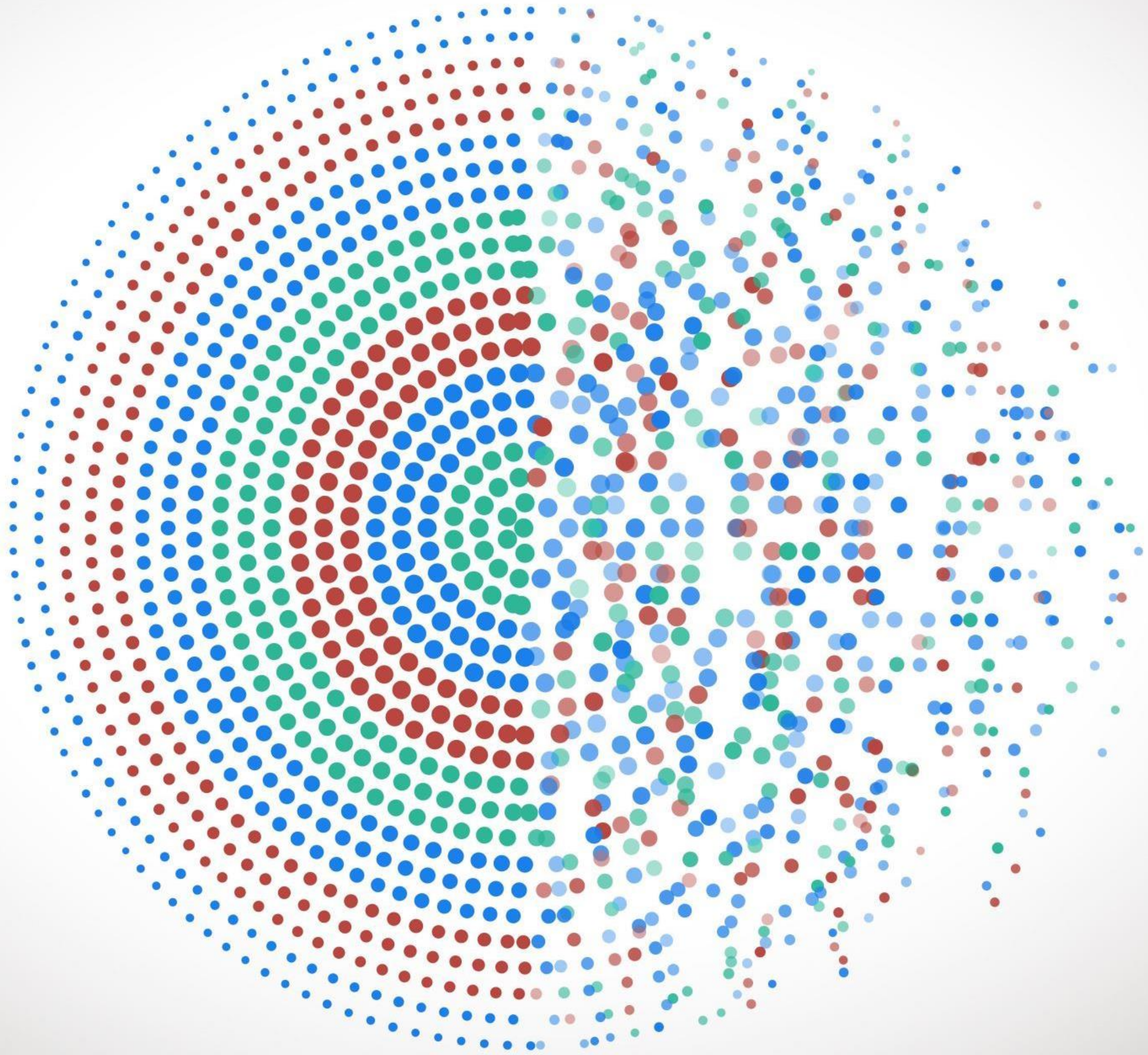
DELE CA2 PART A: GAN

By:

Loh Kong Xuan Kieran (p2309053) &

Soon Jing Yi (p2308940)



OVERVIEW

BACKGROUND RESEARCH

DATASET

The dataset given in this assignment is the EMNIST letters dataset, which is a subset of the Extended MNIST (EMNIST) dataset developed in 2017, which extends the original MNIST dataset to include more handwritten characters. It contains handwritten character digits converted to a 28x28 pixel image format that matches the structure of the MNIST dataset. The original EMNIST Letters dataset contained EMNIST 145,600 characters and 26 balanced classes.

APPLICATIONS OF IMAGE GENERATION

- Art and design
- Pre-visualization in film industry
- Data augmentation, augment training datasets for AI/ML models, improving their performance by providing more diverse and representative sample.
- Style transfer
- Image generation

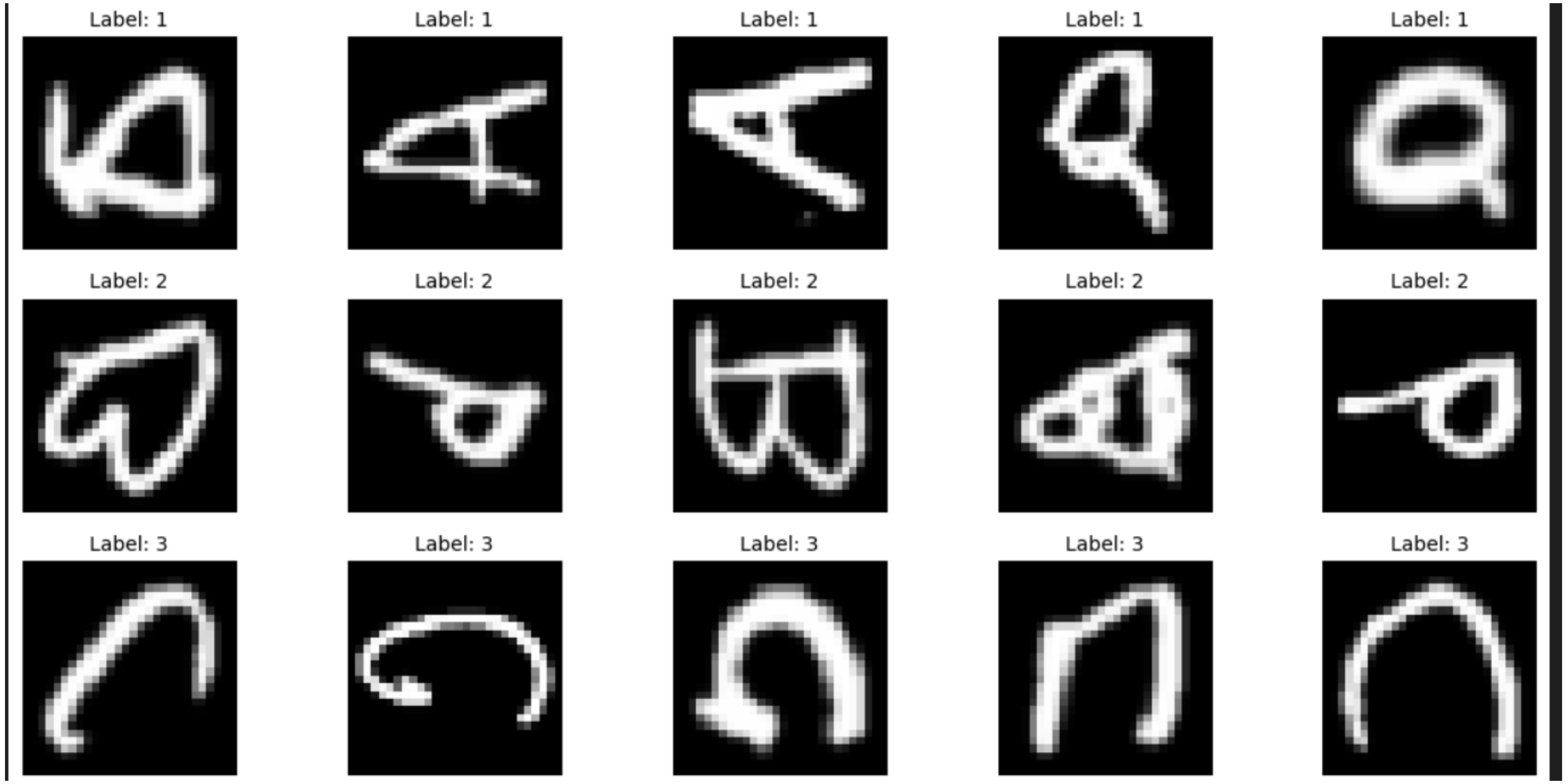
WHAT ARE GANS?

Generative Adversarial Networks (GANs) were introduced by Ian Goodfellow and other researchers in 2014. GANs consist of 2 neu ral networks, a generator and a discriminator, which contest with each other in a zero -sum game.

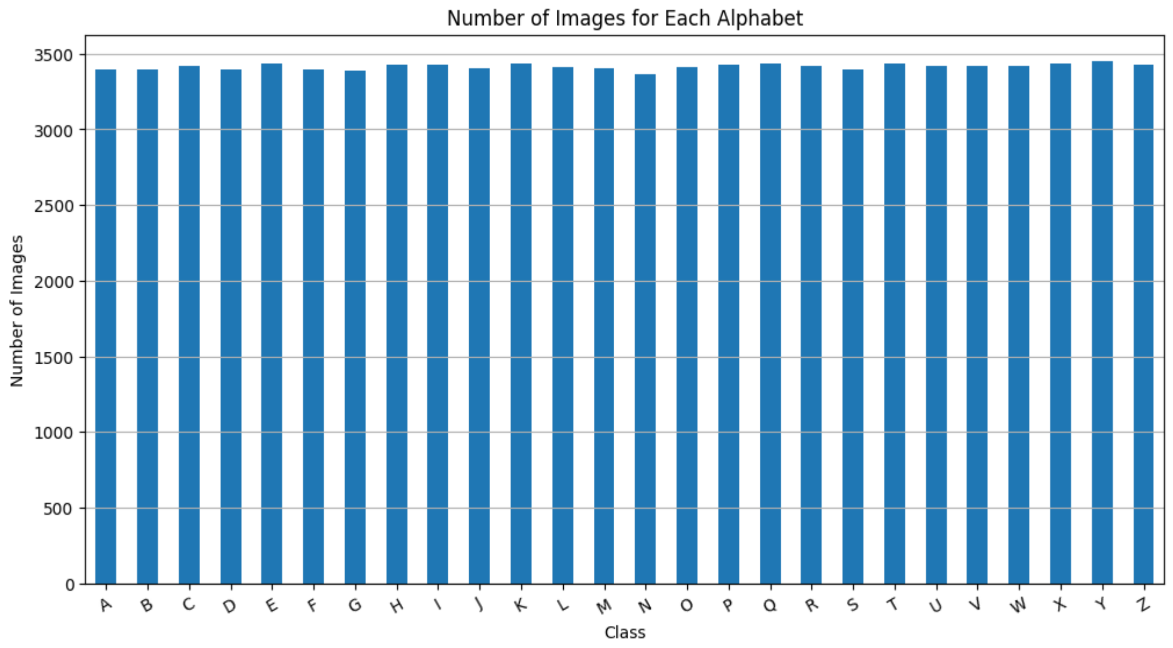
The generator creates the images from noise, while the discriminator evaluates it, aiming to distinguish between real and gen erated data. This adversarial process helps the generator improve its output to fool the discriminator, leading to the creation of h ighly realistic data.

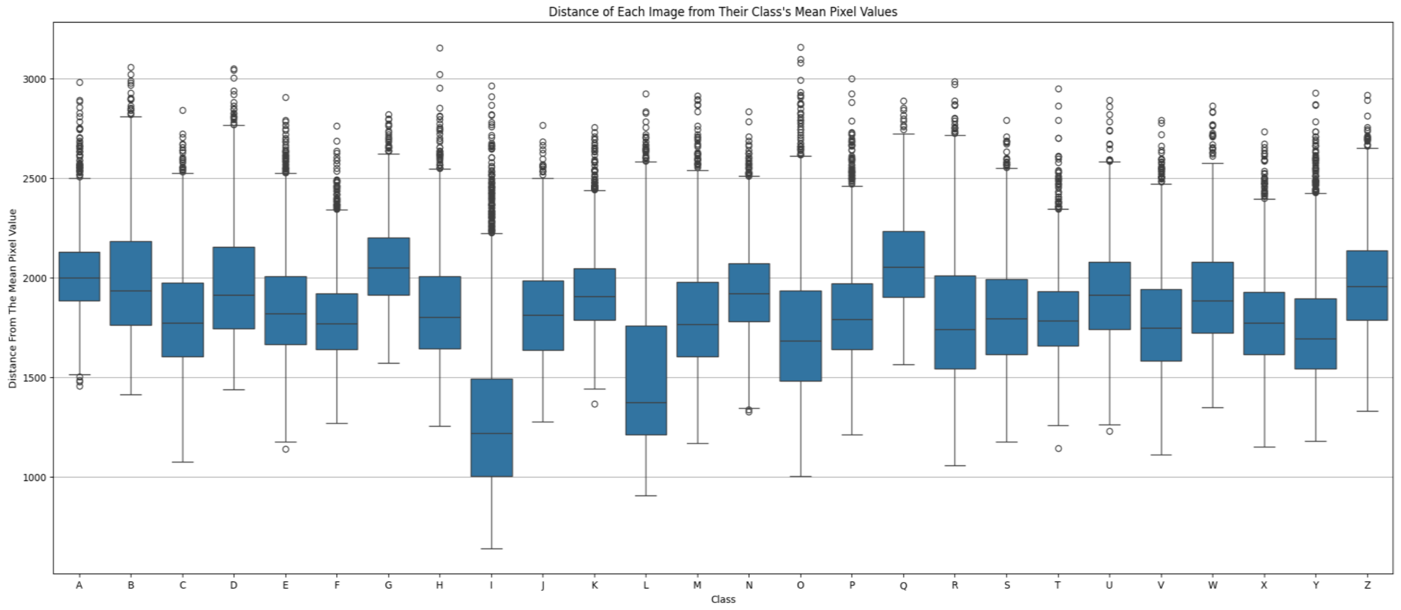
EDA

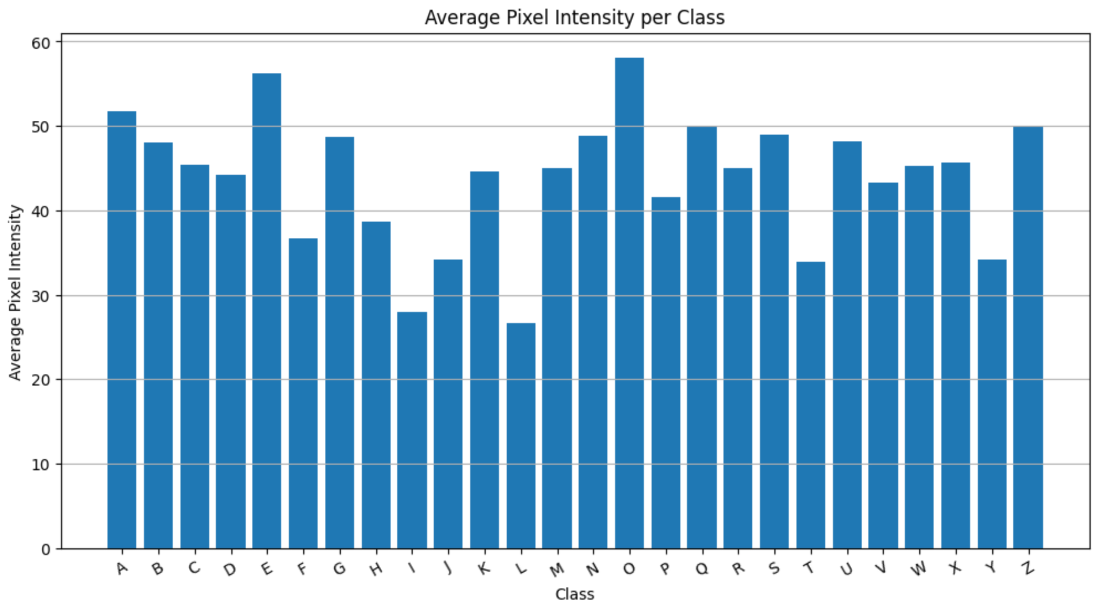
A sample of the dataset images:



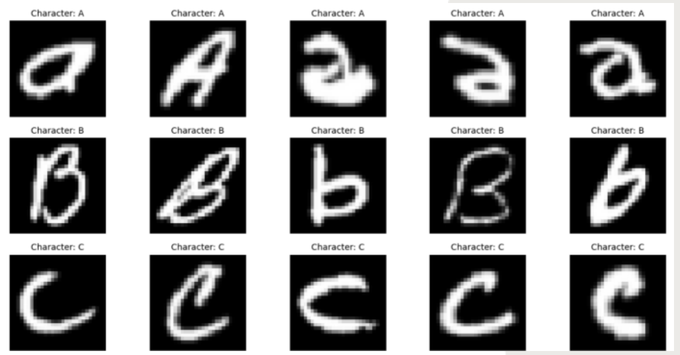
E D A


Number of Images for Each Alphabet
Number of Images
Class
0
500
1000
1500
2000
2500
3000
3500
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z


Distance of Each Image from Their Class's Mean Pixel Values
Distance From The Mean Pixel Value
Class
1000
1500
2000
2500
3000
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z


Average Pixel Intensity per Class
Average Pixel Intensity
Class
0
10
20
30
40
50
60
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

