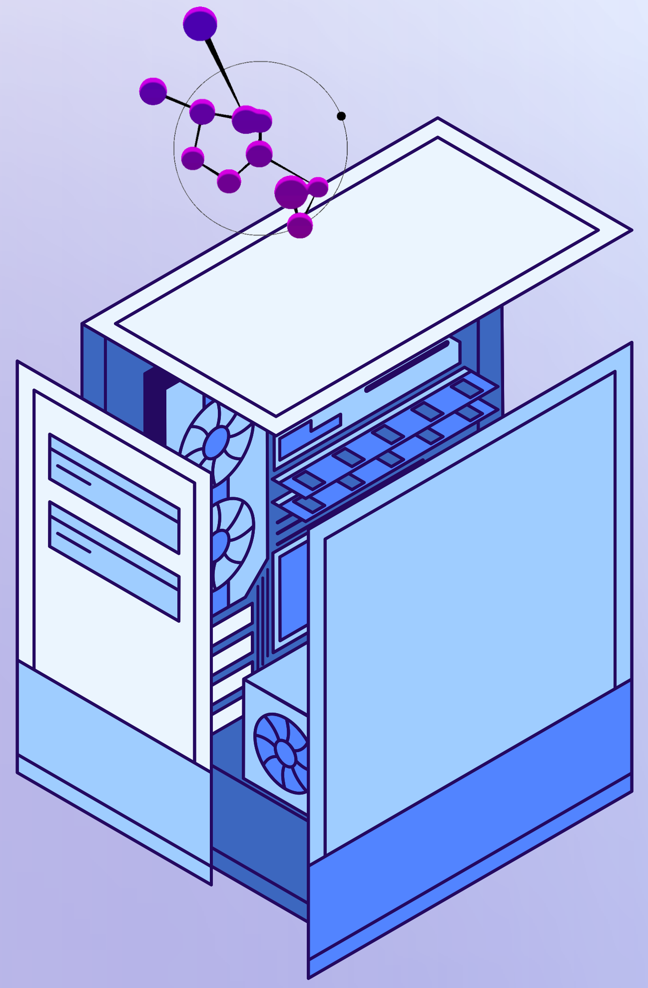
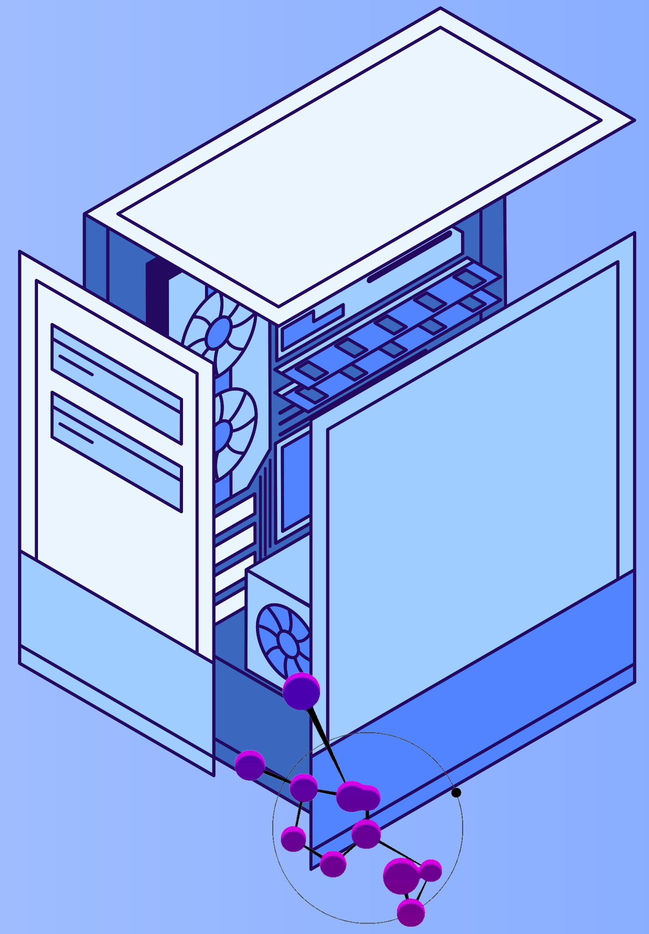
CA2

DEEP LEARNING

By PinPin x Shane



GENERATIVE ADVERSARIAL NETWORK



BACKGROUND INFO

Generating Realistic Images

Use Cases

- It greatly enhances efficiency and scalability, allowing businesses and individuals to produce high-quality images quickly and cost-effectively.

Down Side:

- The technology can be misused to create **deepfakes** or **misleading** information, raising ethical concerns. It might also reduce job opportunities for traditional artists and designers.

