise.

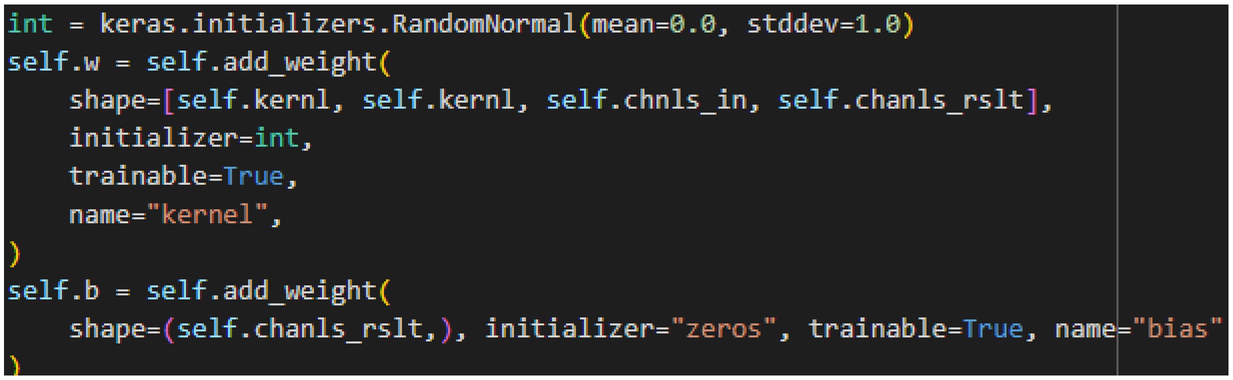




```
n, h, w, c = input_shape[0]
inta = keras.initializers.RandomNormal(mean=0.0, stddev=1.0)
self.b = self.add_weight(
    shape=[1, 1, 1, c], initializer=inta, trainable=True, name="kernel"
)
```

Build a custom convolution layer where the standard devotion of all weights falls from 0 to 1 which helps to initialize the weights in a way that allows for effective learning during the training process and bias is initialized as zero. The scale factor is calculated as the square root of the gain divided by the fadeing value, where fade is the product of kernel size, kernel size, and input channels.

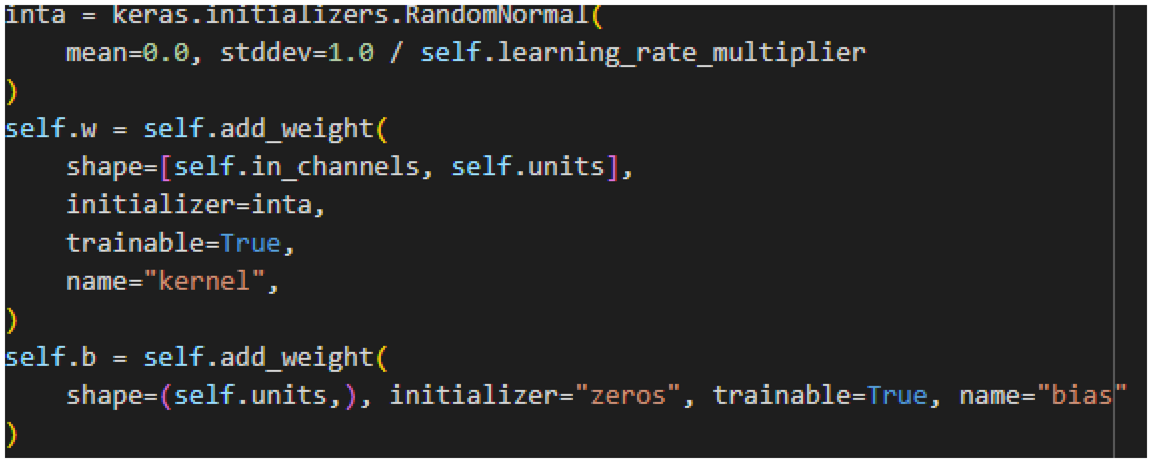
Output = Convolution network (scale factor * weight) + bias.



```
int = keras.initializers.RandomNormal(mean=0.0, stddev=1.0)
self.w = self.add_weight(
    shape=[self.kernl, self.kernl, self.chnls_in, self.chanls_rslt],
    initializer=int,
    trainable=True,
    name="kernel",
)
self.b = self.add_weight(
    shape=(self.chanls_rslt,), initializer="zeros", trainable=True, name="bias"
)
```