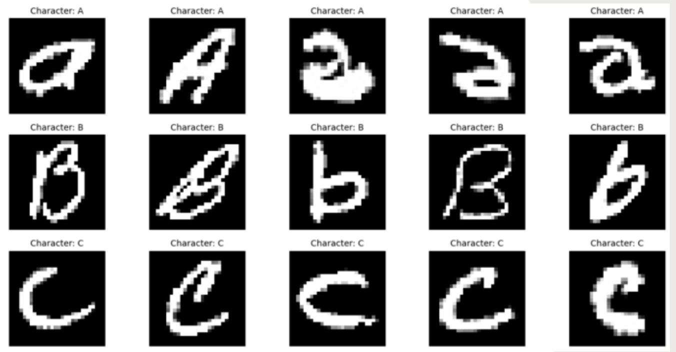
FEATURE ENGINEERING



Standard Preprocessing:

- Remove Outliers (images whose pixel values significantly deviate from the mean pixel values of their class. These outliers can skew the results and obscure meaningful patterns in the data)
- Fixing orientation of the images (rotate the images 90 degrees clockwise and flip horizontally)

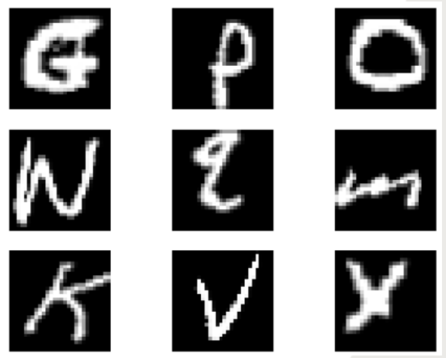
Further Preprocessing:



- Denoising the images (removing unwanted noise from the images)
- Sharpening the images (enhance the edges and details within an image, making the features more distinct)

Augmentation:

- Augment the images further by width and height shifting, shearing and filling.
- Enhanced model generalization
  - Increased data variability
  - Improved model robustness
  - Reduce overfitting
- Potential negative impacts
  - Loss of original image quality
  - Increased computational load
  - Risk of over augmentation
  - Potential decrease in GAN performance



MODEL BUILDING

1. DCGAN
2. CGAN
3. WGAN

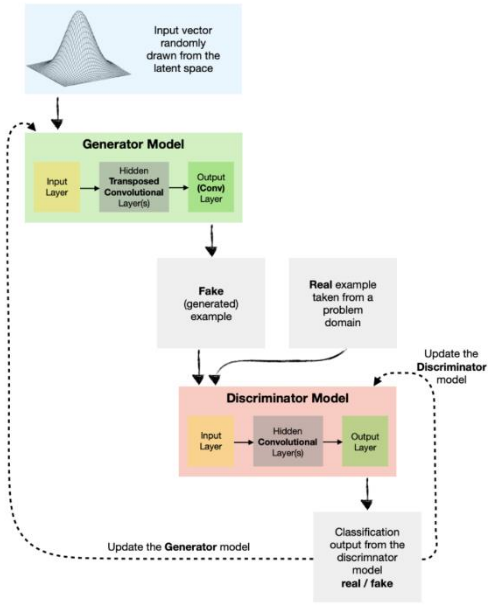
MODEL BUILDING (METRICES)

Explanation of metrices used in training

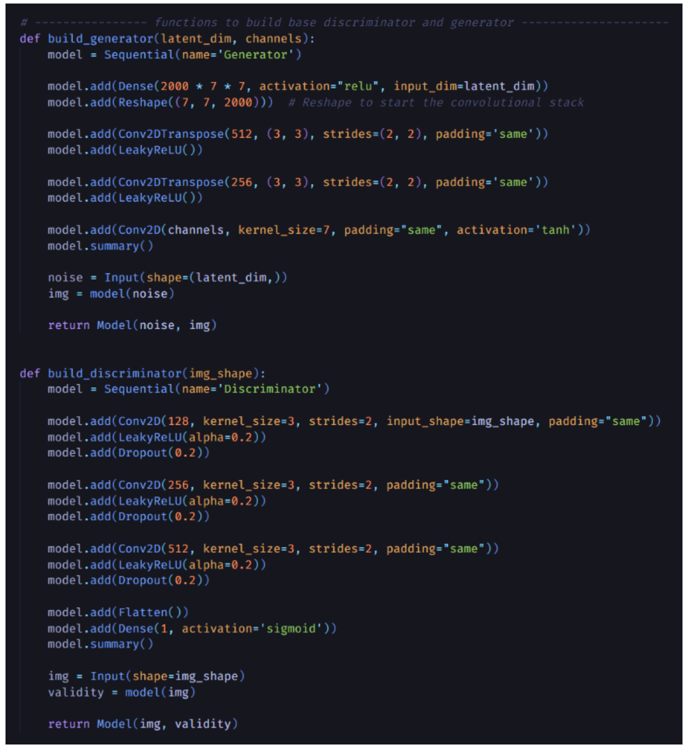
- Discriminator Loss measures how well the discriminator can distinguish between real and fake images, correctly classify real samples as real and fake samples (produced by the generator) as fake
- Generator Loss measures how well the generator is at producing images that can fool the discriminator, produce samples that the discriminator incorrectly classifies as real
- Discriminator Accuracy is the percentage of correctly classified images by the discriminator.
- KL Divergence quantifies the difference between two probability distributions, in this case between the distributions of real and generated images
- JS Divergence is a symmetric measure that also evaluates the difference between these distributions. KL Divergence is useful for understanding how one distribution diverges from another, JS Divergence provides a more balanced view of the divergence between two distributions.

How should the values of the metrices look like for an optimal training?

- Discriminator Loss the graph should stabilize at a low value over time. If the discriminator loss is too low or too high, it may indicate an imbalance where the discriminator is either too powerful or too weak relative to the generator.
- An optimal scenario is where both discriminator and generator losses stabilize and neither the generator nor the discriminator dominates the training process. This balance indicates that the GAN is learning effectively.
- Discriminator accuracy should ideally hover around **50%**. This indicates that the discriminator is finding it equally challenging to distinguish between real and fake images, suggesting that the generator is producing high-quality images.
- During optimal training, KL and JS divergence should decrease and stabilize, indicating that the distribution of generated images is becoming similar to the distribution of real images.


Input vector randomly drawn from the latent space
Generator Model
Input Layer
Hidden Transposed Convolutional Layer(s)
Output (Conv) Layer
Fake (generated) example
Real example taken from a problem domain
Update the Discriminator model
Discriminator Model
Input Layer
Hidden Convolutional Layer(s)
Output Layer
Classification output from the discrimnator model real / fake
Update the Generator model

https://www.linkedin.com/pulse/one-minute-overview-deep-convolutional-generative-networks-dobilas/



```
# ---------------- functions to build base discriminator and generator --------------------
def build_generator(latent_dim, channels):
    model = Sequential(name='Generator')

    model.add(Dense(2000 * 7 * 7, activation="relu", input_dim=latent_dim))
    model.add(Reshape((7, 7, 2000)))  # Reshape to start the convolutional stack

    model.add(Conv2DTranspose(512, (3, 3), strides=(2, 2), padding='same'))
    model.add(LeakyReLU())

    model.add(Conv2DTranspose(256, (3, 3), strides=(2, 2), padding='same'))
    model.add(LeakyReLU())

    model.add(Conv2D(channels, kernel_size=7, padding="same", activation='tanh'))
    model.summary()

    noise = Input(shape=(latent_dim,))
    img = model(noise)

    return Model(noise, img)


def build_discriminator(img_shape):
    model = Sequential(name='Discriminator')

    model.add(Conv2D(128, kernel_size=3, strides=2, input_shape=img_shape, padding="same"))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.2))

    model.add(Conv2D(256, kernel_size=3, strides=2, padding="same"))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.2))

    model.add(Conv2D(512, kernel_size=3, strides=2, padding="same"))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.2))

    model.add(Flatten())
    model.add(Dense(1, activation='sigmoid'))
    model.summary()

    img = Input(shape=img_shape)
    validity = model(img)

    return Model(img, validity)
```

MODEL BUILDING (DCGAN)

DCGANs consist of two main components:

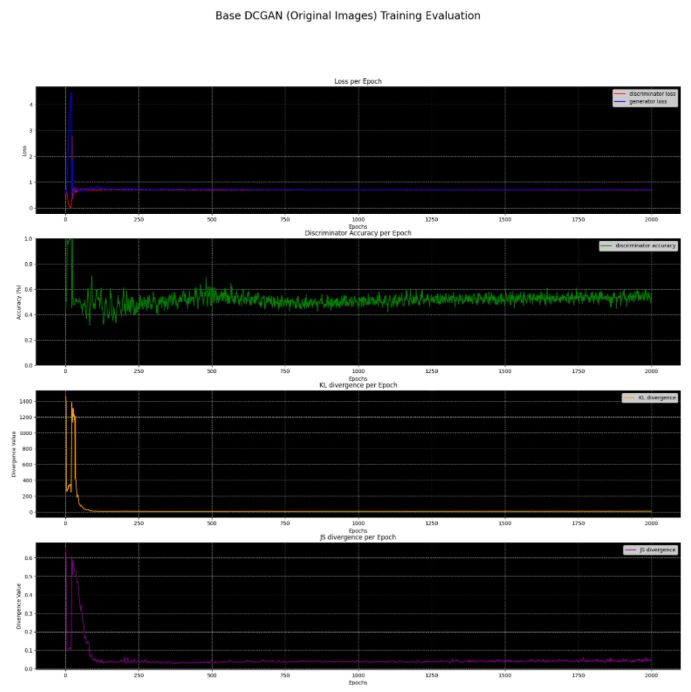
Generator

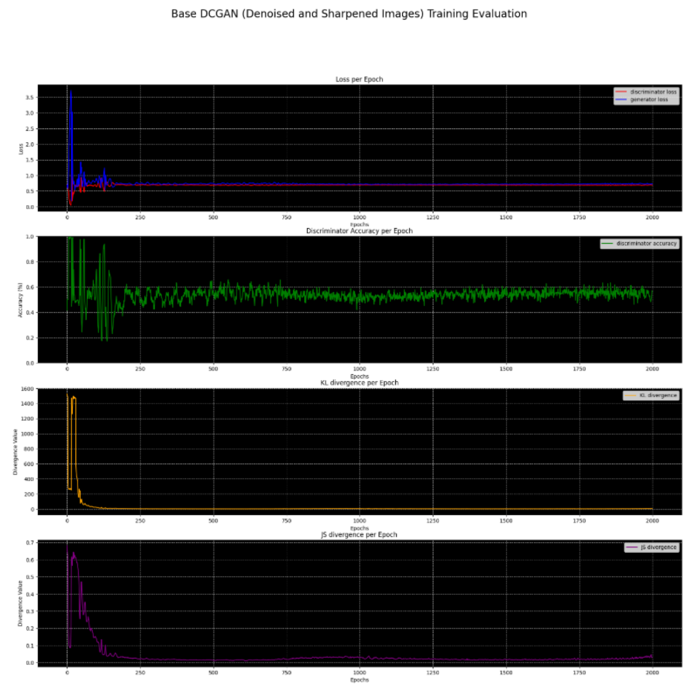
- Input: Takes a random noise vector (latent space) as input.
- Layers: Composed of transposed convolutional layers (also known as deconvolutional layers), which aims to upsample the input noise, resulting in a larger output feature map which would eventually be the generated image. (https://www.linkedin.com/pulse/one-minute-overview-deep-convolutional-generative-networks-dobilas/)
- Output: Produces an image that mimics the training data, passed through a Tanh activation function to map the output to the range [-1, 1]. This matches the range of the pixel values of the input images, which are also normalized to [−1,1]. This ensures consistency between the generated and real images.

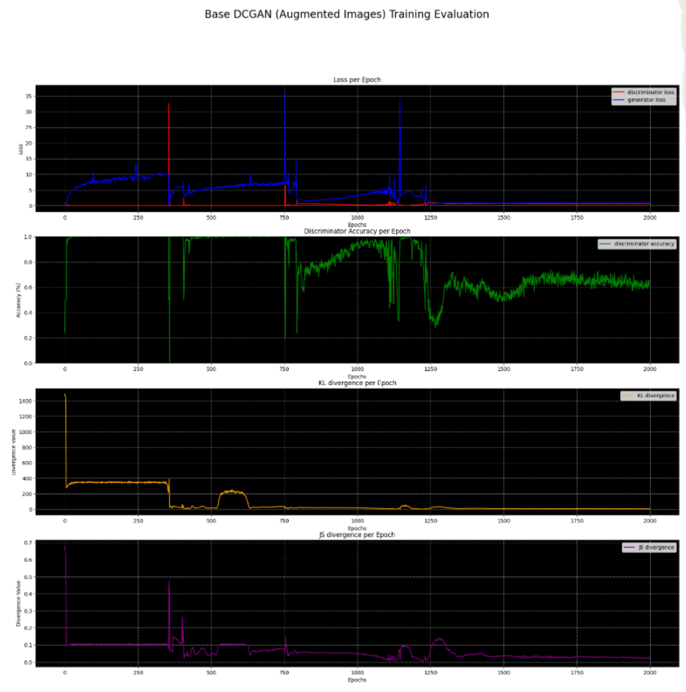
Discriminator

- Input: Takes an image (either real or generated) as input.
- Layers: Consists of strided convolutional layers to downsample the images
- Output: Outputs a probability through a Sigmoid activation function and through cross entropy, outputting a binary classification of whether the input image is real or fake.

MODEL BUILDING (DCGAN)







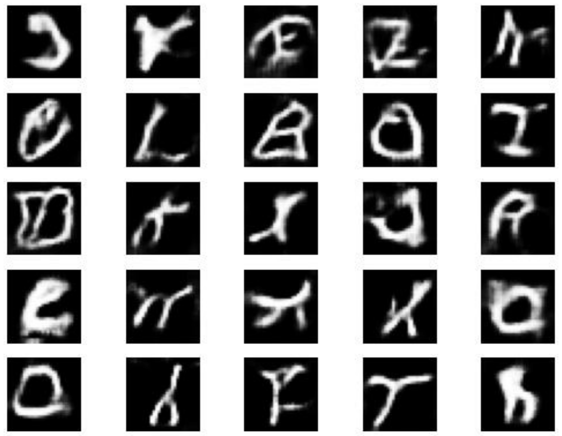
We trained the base DCGAN on the 3 different datasets separately to evaluate whether augmentations are beneficial.

- Overall, the **model fitted with the original images has the most stable training** based on the loss curve and discriminator accuracy, which converged and stabilized the quickest with the least fluctuations.
- The model fitted with the **augmented images is the least stable**, as seen by the fluctuations and slow convergence in both the loss discriminator accuracy graphs. The KL and JS divergences also decreases, but it fluctuates wildly in some instances as well.
- Training on the augmented images introduces challenges due to distortions like height, width, shearing, and filling, making it harder for the DCGAN to learn effectively. This results in less stable training, poorer performance, and more difficulty for the generator and discriminator to adapt, leading to higher losses, fluctuating accuracies, and inconsistent divergences compared to training on original images.

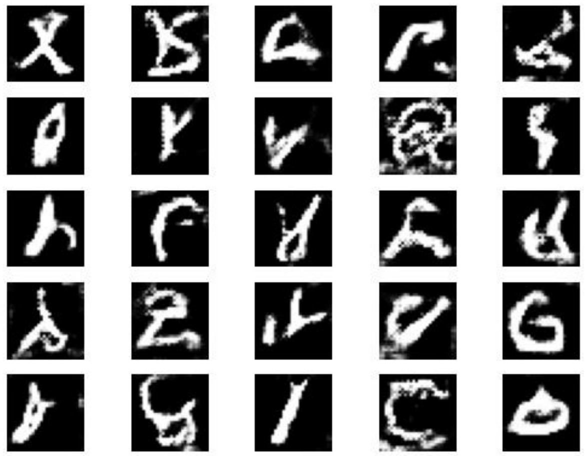
MODEL BUILDING (DCGAN)

Generated Images by the DCGANs at 2000 Epoch

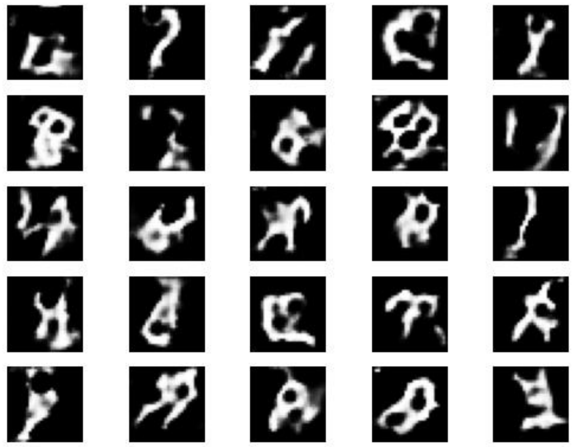
trained on original images



trained on sharpened images



trained on augmented images



Trained on original images

- The generated characters are smoother and more closely resemble natural handwriting. Most of the characters are easily recognizable.