TABLE OF CONTENTS

- Dataset Exploration
- Dataset Processing
- Model Selection
- Model Improvement

DATASET EXPLORATION

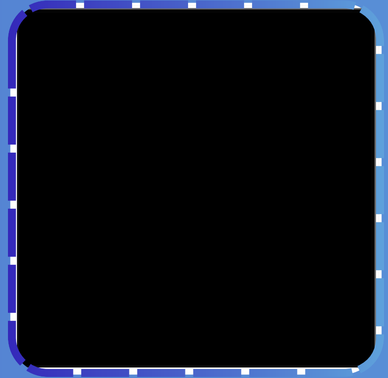
6.

DATA INFORMATION

Dataset

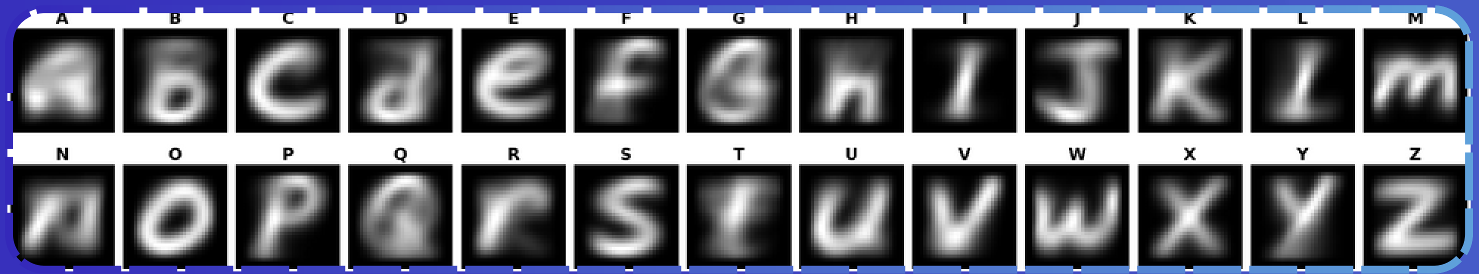
- Images (99040, 28, 28)
- Labels (-1 | 26)