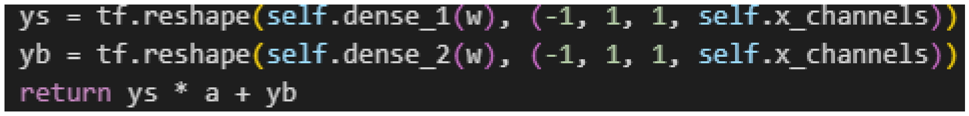
Build a custom dense layer where the standard devotion of all weights falls from 0 to 1 which helps to initialize the weights in a way that allows for effective learning during the training process and bias is initialized as zero. and the output is controlled by the learning rate which is multiplied by the learning rate.

Output = channel of color (it depends on the fade of color) * weight +bias



```
inta = keras.initializers.RandomNormal(
    mean=0.0, stddev=1.0 / self.learning_rate_multiplier
)
self.w = self.add_weight(
    shape=[self.in_channels, self.units],
    initializer=inta,
    trainable=True,
    name="kernel",
)
self.b = self.add_weight(
    shape=(self.units,), initializer="zeros", trainable=True, name="bias"
)
```

The adaptive Instance Normalization layer is used to adjust the style of an input feature map (x) based on the style parameters. The input feature vector from noise and style vector are sent to two different custom dense works and reshaped which results to give two styling parameters. Scaling the input using these two parameters adjusts the style of the input feature map based on the learned style parameters.



```
ys = tf.reshape(self.dense_1(w), (-1, 1, 1, self.x_channels))
yb = tf.reshape(self.dense_2(w), (-1, 1, 1, self.x_channels))
return ys * a + yb
```

Generator:

The generator will grow with respective to resolution.

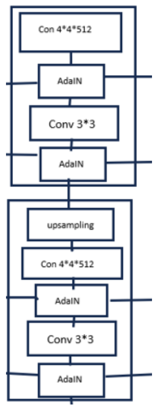
At each stage, a noise input layer is created to introduce random variations into the generator's output. This noise input is specific to the current resolution. An RGB conversion layer is created to convert the activation of the generator block to RGB output. This layer is responsible for generating the final image at the current resolution. If the current stage is not the base stage, up-sampling and a convolution layer are applied to increase the resolution of the generator's output. This allows the generator to generate more detailed images as the resolution increases.

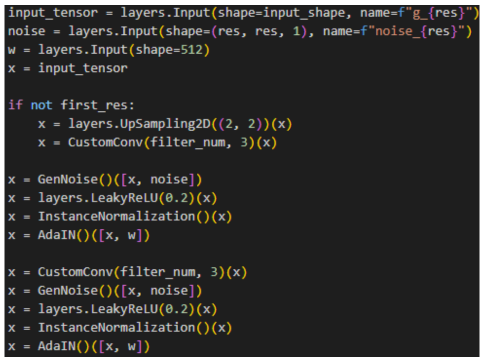
Noise is added to the current flow. This helps introduce additional variations and stochasticity to the generated images.

LeakyReLU activation is applied to introduce non-linearity and help the network learn more complex patterns and structures.

Instance normalization is applied to normalize the activations, which can help with stabilizing training and improving the quality of generated images.

Adaptive Instance Normalization is applied using the activation and the input latent vector w. AdaIN dynamically adjusts the normalization parameters based on the input latent vector, allowing the network to control style and appearance based on the input.





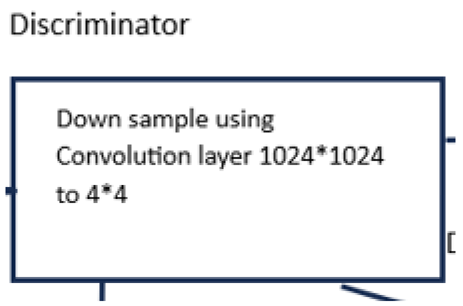
```
input_tensor = layers.Input(shape=input_shape, name=f"g_{res}")
noise = layers.Input(shape=(res, res, 1), name=f"noise_{res}")
w = layers.Input(shape=512)
x = input_tensor

if not first_res:
    x = layers.UpSampling2D((2, 2))(x)
    x = CustomConv(filter_num, 3)(x)

x = GenNoise()([x, noise])
x = layers.LeakyReLU(0.2)(x)
x = InstanceNormalization()(x)
x = AdaIN()([x, w])

x = CustomConv(filter_num, 3)(x)
x = GenNoise()([x, noise])
x = layers.LeakyReLU(0.2)(x)
x = InstanceNormalization()(x)
x = AdaIN()([x, w])
```