Trained on sharpened images

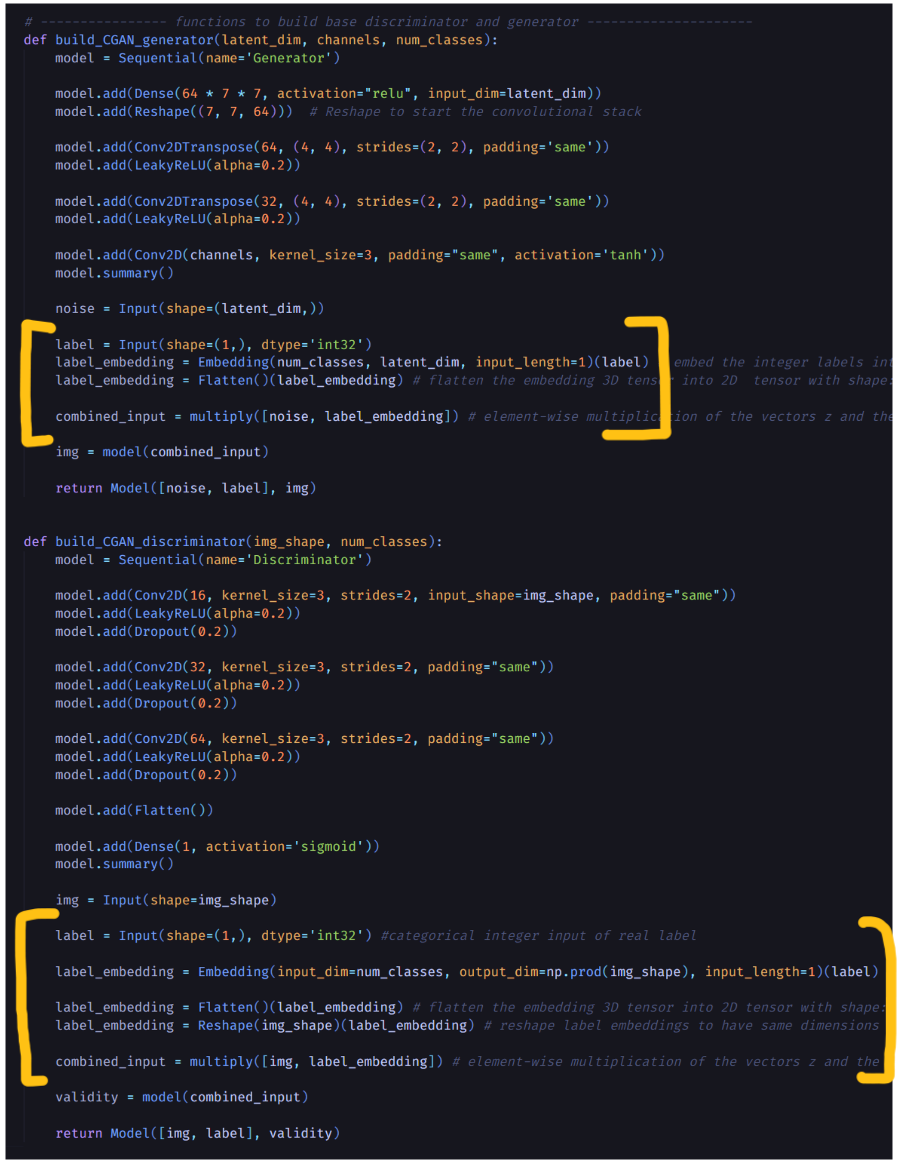
- The generated characters are the sharpest, with a clear distinction between white and black pixels. However, they appear less natural and more "computer generated". Most characters are also recognizable.

Trained on augmented images

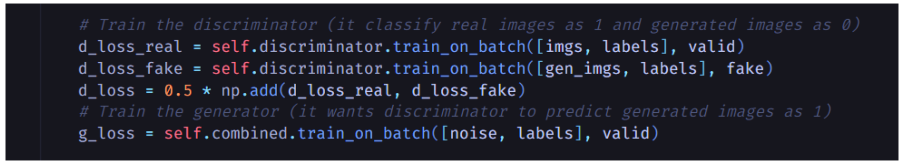
- The generated characters are messy and unrecognizable, with more noise compared to the other two sets. The additional variability from augmentation made it more difficult for the DCGAN to learn consistent features.

Overall, we **chose to use models trained on original images dataset from this point onwards**, as it provided the most stable and efficient training, and the images generated more closely resemble natural handwriting.

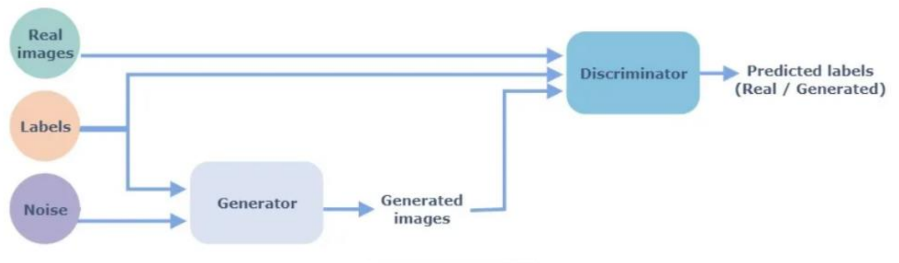
MODEL BUILDING (CGAN)



```
# ---------------- functions to build base discriminator and generator ---------------------
def build_CGAN_generator(latent_dim, channels, num_classes):
    model = Sequential(name='Generator')

    model.add(Dense(64 * 7 * 7, activation="relu", input_dim=latent_dim))
    model.add(Reshape((7, 7, 64)))  # Reshape to start the convolutional stack

    model.add(Conv2DTranspose(64, (4, 4), strides=(2, 2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))

    model.add(Conv2DTranspose(32, (4, 4), strides=(2, 2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))

    model.add(Conv2D(channels, kernel_size=3, padding="same", activation='tanh'))
    model.summary()

    noise = Input(shape=(latent_dim,))

    label = Input(shape=(1,), dtype='int32')
    label_embedding = Embedding(num_classes, latent_dim, input_length=1)(label)  # embed the integer labels int
    label_embedding = Flatten()(label_embedding) # flatten the embedding 3D tensor into 2D  tensor with shape:

    combined_input = multiply([noise, label_embedding]) # element-wise multiplication of the vectors z and the

    img = model(combined_input)

    return Model([noise, label], img)


def build_CGAN_discriminator(img_shape, num_classes):
    model = Sequential(name='Discriminator')

    model.add(Conv2D(16, kernel_size=3, strides=2, input_shape=img_shape, padding="same"))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.2))

    model.add(Conv2D(32, kernel_size=3, strides=2, padding="same"))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.2))

    model.add(Conv2D(64, kernel_size=3, strides=2, padding="same"))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.2))

    model.add(Flatten())

    model.add(Dense(1, activation='sigmoid'))
    model.summary()

    img = Input(shape=img_shape)

    label = Input(shape=(1,), dtype='int32') #categorical integer input of real label

    label_embedding = Embedding(input_dim=num_classes, output_dim=np.prod(img_shape), input_length=1)(label)

    label_embedding = Flatten()(label_embedding) # flatten the embedding 3D tensor into 2D tensor with shape:
    label_embedding = Reshape(img_shape)(label_embedding) # reshape label embeddings to have same dimensions

    combined_input = multiply([img, label_embedding]) # element-wise multiplication of the vectors z and the

    validity = model(combined_input)

    return Model([img, label], validity)
```



```
    # Train the discriminator (it classify real images as 1 and generated images as 0)
    d_loss_real = self.discriminator.train_on_batch([imgs, labels], valid)
    d_loss_fake = self.discriminator.train_on_batch([gen_imgs, labels], fake)
    d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)
    # Train the generator (it wants discriminator to predict generated images as 1)
    g_loss = self.combined.train_on_batch([noise, labels], valid)
```



Conditional GAN

https://datascientest.com/en/what-is-a-conditional-generative-adversarial-network-cgan

The overall concept of CGAN is similar to DCGAN in terms of architecture, compilation and training, however, for CGAN, **labels are inputted to both the generator and discriminator**.

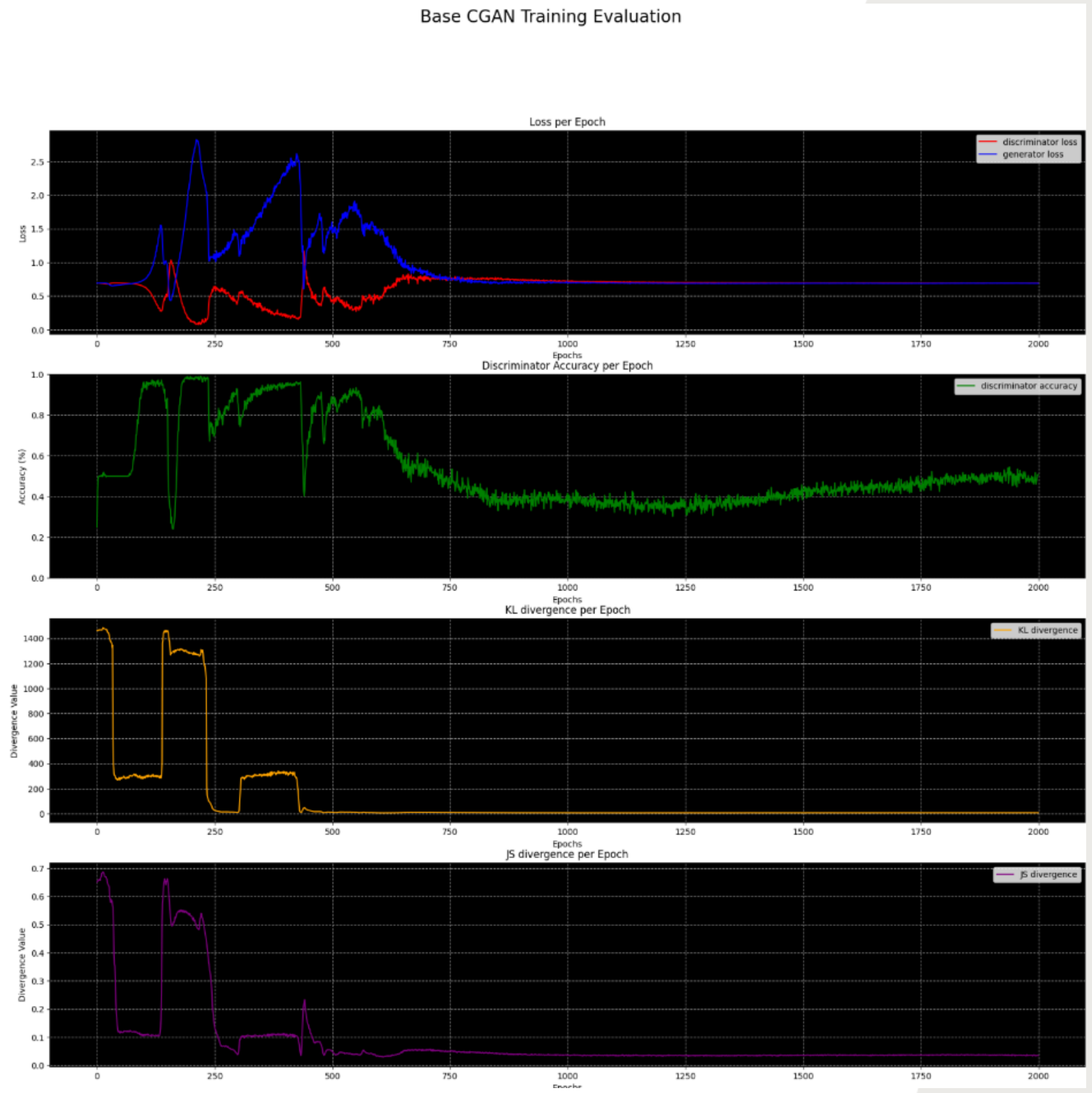
- The generator receives a latent noise vector and a class label as inputs. **The class label is embedded into a dense vector and then multiplied element-wise with the latent noise vector**. This combined vector is used to **generate images that corresponds to the specified class label**.
- On the other hand, the discriminator uses the class labels to improve its ability to distinguish between real and generated images of the specific class. The label is embedded and then multiplied element-wise with the input image. **By knowing the class information, the discriminator can better evaluate whether an image belongs to the correct class or not.** This makes the discriminator more efficient at identifying fake images, as it has additional context, and **encourages the generator to improve on generating class specific features.**
- Other than the ability to generate images of a specific class, CGANs have other benefits compared to DCGANs such as improved performance in generating higher quality outputs since the additional conditional information (label) helps guide the generator more effectively, resulting in more consistent and coherent images (https://www.linkedin.com/pulse/conditional-generative-adversarial-network-cgan-madhavan-vivekanandan-nae7c/)

MODEL BUILDING (CGAN)

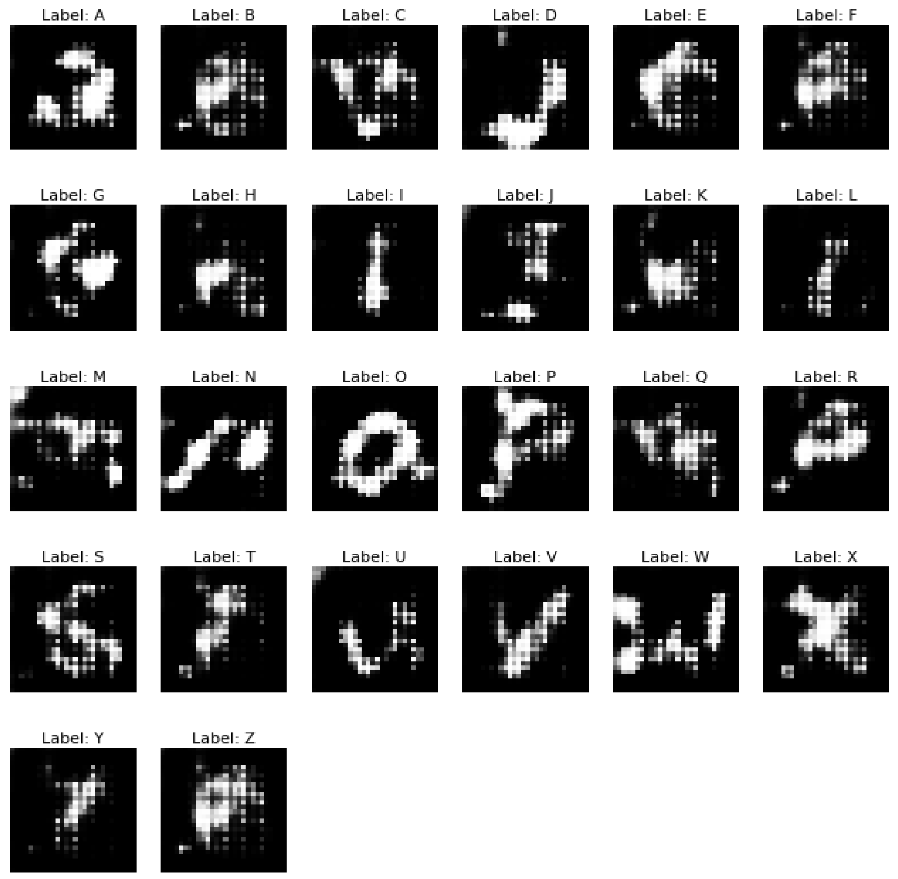
During training, the **generator initially exhibited higher loss compared to the discriminator**, with fluctuations in both losses. This trend indicated that the generator was struggling more than the discriminator. **Over time, both losses stabilized and converged,** reflecting a balance between the generator and discriminator, signifying successful training.

Discriminator accuracy showed an initial fluctuation **but reached equilibrium at around 0.5** after a period of imbalance. This pattern suggests that the CGAN eventually achieved a balanced state where both the generator and discriminator performed equally.

The **KL divergence decreased and stabilized**, indicating that the generated images increasingly resembled the real data distribution. Similarly, the JS divergence dropped to near zero, further suggesting that the generator effectively learned to produce images that are close to the real ones.



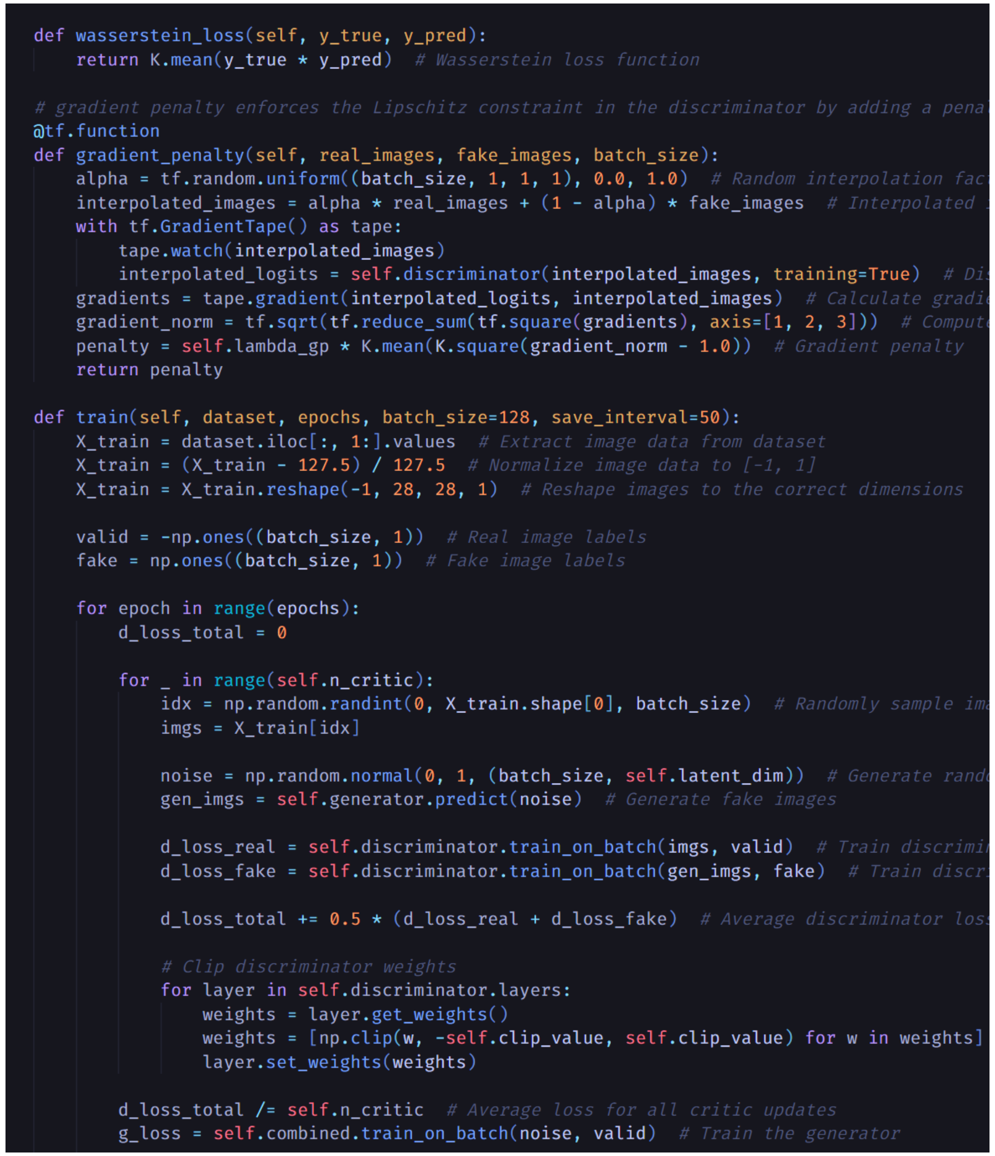
MODEL BUILDING (CGAN)



The base CGAN was able to **generate images of each class.**

- There are some recognizable characters for each letter.
  - For example, generated images for P, S, V and W are recognizable and are correctly corresponded
- The overall **quality of the images are still noisy and there are checkerboard artifacts**. This might be due to uneven overlaps in the spatial dimension during transposed convolutions. It can also be due to insufficient network capacity.

MODEL BUILDING (WGAN)



```python
def wasserstein_loss(self, y_true, y_pred):
    return K.mean(y_true * y_pred)  # Wasserstein loss function

# gradient penalty enforces the Lipschitz constraint in the discriminator by adding a pena
@tf.function
def gradient_penalty(self, real_images, fake_images, batch_size):
    alpha = tf.random.uniform((batch_size, 1, 1, 1), 0.0, 1.0)  # Random interpolation fac
    interpolated_images = alpha * real_images + (1 - alpha) * fake_images  # Interpolated
    with tf.GradientTape() as tape:
        tape.watch(interpolated_images)
        interpolated_logits = self.discriminator(interpolated_images, training=True)  # Di
    gradients = tape.gradient(interpolated_logits, interpolated_images)  # Calculate gradi
    gradient_norm = tf.sqrt(tf.reduce_sum(tf.square(gradients), axis=[1, 2, 3]))  # Compute
    penalty = self.lambda_gp * K.mean(K.square(gradient_norm - 1.0))  # Gradient penalty
    return penalty

def train(self, dataset, epochs, batch_size=128, save_interval=50):
    X_train = dataset.iloc[:, 1:].values  # Extract image data from dataset
    X_train = (X_train - 127.5) / 127.5  # Normalize image data to [-1, 1]
    X_train = X_train.reshape(-1, 28, 28, 1)  # Reshape images to the correct dimensions

    valid = -np.ones((batch_size, 1))  # Real image labels
    fake = np.ones((batch_size, 1))  # Fake image labels

    for epoch in range(epochs):
        d_loss_total = 0

        for _ in range(self.n_critic):
            idx = np.random.randint(0, X_train.shape[0], batch_size)  # Randomly sample ima
            imgs = X_train[idx]

            noise = np.random.normal(0, 1, (batch_size, self.latent_dim))  # Generate rand
            gen_imgs = self.generator.predict(noise)  # Generate fake images

            d_loss_real = self.discriminator.train_on_batch(imgs, valid)  # Train discrimi
            d_loss_fake = self.discriminator.train_on_batch(gen_imgs, fake)  # Train discr

            d_loss_total += 0.5 * (d_loss_real + d_loss_fake)  # Average discriminator loss

            # Clip discriminator weights
            for layer in self.discriminator.layers:
                weights = layer.get_weights()
                weights = [np.clip(w, -self.clip_value, self.clip_value) for w in weights]
                layer.set_weights(weights)

        d_loss_total /= self.n_critic  # Average loss for all critic updates
        g_loss = self.combined.train_on_batch(noise, valid)  # Train the generator
```

WGAN is an enhanced version of the traditional GAN. Unlike conventional GANs, which use a discriminator to classify generated images as either real or fake, WGAN replaces the discriminator with a **critic that assigns a score** reflecting the authenticity of an image.

The Wasserstein loss calculates the mean product of the truth labels, and the score outputted by the discriminator. In the discriminator, since the real labels are -1, to minimize the loss, it **scores real images higher**. On the other hand, since fake labels are 1, to minimize the loss, it **scores fake images lower**.

The **generator aims to increase the discriminator's score for generated images**, thus making them more similar to real images.

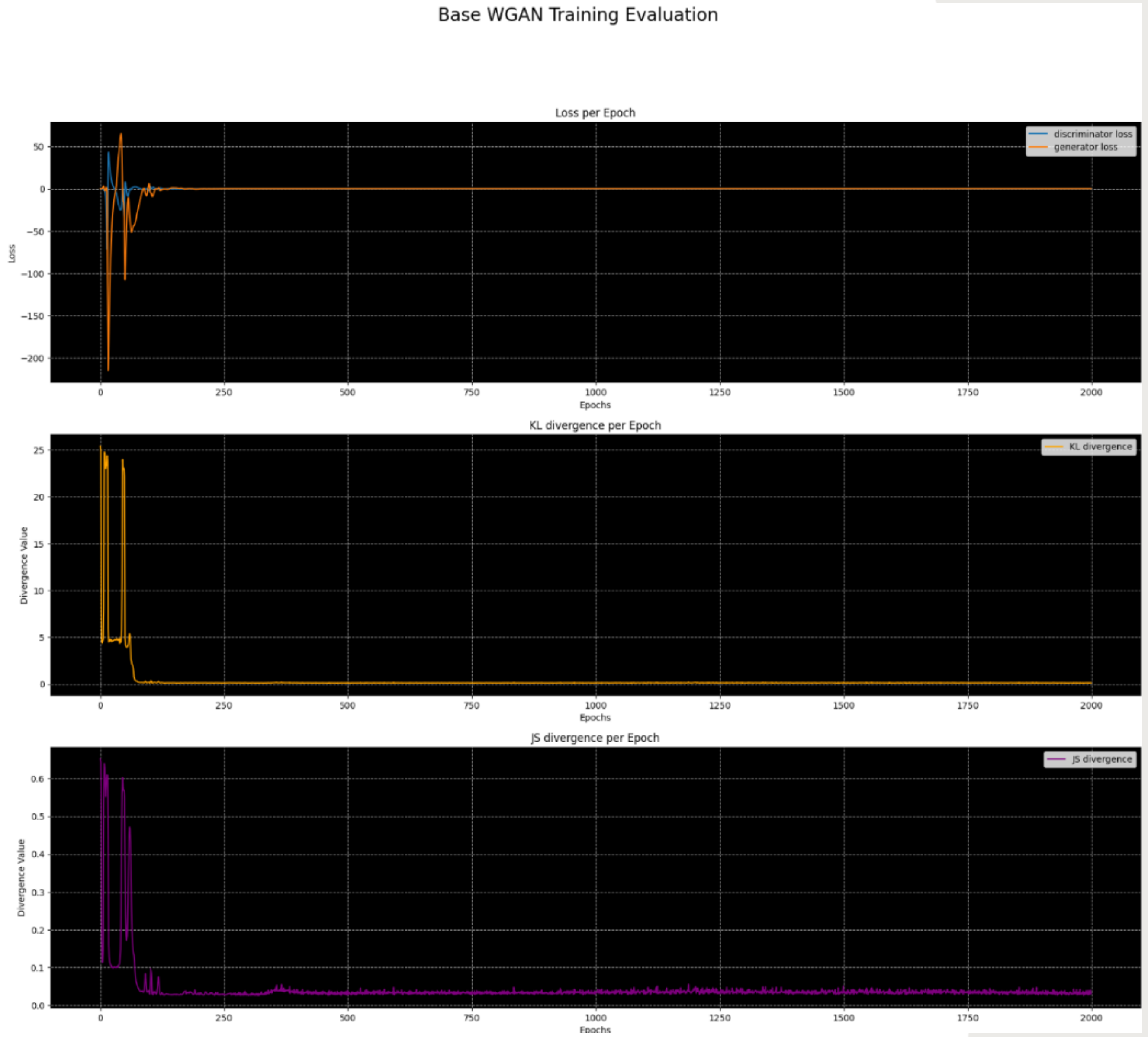
WGAN also incorporates **weight clipping** in the critic's function, which **prevents extreme gradients**, leading to more stable and reliable updates during training.

Overall, it offers a **more stable training process** and is less sensitive to model architecture and hyperparameter settings.

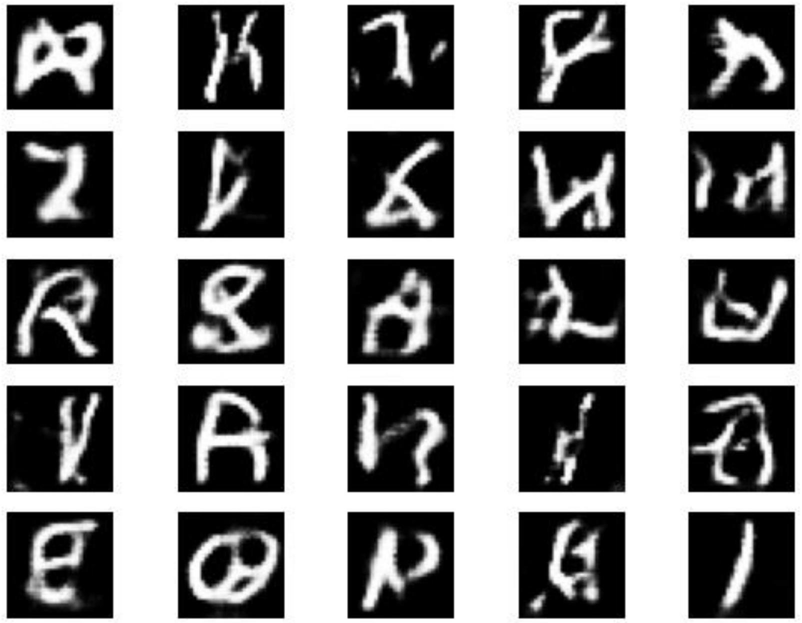
MODEL BUILDING (WGAN)

Overall, the training duration took longer than DCGAN and CGAN at around **50 minutes** in total, compared to around 10 minutes.

As compared to the DCGAN and CGAN, we can see that the **training curves are much more stable and converges faster**. We can see that in all 3 graphs, they converge and stabilize at around **100 epochs**. This means that the WGAN has a high training efficiency and stability.



MODEL BUILDING (WGAN)



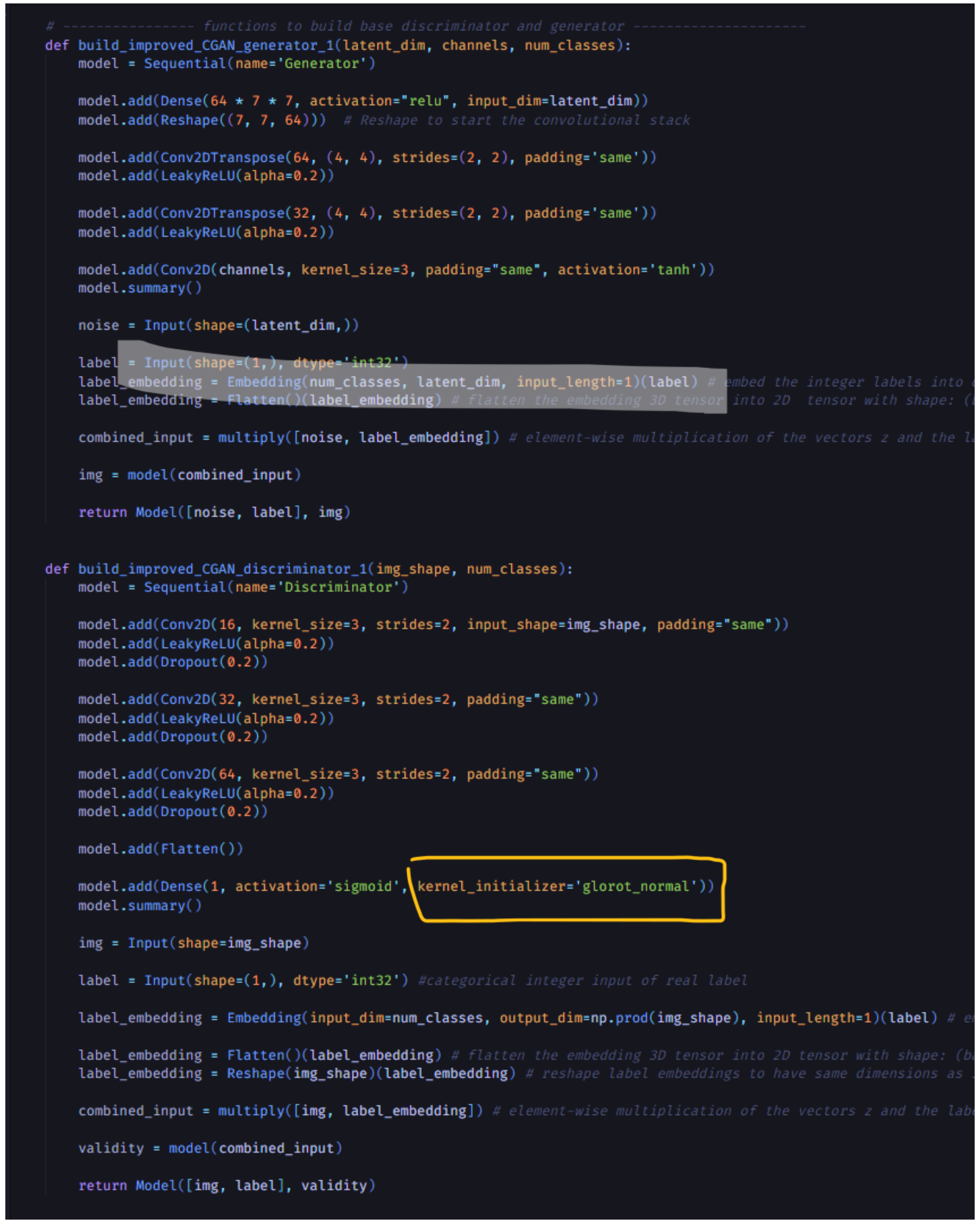
The model has managed to **produce lines that somewhat mimic handwriting, however, many of the characters remain unrecognizable**. This indicates that while the WGAN has learned some features of handwritten characters, but further training or model adjustments may be necessary to enhance the clarity and accuracy of the generated characters

MODEL IMPROVEMENT

- We decide to **focus on CGAN** as it has a better potential in generating images compared to GANs and WGANs, due to its ability to generated specific images, as well as the advantages of having extra information from the labels.

- Furthermore, it takes relatively less time to train a CGAN model as compared to WGAN in our case.

MODEL IMPROVEMENT 1(KERNEL INITIALIZER)



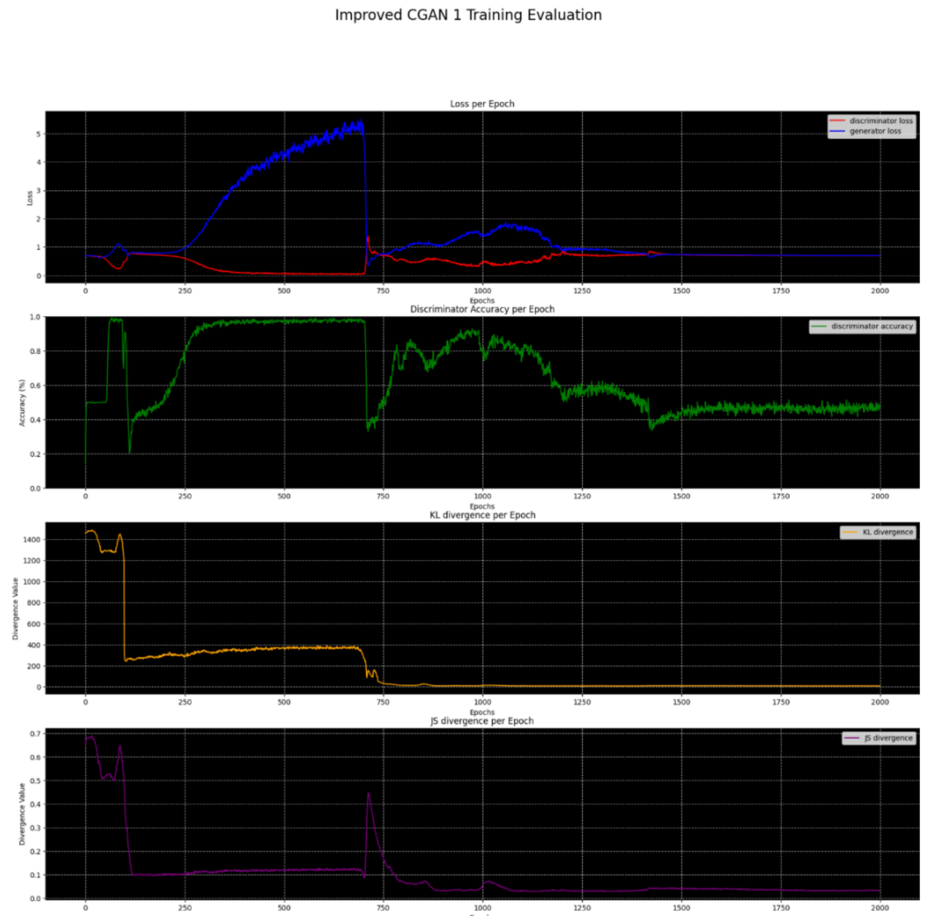
```python
# ---------------- functions to build base discriminator and generator ---------------------
def build_improved_CGAN_generator_1(latent_dim, channels, num_classes):
    model = Sequential(name='Generator')

    model.add(Dense(64 * 7 * 7, activation="relu", input_dim=latent_dim))
    model.add(Reshape((7, 7, 64)))  # Reshape to start the convolutional stack

    model.add(Conv2DTranspose(64, (4, 4), strides=(2, 2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))

    model.add(Conv2DTranspose(32, (4, 4), strides=(2, 2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))

    model.add(Conv2D(channels, kernel_size=3, padding="same", activation='tanh'))
    model.summary()

    noise = Input(shape=(latent_dim,))

    label = Input(shape=(1,), dtype='int32')
    label_embedding = Embedding(num_classes, latent_dim, input_length=1)(label) # embed the integer labels into 
    label_embedding = Flatten()(label_embedding) # flatten the embedding 3D tensor into 2D  tensor with shape: (

    combined_input = multiply([noise, label_embedding]) # element-wise multiplication of the vectors z and the l

    img = model(combined_input)

    return Model([noise, label], img)


def build_improved_CGAN_discriminator_1(img_shape, num_classes):
    model = Sequential(name='Discriminator')

    model.add(Conv2D(16, kernel_size=3, strides=2, input_shape=img_shape, padding="same"))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.2))

    model.add(Conv2D(32, kernel_size=3, strides=2, padding="same"))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.2))

    model.add(Conv2D(64, kernel_size=3, strides=2, padding="same"))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.2))

    model.add(Flatten())

    model.add(Dense(1, activation='sigmoid', kernel_initializer='glorot_normal'))
    model.summary()

    img = Input(shape=img_shape)

    label = Input(shape=(1,), dtype='int32') #categorical integer input of real label

    label_embedding = Embedding(input_dim=num_classes, output_dim=np.prod(img_shape), input_length=1)(label) # e

    label_embedding = Flatten()(label_embedding) # flatten the embedding 3D tensor into 2D tensor with shape: (b
    label_embedding = Reshape(img_shape)(label_embedding) # reshape label embeddings to have same dimensions as 

    combined_input = multiply([img, label_embedding]) # element-wise multiplication of the vectors z and the lab

    validity = model(combined_input)

    return Model([img, label], validity)
```