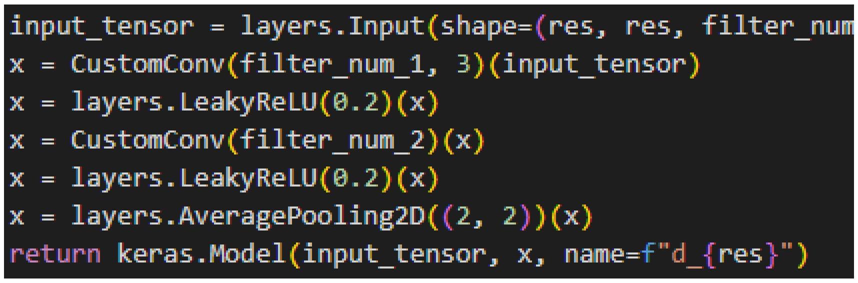
Discriminator

The discriminator block is applied from higher resolution to lower resolution.

In the first stage, it takes the input tensor calculates the average standard deviation across grouped samples in the input tensor and then replicates it to match the shape of the input tensor, an equalized convolution layer, LeakyReLU activation, flattening, two equalized dense layers with LeakyReLU activations, and a final equalized dense layer to output a single value representing the decision of real or fake.

In the higher resolution, the input is passed to a series of custom convolution layers and activation functions and down samples the resolution using average pooling.





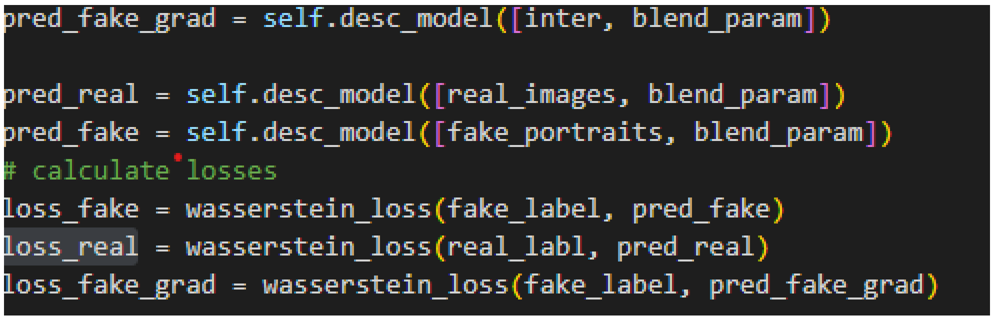
```
input_tensor = layers.Input(shape=(res, res, filter_num
x = CustomConv(filter_num_1, 3)(input_tensor)
x = layers.LeakyReLU(0.2)(x)
x = CustomConv(filter_num_2)(x)
x = layers.LeakyReLU(0.2)(x)
x = layers.AveragePooling2D((2, 2))(x)
return keras.Model(input_tensor, x, name=f"d_{res}")
```

Style GAN

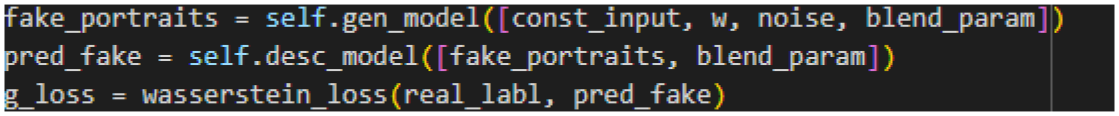
It is responsible for growing discriminators and generators with respect to the target resolution.

In training face the generator generates fake images using the generator, computes the generator loss based on the Wasserstein loss, and calculates and applies gradients to update the generator's trainable weights. It then computes the discriminator loss based on the Wasserstein loss, gradient penalty, and drift loss, and applies gradients to update the discriminator's trainable weights which control the discriminator decisions.