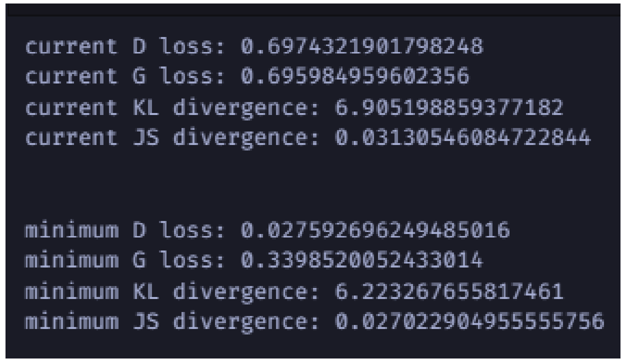
Proper kernel initialization ensures that the starting weights of these networks are neither too large nor too small, which helps in **maintaining balanced gradients during backpropagation**. If weights are poorly initialized, they can lead to gradients that either vanish or explode, making the training process slow and unstable.

In this case, we used the **glorot_normal initializer**, which scales weights based on the layer's size, **helps maintain a consistent variance of activations and gradients** across layers.

This balance **promotes smoother and faster learning**, leading to better performance of the GAN in generating high-quality images.

MODEL IMPROVEMENT 1(KERNEL INITIALIZER)



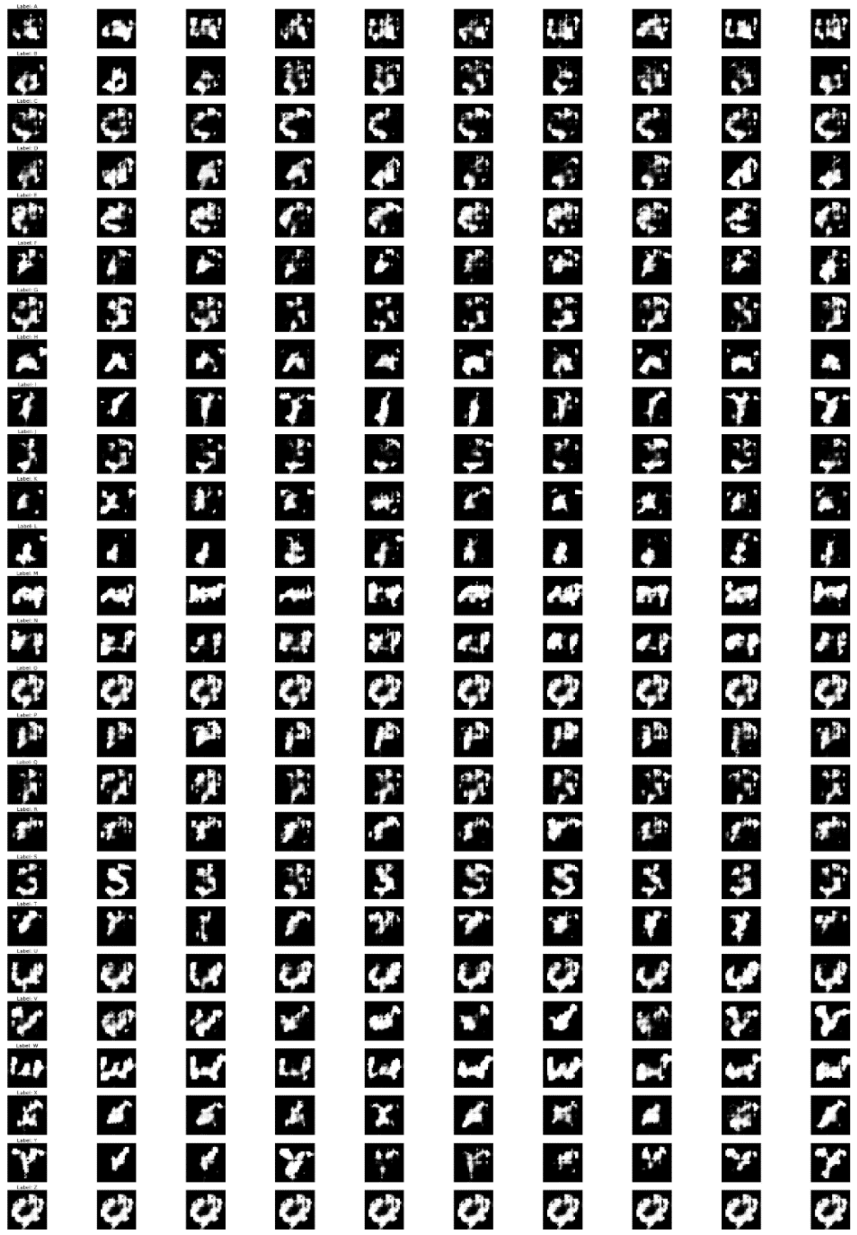


```
current D loss: 0.6974321901798248
current G loss: 0.695984959602356
current KL divergence: 6.905198859377182
current JS divergence: 0.03130546084722844


minimum D loss: 0.027592696249485016
minimum G loss: 0.3398520052433014
minimum KL divergence: 6.223267655817461
minimum JS divergence: 0.027022904955555756
```

The overall graphs are **still somewhat unstable, with some fluctuations**. Looking at the losses and discriminator accuracy, the model converged at around 1500 epochs.

During the training, the model reached a minimum Kl and JS divergence of **6.2** and **0.027** respectively.

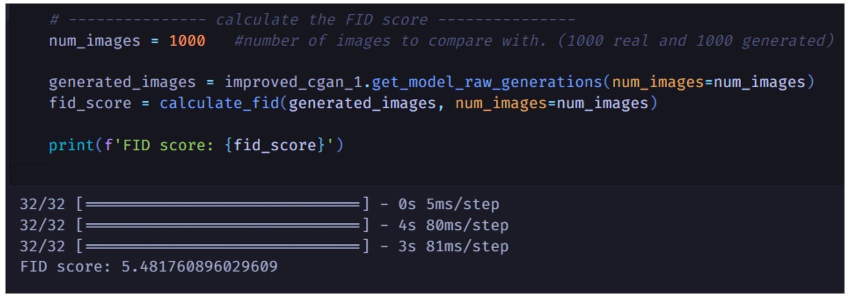
MODEL IMPROVEMENT 1(KERNEL INITIALIZER)



To evaluate the quality of images produced, we introduced **Fréchet Inception Distance (FID),** which measures **how similar the features of generated images are to the features of real images**, using the Inception V3 network. **Lower scores suggests two groups of images are more similar,** or have more similar statistics, with a perfect score being 0 indicating that the two groups of images are identical.

Evaluating “by-eye”, the **generated images are still noisy**, with only some classes like 'S' and 'O' being recognizable. Additionally, there is less variability within certain classes, such as 'O', where all 10 images are quite similar, suggesting possible mode collapse.

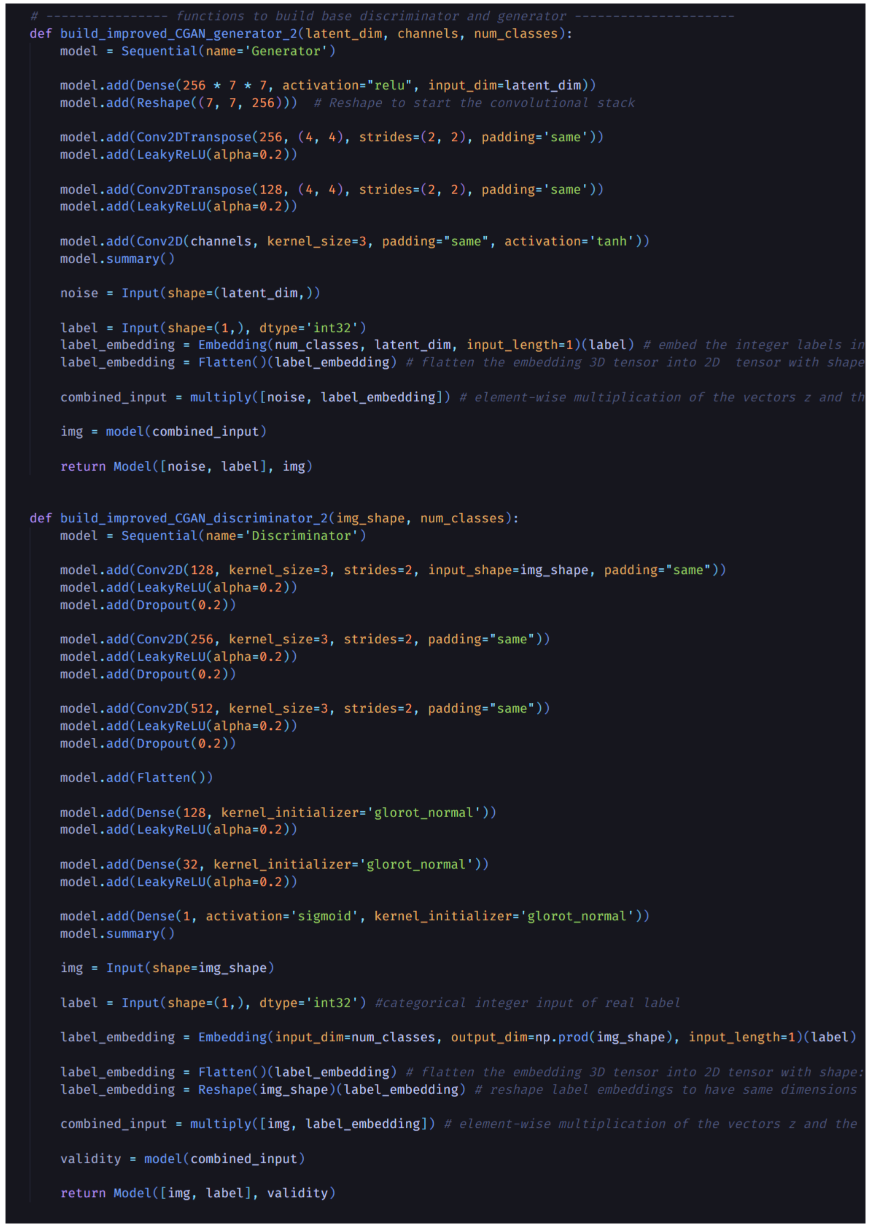
Using FID, comparing with a total of 1000 real and 1000 fake images, an **FID score of 5.48** was obtained. We will be using this score to compare with the improved models later on.



```python
# --------------- calculate the FID score ---------------
num_images = 1000   #number of images to compare with. (1000 real and 1000 generated)

generated_images = improved_cgan_1.get_model_raw_generations(num_images=num_images)
fid_score = calculate_fid(generated_images, num_images=num_images)

print(f'FID score: {fid_score}')
```

```
32/32 [==============================] - 0s 5ms/step
32/32 [==============================] - 4s 80ms/step
32/32 [==============================] - 3s 81ms/step
FID score: 5.481760896029609
```

MODEL IMPROVEMENT 2 (INCREASE COMPLEXITY)



```python
# ---------------- functions to build base discriminator and generator ---------------------
def build_improved_CGAN_generator_2(latent_dim, channels, num_classes):
    model = Sequential(name='Generator')

    model.add(Dense(256 * 7 * 7, activation="relu", input_dim=latent_dim))
    model.add(Reshape((7, 7, 256)))  # Reshape to start the convolutional stack

    model.add(Conv2DTranspose(256, (4, 4), strides=(2, 2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))

    model.add(Conv2DTranspose(128, (4, 4), strides=(2, 2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))

    model.add(Conv2D(channels, kernel_size=3, padding="same", activation='tanh'))
    model.summary()

    noise = Input(shape=(latent_dim,))

    label = Input(shape=(1,), dtype='int32')
    label_embedding = Embedding(num_classes, latent_dim, input_length=1)(label) # embed the integer labels in
    label_embedding = Flatten()(label_embedding) # flatten the embedding 3D tensor into 2D  tensor with shape

    combined_input = multiply([noise, label_embedding]) # element-wise multiplication of the vectors z and th

    img = model(combined_input)

    return Model([noise, label], img)


def build_improved_CGAN_discriminator_2(img_shape, num_classes):
    model = Sequential(name='Discriminator')

    model.add(Conv2D(128, kernel_size=3, strides=2, input_shape=img_shape, padding="same"))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.2))

    model.add(Conv2D(256, kernel_size=3, strides=2, padding="same"))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.2))

    model.add(Conv2D(512, kernel_size=3, strides=2, padding="same"))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.2))

    model.add(Flatten())

    model.add(Dense(128, kernel_initializer='glorot_normal'))
    model.add(LeakyReLU(alpha=0.2))

    model.add(Dense(32, kernel_initializer='glorot_normal'))
    model.add(LeakyReLU(alpha=0.2))

    model.add(Dense(1, activation='sigmoid', kernel_initializer='glorot_normal'))
    model.summary()

    img = Input(shape=img_shape)

    label = Input(shape=(1,), dtype='int32') #categorical integer input of real label

    label_embedding = Embedding(input_dim=num_classes, output_dim=np.prod(img_shape), input_length=1)(label)

    label_embedding = Flatten()(label_embedding) # flatten the embedding 3D tensor into 2D tensor with shape:
    label_embedding = Reshape(img_shape)(label_embedding) # reshape label embeddings to have same dimensions

    combined_input = multiply([img, label_embedding]) # element-wise multiplication of the vectors z and the

    validity = model(combined_input)

    return Model([img, label], validity)
```

For the generator,

- the number of neurons in the dense and number of filters in the Cov2dTranspose layers are increased by **4 times**.
- This added capacity enables the generator to produce more detailed, higher-quality images by **better capturing the data distribution's complexity and reducing blurriness** in the generated images.