Model inference by generating images based on input style codes, latent vectors, noise, batch size, and alpha values. It uses the generator to generate images and applies and transforms the pixels of the image to the valid range.



```
pred_fake_grad = self.desc_model([inter, blend_param])

pred_real = self.desc_model([real_images, blend_param])
pred_fake = self.desc_model([fake_portraits, blend_param])
# calculate losses
loss_fake = wasserstein_loss(fake_label, pred_fake)
loss_real = wasserstein_loss(real_labl, pred_real)
loss_fake_grad = wasserstein_loss(fake_label, pred_fake_grad)
```



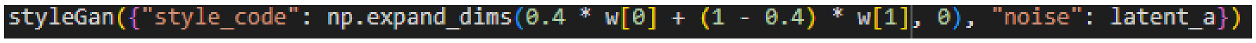
```
fake_portraits = self.gen_model([const_input, w, noise, blend_param])
pred_fake = self.desc_model([fake_portraits, blend_param])
g_loss = wasserstein_loss(real_labl, pred_fake)
```

Style mixing

Mix two style vectors using linear interpolation. This will create a style-mixed mapping network and an additional dimension. Input this to generated StyleGan weights to create a new style.

A mixed style code is computed by linearly interpolating between the style codes and using blend. This interpolation combines the styles of the two vectors and introduces a new dimension representing the mixed style.

The mixed-style vector and expanded noise tensors are used as inputs to the model, generating mixed images. These images reflect the combined style of the original two images, allowing for the exploration of new visual styles.



```
styleGan({"style_code": np.expand_dims(0.4 * w[0] + (1 - 0.4) * w[1], 0), "noise": latent_a})
```