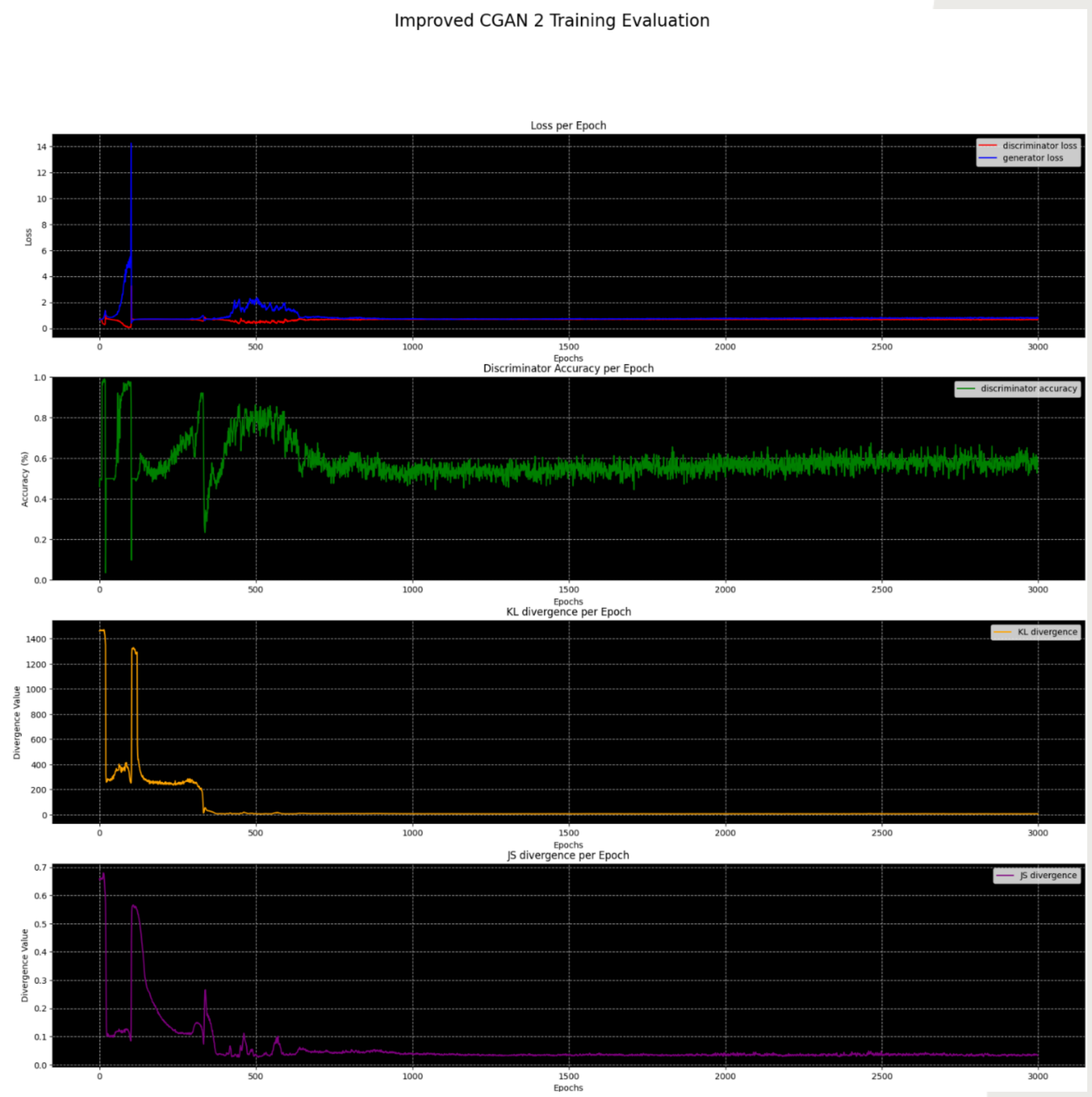
For the discriminator,

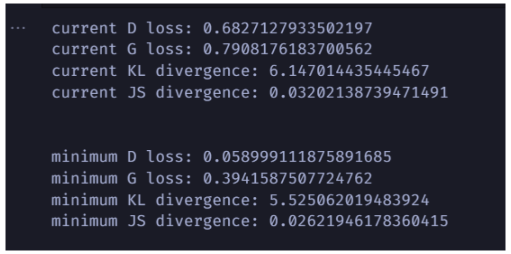
- the number of filters in the Conv2D layers are increased by **8 times**. There is also an additional two dense layers at the end with 128 and 32 neurons, respectively.
- Increasing the number of filters in the Conv2D layers improves the discriminator's ability to detect finer details and complex features in both real and generated images. This **enhances the discriminator's ability to distinguish between real and fake images**.
- Adding extra Dense layers with more neurons **refines this classification further**, increasing the performance of the discriminator and driving the generator to produce higher-quality images.

MODEL IMPROVEMENT 2 (INCREASE COMPLEXITY)

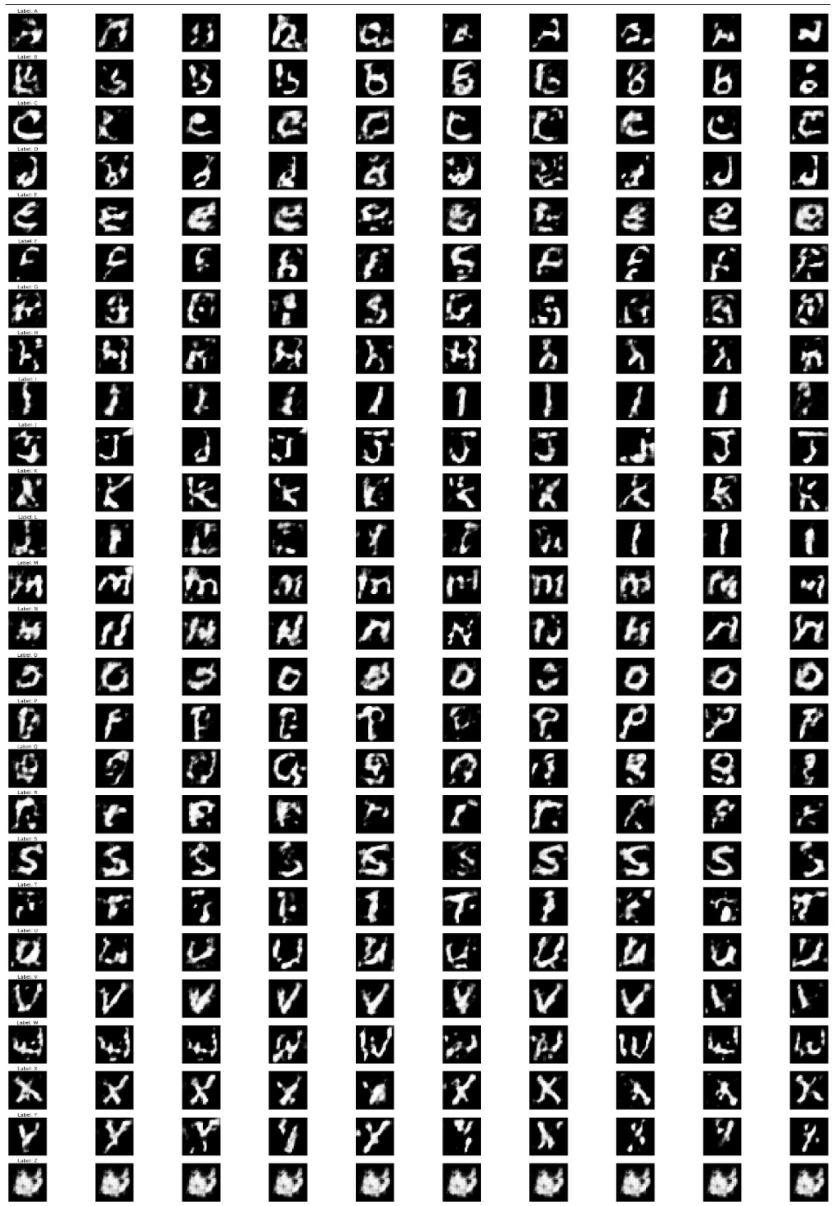
Overall, the graphs **are more stable** compared to the previous model. Additionally, the losses and discriminator accuracy show that the model **converged faster**, around 1,000 epochs.

The minimum KL and JS divergence values were also **lower**, reaching **5.5** and **0.026**, respectively. This indicates that the generated results are more closely aligned with the distribution of the real images.



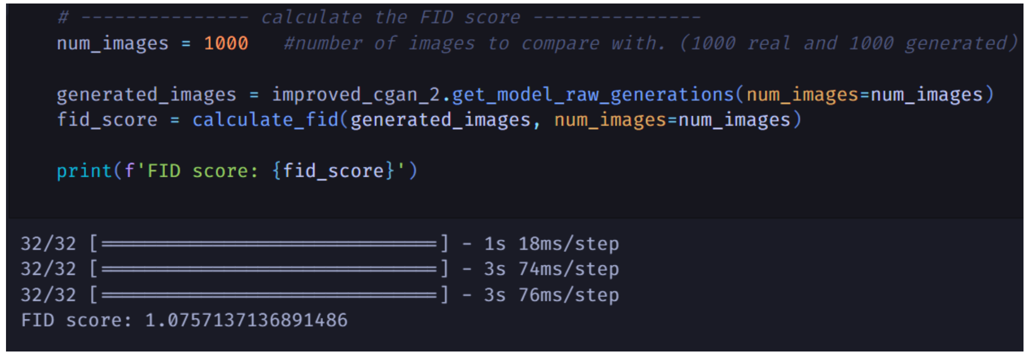

```
current D loss: 0.6827127933502197
current G loss: 0.7908176183700562
current KL divergence: 6.147014435445467
current JS divergence: 0.03202138739471491


minimum D loss: 0.058999111875891685
minimum G loss: 0.3941587507724762
minimum KL divergence: 5.525062019483924
minimum JS divergence: 0.02621946178360415
```

MODEL IMPROVEMENT 2 (INCREASE COMPLEXITY)



As compared to model 1, we can see that this CGAN with increased complexity generated letters that are **more defined and less noisy.**

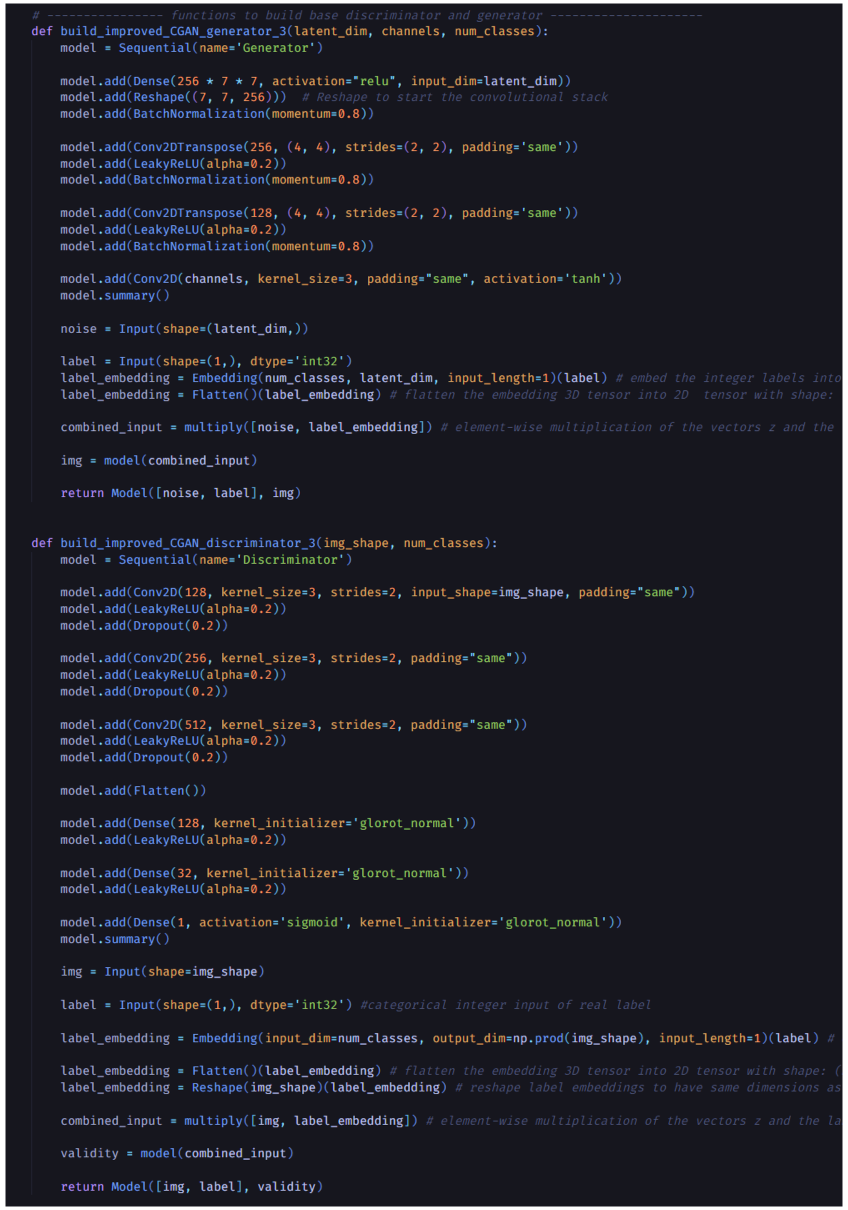
The **FID Score also decreased from 5.48 to 1.08**, this indicates that the model's generated images have become significantly more similar to the real images



```
# --------------- calculate the FID score ---------------
num_images = 1000   #number of images to compare with. (1000 real and 1000 generated)

generated_images = improved_cgan_2.get_model_raw_generations(num_images=num_images)
fid_score = calculate_fid(generated_images, num_images=num_images)

print(f'FID score: {fid_score}')
```

```
32/32 [==============================] - 1s 18ms/step
32/32 [==============================] - 3s 74ms/step
32/32 [==============================] - 3s 76ms/step
FID score: 1.0757137136891486
```

MODEL IMPROVEMENT 3 (BATCH NORMALIZATION AND BATCHSIZE)



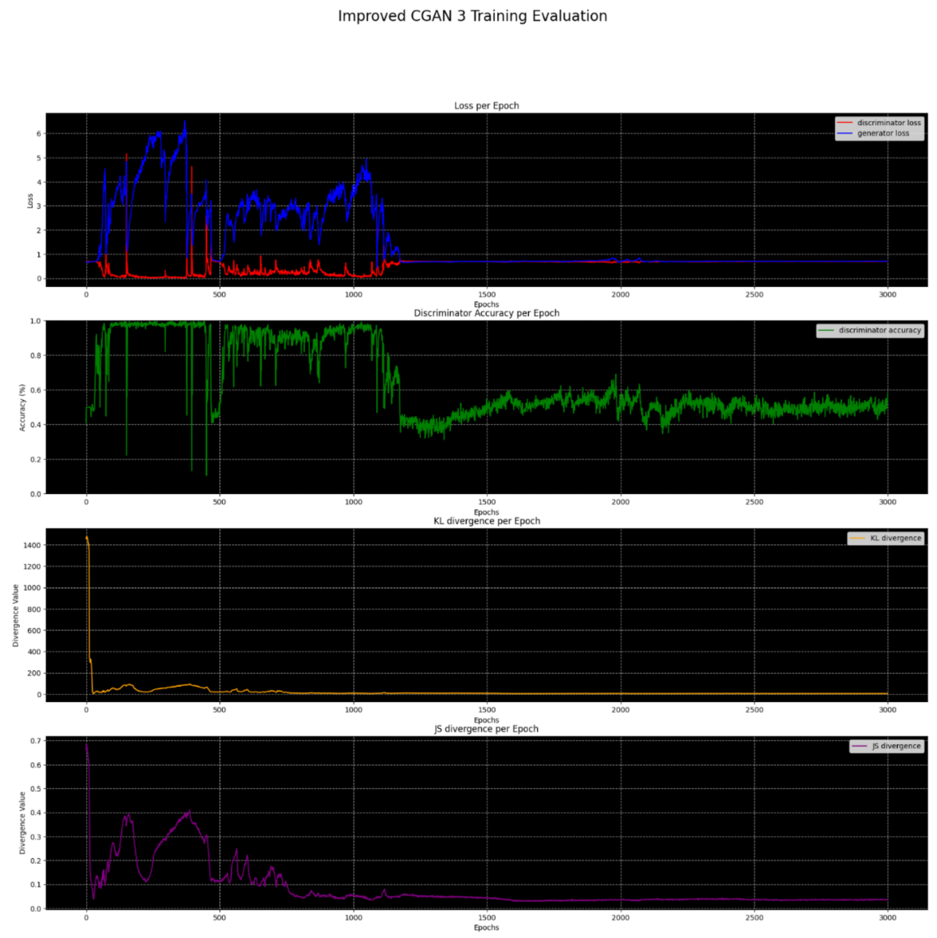
```python
# ---------------- functions to build base discriminator and generator ---------------------
def build_improved_CGAN_generator_3(latent_dim, channels, num_classes):
    model = Sequential(name='Generator')

    model.add(Dense(256 * 7 * 7, activation="relu", input_dim=latent_dim))
    model.add(Reshape((7, 7, 256)))  # Reshape to start the convolutional stack
    model.add(BatchNormalization(momentum=0.8))

    model.add(Conv2DTranspose(256, (4, 4), strides=(2, 2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))

    model.add(Conv2DTranspose(128, (4, 4), strides=(2, 2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))

    model.add(Conv2D(channels, kernel_size=3, padding="same", activation='tanh'))
    model.summary()

    noise = Input(shape=(latent_dim,))

    label = Input(shape=(1,), dtype='int32')
    label_embedding = Embedding(num_classes, latent_dim, input_length=1)(label) # embed the integer labels into
    label_embedding = Flatten()(label_embedding) # flatten the embedding 3D tensor into 2D  tensor with shape:

    combined_input = multiply([noise, label_embedding]) # element-wise multiplication of the vectors z and the

    img = model(combined_input)

    return Model([noise, label], img)


def build_improved_CGAN_discriminator_3(img_shape, num_classes):
    model = Sequential(name='Discriminator')

    model.add(Conv2D(128, kernel_size=3, strides=2, input_shape=img_shape, padding="same"))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.2))

    model.add(Conv2D(256, kernel_size=3, strides=2, padding="same"))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.2))

    model.add(Conv2D(512, kernel_size=3, strides=2, padding="same"))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.2))

    model.add(Flatten())

    model.add(Dense(128, kernel_initializer='glorot_normal'))
    model.add(LeakyReLU(alpha=0.2))

    model.add(Dense(32, kernel_initializer='glorot_normal'))
    model.add(LeakyReLU(alpha=0.2))

    model.add(Dense(1, activation='sigmoid', kernel_initializer='glorot_normal'))
    model.summary()

    img = Input(shape=img_shape)

    label = Input(shape=(1,), dtype='int32') #categorical integer input of real label

    label_embedding = Embedding(input_dim=num_classes, output_dim=np.prod(img_shape), input_length=1)(label) #

    label_embedding = Flatten()(label_embedding) # flatten the embedding 3D tensor into 2D tensor with shape: (
    label_embedding = Reshape(img_shape)(label_embedding) # reshape label embeddings to have same dimensions as

    combined_input = multiply([img, label_embedding]) # element-wise multiplication of the vectors z and the la

    validity = model(combined_input)

    return Model([img, label], validity)
```

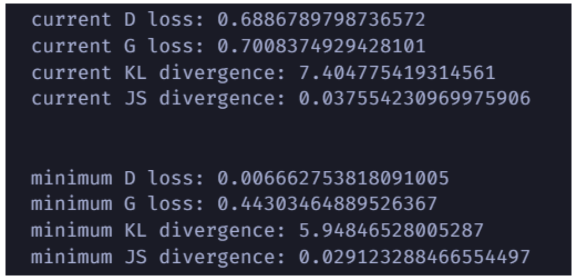
Batch normalization can **accelerate training** by allowing higher learning rates and reducing the sensitivity to parameter initialization. This can lead to faster convergence of the GAN model.