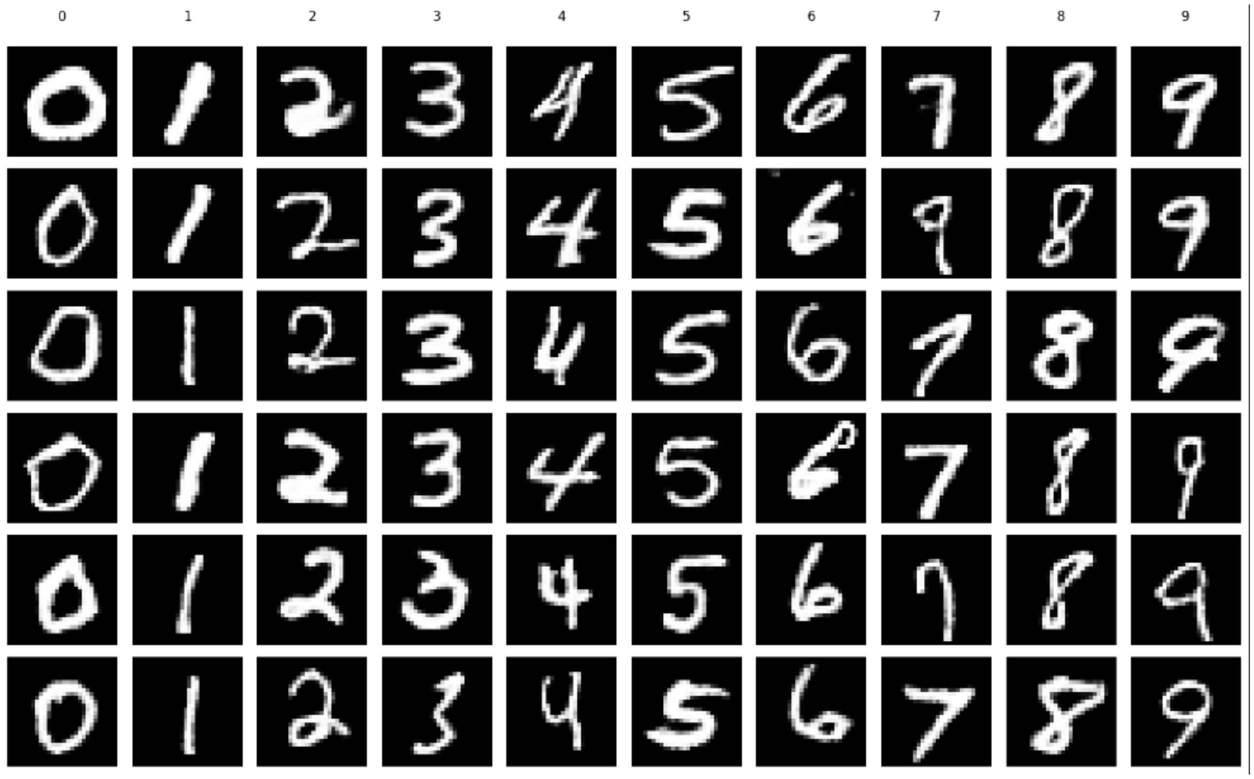
RESULTS

Conditional GAN

The model's performance was evaluated based on image diversity, image quality, and label conditioning. The results demonstrated that the cGAN model achieved good outcomes in terms of generating diverse numbers and maintaining good image quality while conditioning on specific labels. It successfully captured variations in the handwritten digits, producing a wide range of different shapes and styles. The generated images exhibited clear and well-defined digits, with smooth edges and consistent stroke thickness.

Furthermore, the cGAN model demonstrated effective conditioning on specific labels. The generated images accurately represented the specified labels, aligning with the desired digit class. This successful conditioning indicates that the model effectively learned the relationship between the input labels and the corresponding image features.

The generated samples closely resembled the real dataset, with consistent proportions, recognizable shapes, and accurate stroke patterns. However, some variations were also noticeable in the generated images. While these variations were present, they did not deviate significantly from the overall digit structure and were within an acceptable range.



Divers images were generated with different stocks with conditioning.

StyleGAN

The StyleGAN model was employed to generate portraits of 64x64 resolution images based on the latent space. Due to the relatively low resolution, the generated images may exhibit certain limitations in terms of fine details and overall image quality. This can be attributed to the inherent constraints of working with a lower resolution, which can limit the ability to capture intricate facial features and nuances.

Despite the limitations imposed by the lower resolution, the style mixing technique employed in the project yielded interesting results. By mixing the styles of the first and second images with the last image, the model was able to generate hybrid portraits that combine characteristics from different styles.

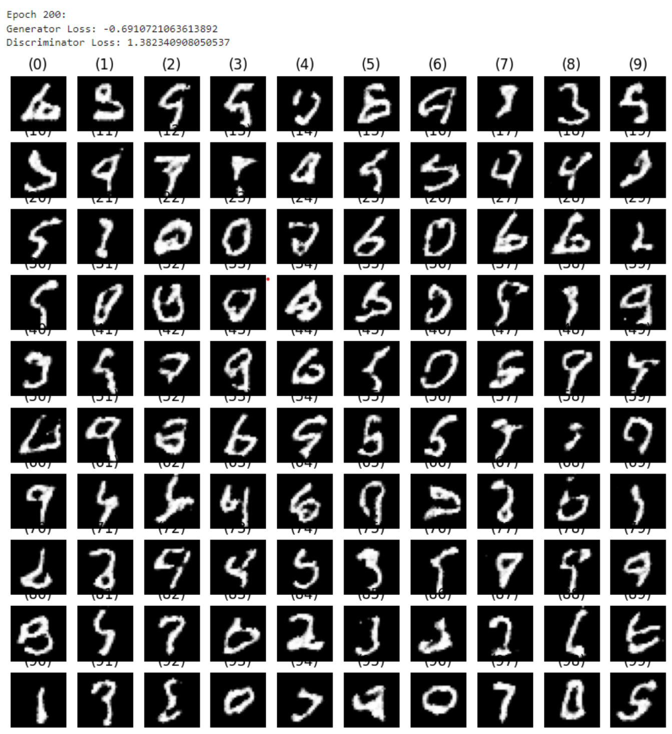
Future iterations of the project could explore higher-resolution images with increased computational power to further enhance the level of detail and realism in the generated portraits.