It also helps **mitigate the internal covariate shift problem**, where the distribution of network activations changes during training. It can have a regularizing effect on the model, which may help prevent overfitting and improve generalization. This can make the training process more robust and help the model generalize the letters better.

We **reduced the batchsize from 256 to 128**, this is because smaller batch sizes can sometimes lead to **more stable training, especially in the early stages of GAN training.** This is because they **introduce more noise into the gradient estimates**. Smaller batch sizes can also lead to better generalization. This is because the added noise from smaller batches can act as a form of regularization, potentially preventing overfitting

MODEL IMPROVEMENT 3 (BATCH NORMALIZATION AND BATCHSIZE)





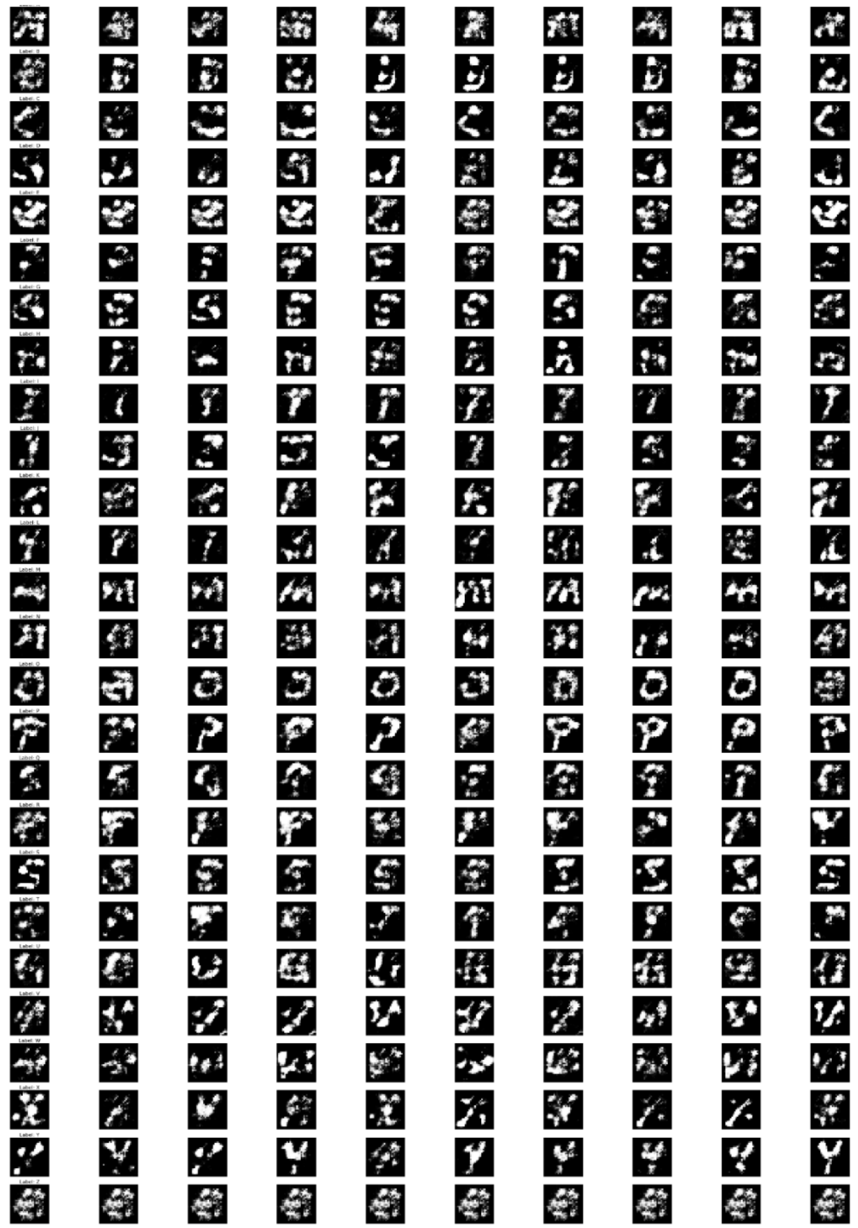
```
current D loss: 0.6886789798736572
current G loss: 0.7008374929428101
current KL divergence: 7.404775419314561
current JS divergence: 0.037554230969975906


minimum D loss: 0.006662753818091005
minimum G loss: 0.44303464889526367
minimum KL divergence: 5.94846528005287
minimum JS divergence: 0.029123288466554497
```

We can see that the training is **not as fast as model 2**, and that it took around 1200 epochs for the losses to converge and about 2500 epochs for the discriminator accuracy to stabilize at 0.5.

Overall, the KL and JS divergence **increased slightly to 5.9 and 0.029**. This increase can be due to the added noise caused by batch normalization.

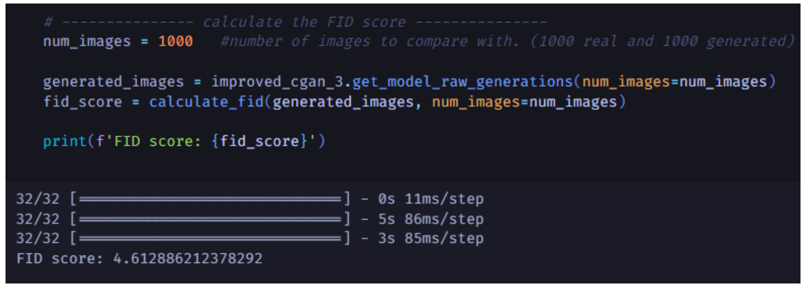
Batch normalization may **cause the network to be more sensitive to learning rates**. Hence, training might become **slower or less stable**. Therefore, we have to tune the learning rates after this.

MODEL IMPROVEMENT 3 (BATCH NORMALIZATION AND BATCHSIZE)



The generated images is **more noisy and less defined** compared to CGAN 2. The **FID also increased back up to 4.61**, indicating lower quality images generated. This can be due to the additional noise introduced through batch normalization.

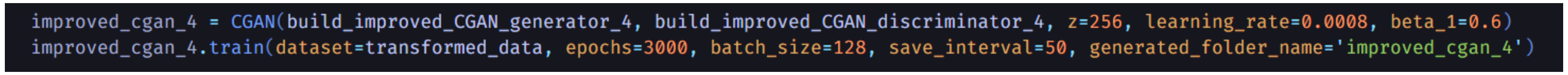
Hence, we will be tuning the hyperparameters of the networks, especially the learning rates



```
# -------------- calculate the FID score --------------
num_images = 1000   #number of images to compare with. (1000 real and 1000 generated)

generated_images = improved_cgan_3.get_model_raw_generations(num_images=num_images)
fid_score = calculate_fid(generated_images, num_images=num_images)

print(f'FID score: {fid_score}')
```

```
32/32 [==============================] - 0s 11ms/step
32/32 [==============================] - 5s 86ms/step
32/32 [==============================] - 3s 85ms/step
FID score: 4.612886212378292
```

MODEL IMPROVEMENT 4 (LEARNING RATE, BETA, NOISE DIMENSION)



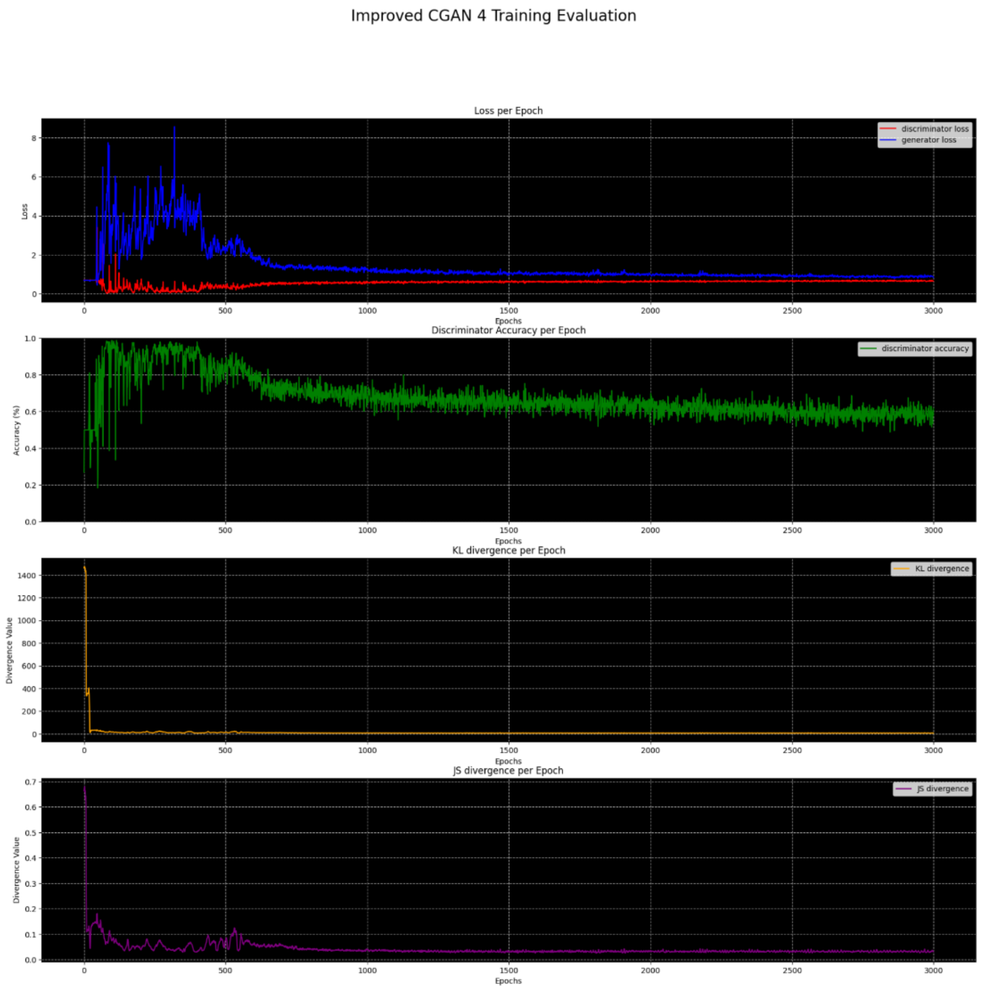
```
improved_cgan_4 = CGAN(build_improved_CGAN_generator_4, build_improved_CGAN_discriminator_4, z=256, learning_rate=0.0008, beta_1=0.6)
improved_cgan_4.train(dataset=transformed_data, epochs=3000, batch_size=128, save_interval=50, generated_folder_name='improved_cgan_4')
```

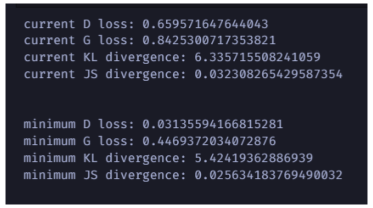
Increased learning rate from 0.0002 to 0.0008 for faster convergence. A higher learning rate allows the model to make larger updates to the weights during training, which can speed up the learning process. We also ensured that this higher learning rate is still stable enough for the model, by evaluating the training curves.

Increased noise dimension from 100 to 256. Increasing the noise dimension allows the generator to capture more complex patterns and details in the data, this **provides more freedom for the generator to create diverse and higher quality images**. With a larger noise dimension, the generator can potentially cover more modes of the data distribution, **reducing issues like mode collapse** where the generator produces limited and repetitive outputs.

Increased beta_1 from 0.5 to 0.6. beta_1 controls how much the optimizer "remembers" previous gradients. It's used to calculate a running average of the gradients, which helps smooth out the optimization process. Increasing beta_1 can **lead to smoother and more stable updates by giving more weight to past gradients**, which can help in stabilizing the training process.

MODEL IMPROVEMENT 4 (LEARNING RATE, BETA, NOISE DIMENSION)





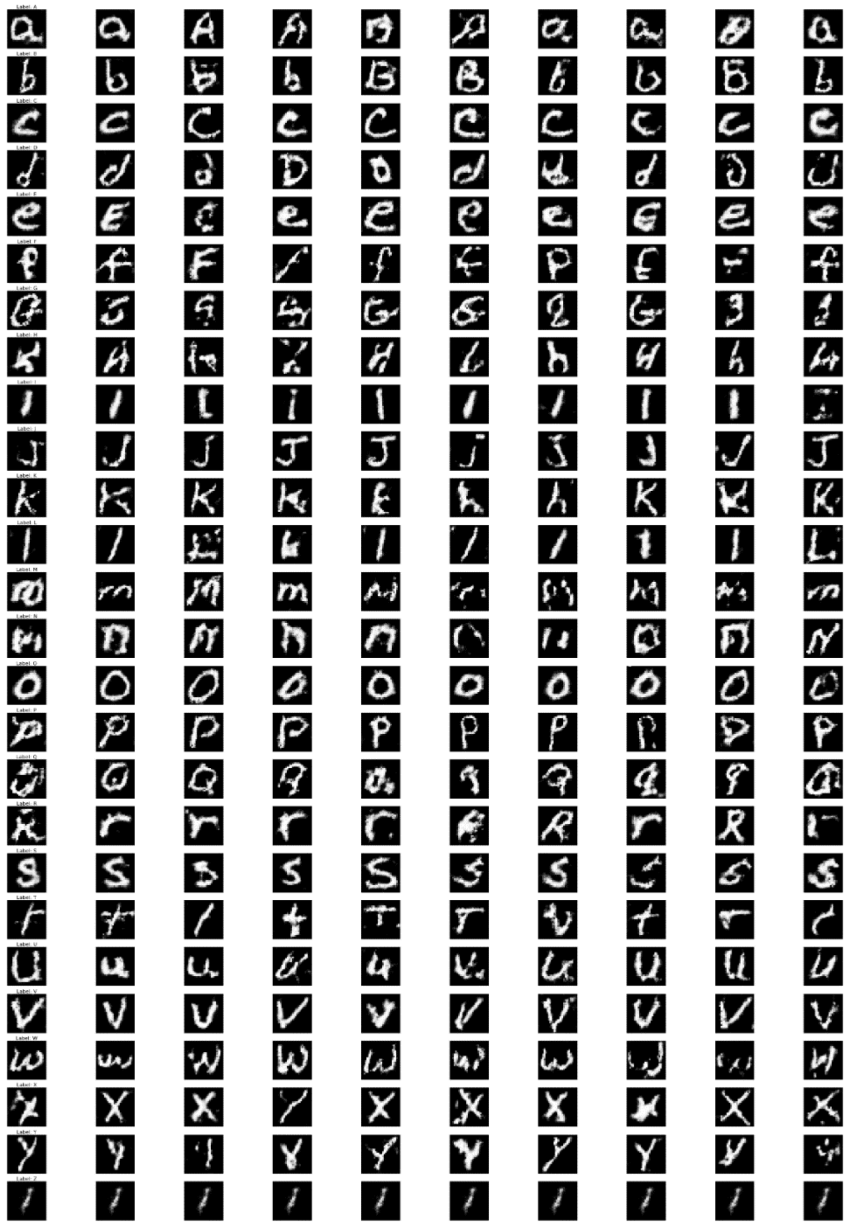
```
current D loss: 0.659571647644043
current G loss: 0.8425300717353821
current KL divergence: 6.335715508241059
current JS divergence: 0.032308265429587354


minimum D loss: 0.03135594166815281
minimum G loss: 0.4469372034072876
minimum KL divergence: 5.42419362886939
minimum JS divergence: 0.025634183769490032
```

The **losses and accuracy graphs is smoother, with no sudden peaks or dip**s. The losses converges and stabilizes at around 2700 epochs. The accuracy curve gradually decreases to about 0.6.

The minimum KL and JS divergence also **decreased to 5.4 and 0.026** respectively.

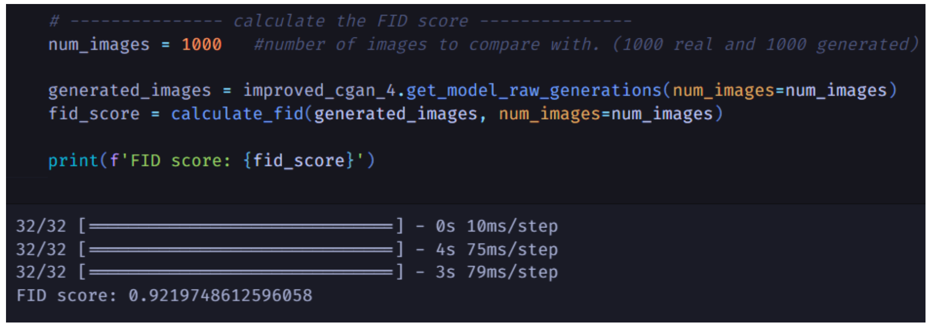
The Smooth training curves indicate that the model is making **consistent progress throughout the training process**, rather than experiencing sudden performance drops. Furthermore, **smooth training helps mitigate the risk of mode collapse**, where the generator produces limited varieties of samples.

MODEL IMPROVEMENT 4 (LEARNING RATE, BETA, NOISE DIMENSION)



We can see that the letters are more defined and readable, with significantly less noise.

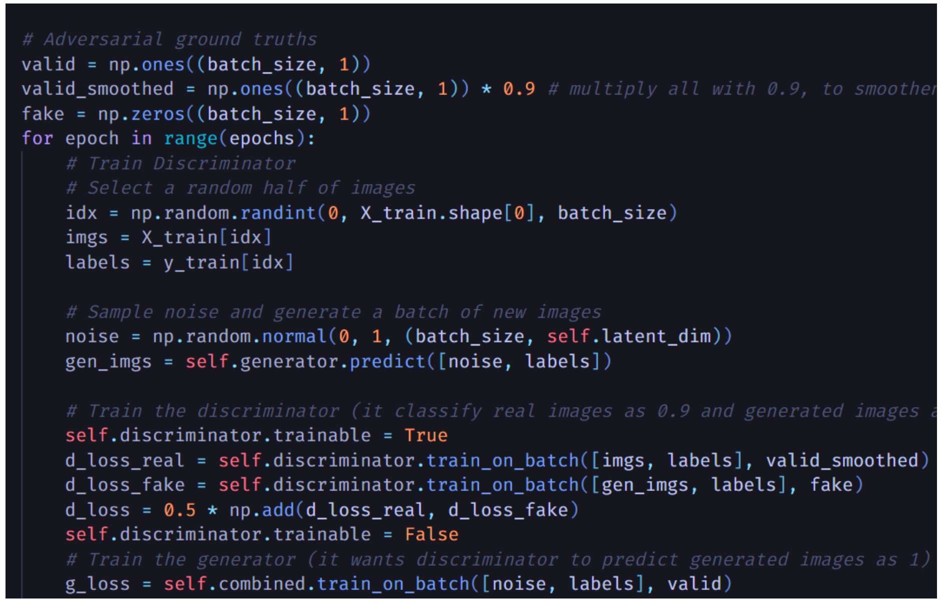
The **FID decreased to 0.922**, indicating that the generated images are closer to the real images compared to the previous models



```
# --------------- calculate the FID score ---------------
num_images = 1000   #number of images to compare with. (1000 real and 1000 generated)

generated_images = improved_cgan_4.get_model_raw_generations(num_images=num_images)
fid_score = calculate_fid(generated_images, num_images=num_images)

print(f'FID score: {fid_score}')
```

```
32/32 [==============================] - 0s 10ms/step
32/32 [==============================] - 4s 75ms/step
32/32 [==============================] - 3s 79ms/step
FID score: 0.9219748612596058
```

MODEL IMPROVEMENT 5 (ONE SIDED LABEL SMOOTHING)



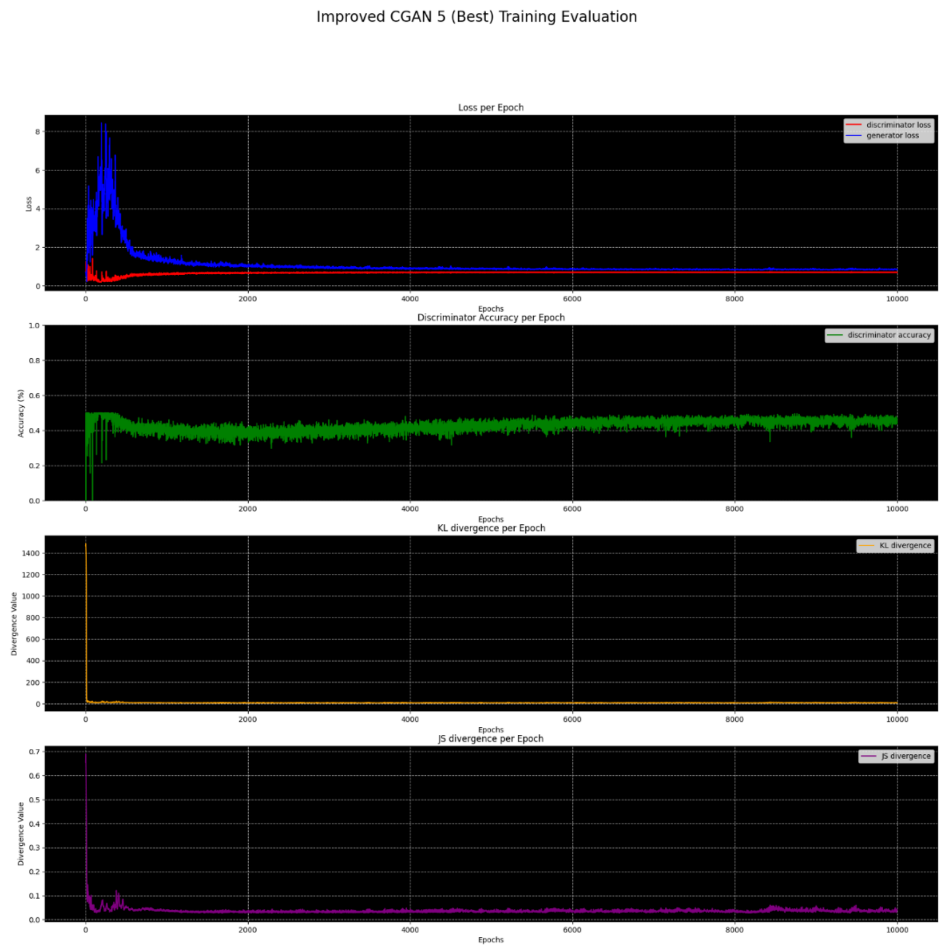
```python
# Adversarial ground truths
valid = np.ones((batch_size, 1))
valid_smoothed = np.ones((batch_size, 1)) * 0.9 # multiply all with 0.9, to smoother
fake = np.zeros((batch_size, 1))
for epoch in range(epochs):
    # Train Discriminator
    # Select a random half of images
    idx = np.random.randint(0, X_train.shape[0], batch_size)
    imgs = X_train[idx]
    labels = y_train[idx]

    # Sample noise and generate a batch of new images
    noise = np.random.normal(0, 1, (batch_size, self.latent_dim))
    gen_imgs = self.generator.predict([noise, labels])

    # Train the discriminator (it classify real images as 0.9 and generated images a
    self.discriminator.trainable = True
    d_loss_real = self.discriminator.train_on_batch([imgs, labels], valid_smoothed)
    d_loss_fake = self.discriminator.train_on_batch([gen_imgs, labels], fake)
    d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)
    self.discriminator.trainable = False
    # Train the generator (it wants discriminator to predict generated images as 1)
    g_loss = self.combined.train_on_batch([noise, labels], valid)
```

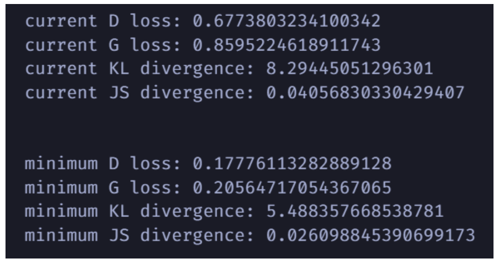
We changed the **labels for real images from 1.0 to 0.9** as the target values for the discriminator during training.

One sided label smoothing helps **prevent the discriminator from becoming too confident in its predictions for real images**. By setting the target for real images to a value slightly below 1.0, it encourages the discriminator to be less certain, which can improve overall training stability. By smoothing only the positive labels, it **prevents the discriminator from giving very large gradients to the generator**. This can help stabilize the training process and prevent issues like mode collapse (https://www.researchgate.net/publication/380084231_Application_of_Smoothing_Labels_to_Alleviate_Overconfident_of_the_GAN's_Discriminator)

Furthermore, unlike two-sided label smoothing, **one-sided smoothing maintains the incentive for fake samples to move towards the real data distribution**, as it doesn't modify the labels for fake samples

MODEL IMPROVEMENT 5 (ONE SIDED LABEL SMOOTHING)





```
current D loss: 0.6773803234100342
current G loss: 0.8595224618911743
current KL divergence: 8.29445051296301
current JS divergence: 0.04056830330429407


minimum D loss: 0.17776113282889128
minimum G loss: 0.20564717054367065
minimum KL divergence: 5.488357668538781
minimum JS divergence: 0.026098845390699173
```

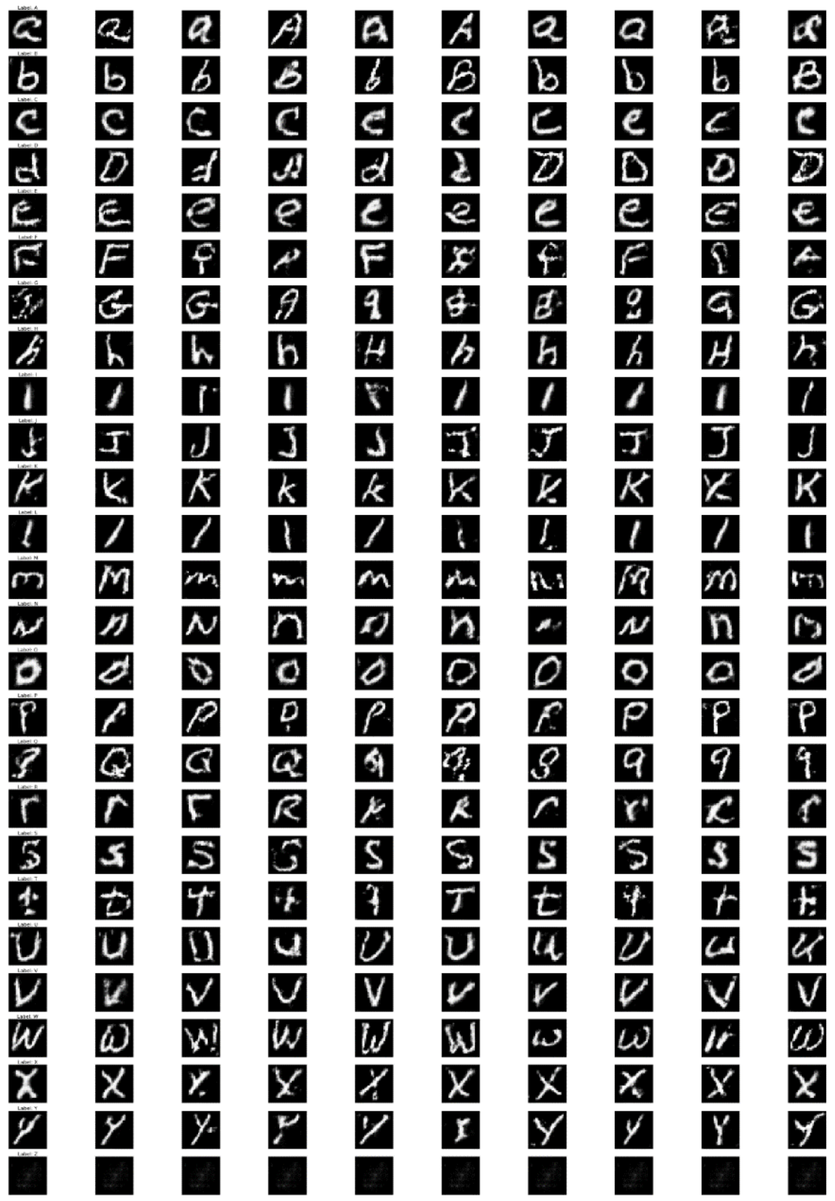
Comparing the training graph to the previous 4 models, we can see that the discriminator accuracy curve is **smoother and more stable, hovering at around 0.5 throughout**, instead of spiking or fluctuating.
The discriminator and generator loss is also smoother, we less fluctuations and converges to a stable point

We **increased the number of epochs** for the model to train since the **convergence took slightly longer** than the previous models mainly **due to the slower learning rate introduced in model 4.** By increasing the number of epochs, we can also ensure that the model l**earns the distribution of each class accurately,** furthermore, in this case, there was no overfitting occurring due to the help of one-sided label smoothing.

The minimum KL and JS was **maintained at around 5.5 and 0.026.**

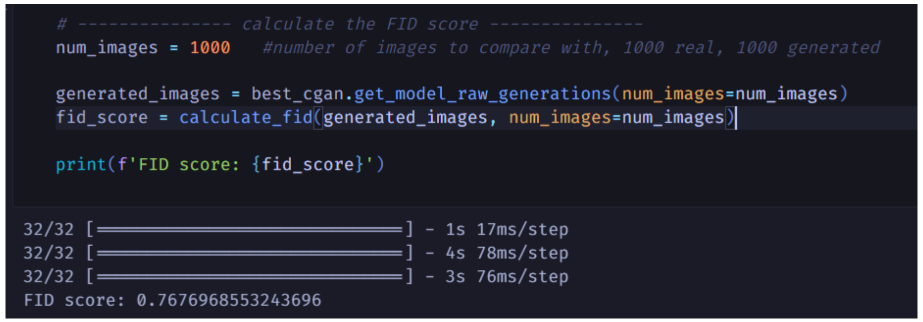
BEST MODEL EVALUATION

BEST MODEL EVALUATION



Most images produces are **sharp and defined**, with a few exceptions. We can see that the model has learnt to **produce different styles of the same class**. For example, it has learnt to produce lower and uppercase version of 'a'. This variation suggests that the generator has effectively learned the nuances and variations within each class, generating diverse outputs. However, can see that **'z' was the hardest class to generate, with the images almost being blank.**

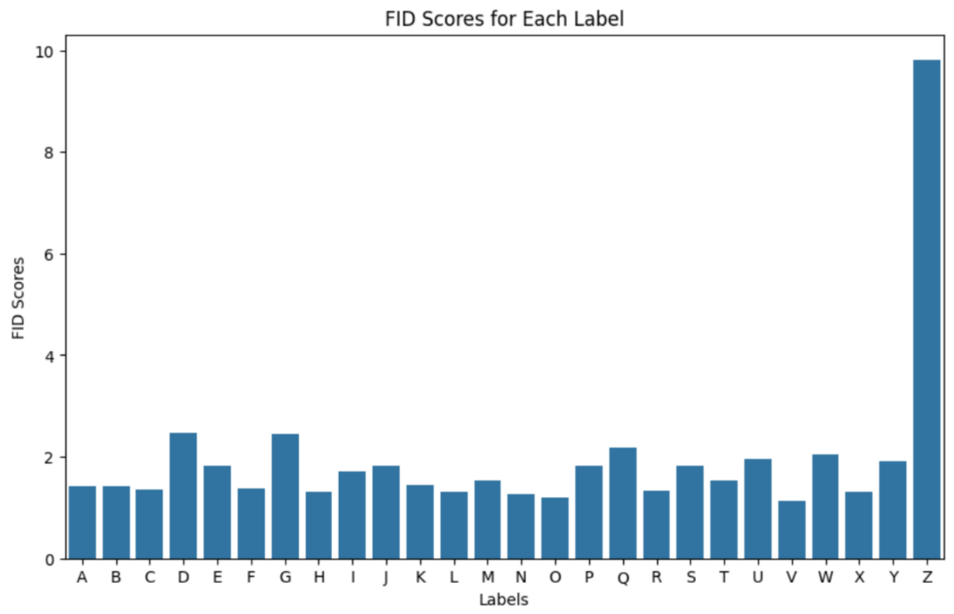
With the **lowest FID score of 0.768**, the generated images are the best (closest to the real images) and **improved by about 4.72 FID from the first CGAN model.**



```
# --------------- calculate the FID score ---------------
num_images = 1000   #number of images to compare with, 1000 real, 1000 generated

generated_images = best_cgan.get_model_raw_generations(num_images=num_images)
fid_score = calculate_fid(generated_images, num_images=num_images)

print(f'FID score: {fid_score}')
```

```
32/32 [==============================] - 1s 17ms/step
32/32 [==============================] - 4s 78ms/step
32/32 [==============================] - 3s 76ms/step
FID score: 0.7676968553243696
```

BEST MODEL EVALUATION
WHICH CLASSES ARE EASIER/HARDER TO GENERATE?



Class ‘Z’ is the hardest to generate and has generated images that differs the most from the real images based on the **significantly higher FID score of close to 10** compared to the rest of the classes. This might be due to the images of 'z' having significantly more diverse variations of handwritings. There also might be problems with the label embeddings for the class 'z'. There could be a possibility that the embedding for 'z' is misaligned or does not effectively map to the corresponding generator layers, since it was the last class. This misalignment can result in the generator failing to learn and generate the specific features of the letter 'z’.

Classes ‘V’ and ‘O’ are the easiest to generate. Class 'V' has the lowest FID followed by 'O', at around 1.0. These classes might have a low FID score as these letters are relatively consistent in the handwriting images, with its uppercases and lowercases being the same. The reduced variability in the shapes of 'V' and 'O' compared to other characters may contribute to more consistent and accurate generation.

CONCLUSION

In summary, **three different variations** of GANs were built (DCGAN, CGAN, WGAN). We proposed a way of generating images of a specific class using label embeddings in CGAN, which we focused on for model improvement. **Several metrices** were also used to evaluate the quality of images apart from a simple "by-eye" method, we used KL and JS divergences, as well as FID. During the improvement, we managed to **decrease the FID score of the generated images from 5.48 to 0.768**.

If we were asked to generate coloured images instead of black and white ones, we think that **it is harder** to produce better quality results in terms of model building and training.

- This is because **coloured images have 3 channels compared to 1** in grayscale images. This means that apart from the spatial dimensions, there is an added "colour dimension" that the model has the learn and capture. This would **increase the input size and complexity** of the data that the GAN has to learn.
- With this added dimension, there are also **higher variability** and hence, more training data is required to capture the diversity of the colours. The generator also needs to **maintain color consistency across the image**, which can be challenging, especially for complex scenes or objects as it **may produce color artifacts or unrealistic color combinations** that are more noticeable than artifacts in grayscale images.
- It can also be **harder to evaluate the quality** of the generated coloured images since the **metrices would have to consider both structural and colour fidelity**.
- Overall, the **model's complexity might have to be increased**, and **training time might take longer**. Hence, it would be harder to produce better quality results for coloured images.

Additionally, other experiments could include using instance normalization and separating upsampling from convolution by applying techniques such as nearest-neighbor interpolation to reduce checkerboard artifacts (https://distill.pub/2016/deconv-checkerboard/). Other GAN architectures, such as ACGAN, involve a discriminator that only takes the image as input and outputs both the probability that the image is real and a prediction of the image's class label (https://www.researchgate.net/figure/GAN-conditional-GAN-CGAN-and-auxiliary-classifier-GAN-ACGAN-architectures-where-x_fig1_330474693). Exploring these architectures may lead to further improvements in generating the EMNIST letters.