Mixed image of 64*64 resolution

ANALYSIS/EVALUATION

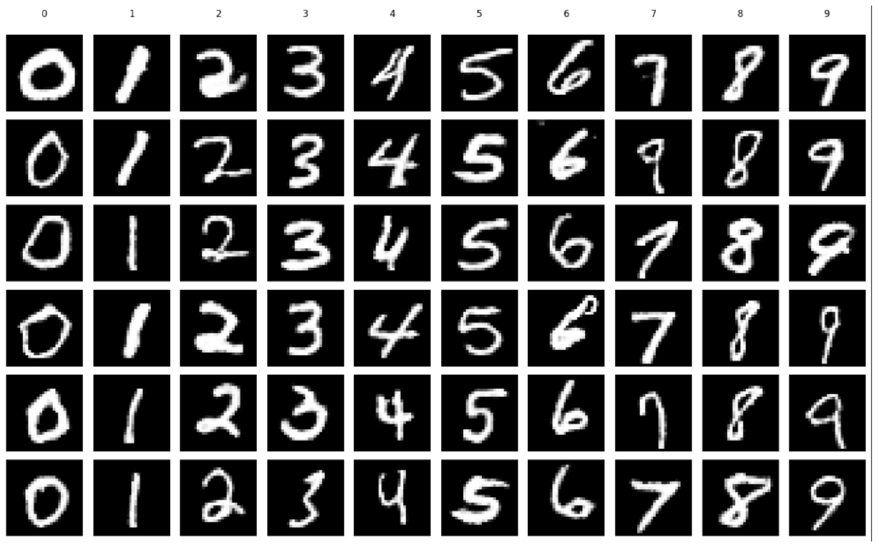
The training is performed over 100 epochs, and within each epoch, the data is divided into batches. For batchs of input data and corresponding target values and performing a single optimization step on the model using that batch of data. It updates the model's parameters based on the calculated loss and the adam optimizer algorithm. It trainis the discriminator and generator models simultaneously using batches of real and fake images along with their respective labels.

With a learning rate of 0.0001, sigmoid activation, a kernel size of 3, and a stride of 2. In this setting, the generated images showed less accuracy. Setting with a learning rate of 0.0003, LeakyReLU activation, a kernel size of 4, and a stride of 3. In this setting, the generated images exhibited improved clarity and were more recognizable as digits. The higher learning rate and the use of the LeakyReLU activation function contributed to this enhancement in image quality. The evaluation also considered the training time required for each setting. The setting with a learning rate of 0.0003, LeakyReLU activation, and a larger kernel size took longer to train, approximately 3.4 hours. However, this additional training time resulted in the generation of clearer and more visually appealing images. Based on the analysis, it can be concluded that the second set with a learning rate of 0.0003, LeakyReLU activation, a kernel size of 4, and a stride of 3 worked better in terms of generating clearer and more distinct digit images. This configuration resulted in improved performance and yielded more satisfying results.



```
Epoch 200:
Generator Loss: -0.6910721063613892
Discriminator Loss: 1.382340908050537
```

Image genaraion without condional gan in assignment



Condtional gan output

StyleGAN

Latent space mixing: The code snippet demonstrates the process of mixing two latent vectors (w[0] and w[1]) using a mixing factor of 0.4 and 0.6, respectively, to create a new latent vector (mix_style). This mixing operation allows for blending the styles of the two original latent vectors and generating a new image with a combination of their attributes. The model resolution is set to 64x64. It's important to note that the resolution of the generated images can impact their visual quality and level of detail. Higher-resolution images often require more computational resources and longer training times. The training was conducted for 5000 epochs, consisting of a transition phase and a stable phase. During the transition phase, the model goes through a gradual increase in resolution, starting from 4x4 and targeting 128x128. The stable phase implies that the model's resolution remains constant at 128x128 for the remainder of the training process. Negative values of the discriminator loss indicate that the discriminator is performing well in distinguishing between real and fake samples, and the generator loss being positive indicates that the generator is generating plausible samples that the discriminator has difficulty classifying as fake.

CONCLUSION

A variation on the GAN model, CGAN (Conditional Generative Adversarial Network) adds conditional information to the generator and discriminator. Additional conditioning factors, like class labels or input photos, can help CGAN produce more precise and in-control results. The CGANs showcased the potential for controlled image synthesis based on specific conditions.

Style Gan can create realistic images in a variety of styles by learning from huge data sets. It operates by mapping random noise vectors to a learned latent space, which controls the visual characteristics of the generated images. Any mathematical method, such as interpolating between two latent vectors, can change the style of the image by adjusting this vector. To assure the stability and quality of the output images, StyleGAN also uses methods like feature vector normalization and progressive growth. The project highlighted the importance of conditioning and latent space manipulation in generating desired and visually appealing images.

CREATIVE CONTRIBUTIONS

Embedded layers in the conditional GAN enable the generator to generate digit images based on specific labels or desired digit classes. This embedding allows the generator to learn a continuous representation of the labels, which is then be concatenated with other input vectors, such as noise, to generate images conditioned on specific labels. The embedded layers ensure that the label information is incorporated into the generator's input and influences the generated images without reducing the dimensionality of the label information. This allows the generator to understand and utilize the label information to generate images that correspond to the specified digit class. By incorporating the label information into the generation process, resulting in more controlled and targeted image synthesis.

The conditional GAN discriminator incorporating the MaxPooling2D layer, assists in downsampling the feature maps while maintaining information about the digit patterns in handwritten digit. By selecting the maximum value within each pooling window, MaxPooling2D helps in identifying the key features of each digit, such as the shape, orientation, and position of the strokes. This gives the discriminator the enables to concentrate on the digits' most discriminating features, enhancing its ability to distinguish between genuine and false samples. As the handwritten digit images in MNIST are composed of grayscale images with pixel values ranging from 0 to 255, the non-linearity introduced by LeakyReLU aids in capturing the intricate variations and nuances in the images. LeakyReLU prevents the problem of "dying ReLU" where gradients become less than zero by allowing a small negative slope for negative input values. LeakyReLU makes it possible for the model to recognize patterns, such as the curves and edges in the digits, and to more effectively deal with the noise and variability present in actual handwritten digits

Due to large variations in the data set of portraits, the discriminator may suffer from high sensitivity to the data to avoid this a Lipschitz constraint is added to prevent the discriminator too judgy from small changes in the input. Which also prevents unstable training. This a regularization technique that helps to keep the discriminator's gradients smooth and prevents them from exploding or vanishing during training. The generator receives more reliable and consistent feedback from the discriminator. This is achieved by adding a gradient penalty which is computed using the Wasserstein loss and the gradients of the discriminator's output with respect to the interpolated portraits (real portraits and fake portraits) and drift loss which is computed by taking the mean squared value of the discriminator loss. Both these values can be controlled by hyperparameters.

REFERENCES

[1] Kingma, D. P., & Welling, M. (2013). Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*.

[2] Van Den Oord, A., Kalchbrenner, N., & Kavukcuoglu, K. (2016, June). Pixel recurrent neural networks. In *International conference on machine learning* (pp. 1747-1756). PMLR.

[3] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., ... & Bengio, Y. (2020). Generative adversarial networks. *Communications of the ACM*, *63*(11), 139-144.

[4] Matas, J., Sebe, N., Welling, M., & Leibe, B. (Eds.). (2016, September 17). *Computer Vision - ECCV 2016: 14th European Conference, Amsterdam, the Netherlands, October 11-14, 2016, Proceedings* (Vol. 9907). https://doi.org/10.1007/978-3-319-46487-9

[5] Mirza, M., & Osindero, S. (2014). Conditional generative adversarial nets. arXiv preprint arXiv:1411.1784.

[6] Barron, J. T., & Poole, B. (2016). The fast bilateral solver. In *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part III 14* (pp. 617-632). Springer International Publishing.

[7] Team, K. (n.d.). *Keras documentation: Generative Deep Learning*. Generative Deep Learning. https://keras.io/examples/generative/

[8] Dobilas, S. (2022, October 15). *cGAN: Conditional Generative Adversarial Network—How to Gain Control Over GAN Outputs*. Medium. https://towardsdatascience.com/cgan-conditional-generative-adversarial-network-how-to-gain-control-over-gan-outputs-b30620bd0cc8

[9] Lauw, H. W., Wong, R. C. W., Ntoulas, A., Lim, E. P., Ng, S. K., & Pan, S. J. (Eds.). (2020). *Advances in Knowledge Discovery and Data Mining: 24th Pacific-Asia Conference, PAKDD 2020, Singapore, May 11-14, 2020, Proceedings, Part II* (Vol. 12085). Springer Nature.