Label (-1) Was just a blank Image

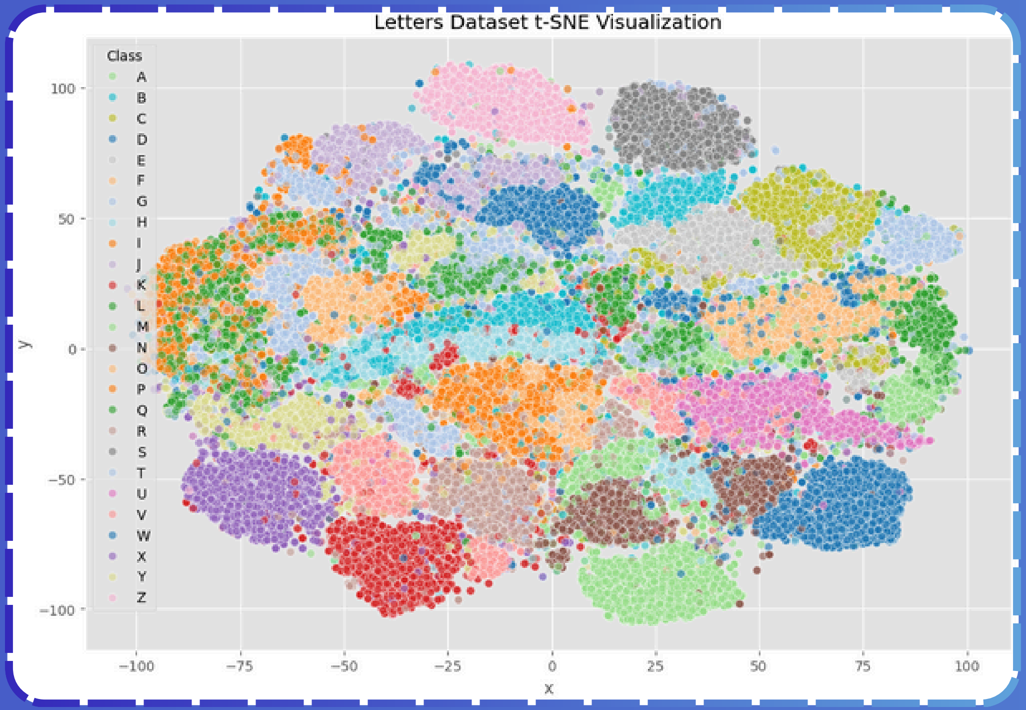


Raw Images

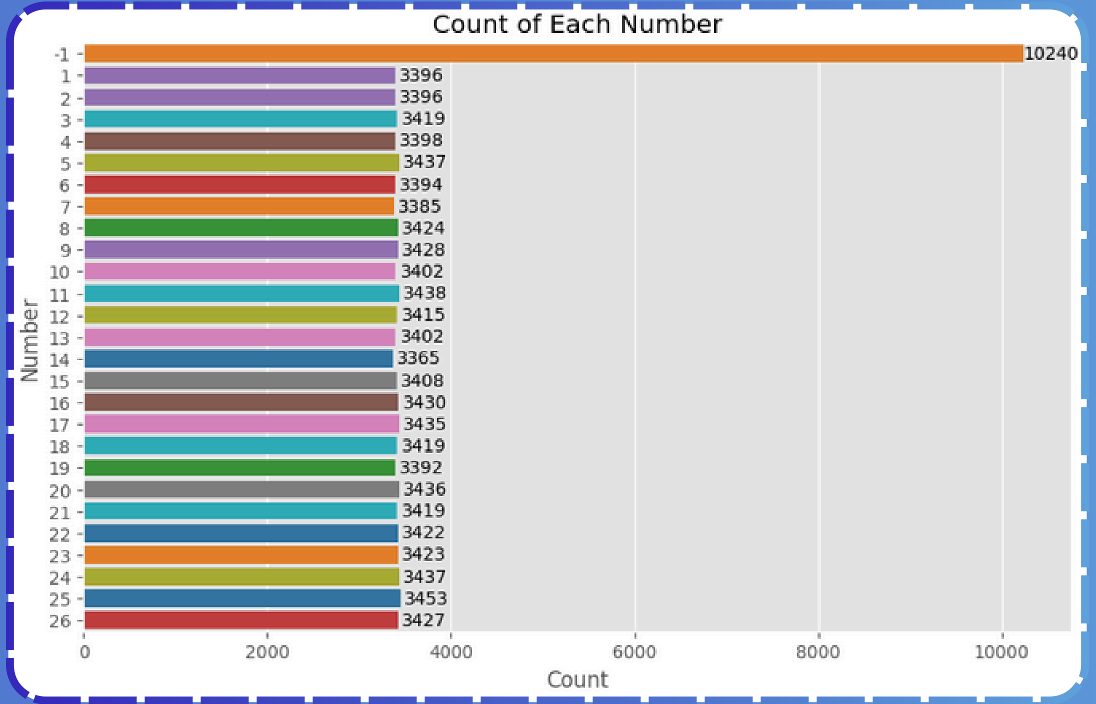
Average Pixel Images



t-SNE Distribution



Count of Labels



DATASET PROCESSING

DATA PROCESSING

Data Augmentation

Gaussian Noise

- A type of statistical noise that has a probability density function (PDF) equal to that of the normal distribution

Mean (μ) = 0
Standard Deviation (σ)

Salt & Pepper Noise

- It presents itself as randomly scattered white (salt) and black (pepper) pixels. It represents a type of impulsive noise

Density = Area Affected

MODEL SELECTION

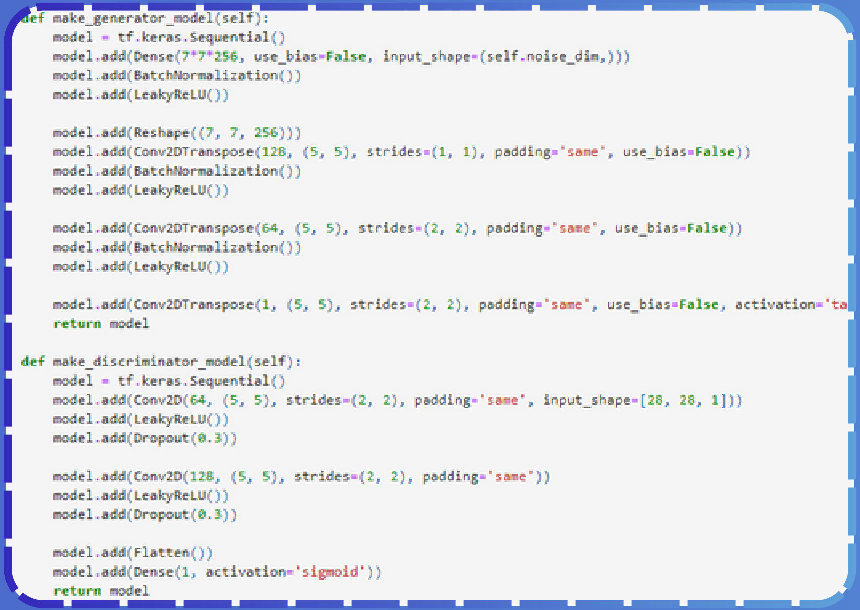
MODEL BUILDING (BASELINE)

Observation

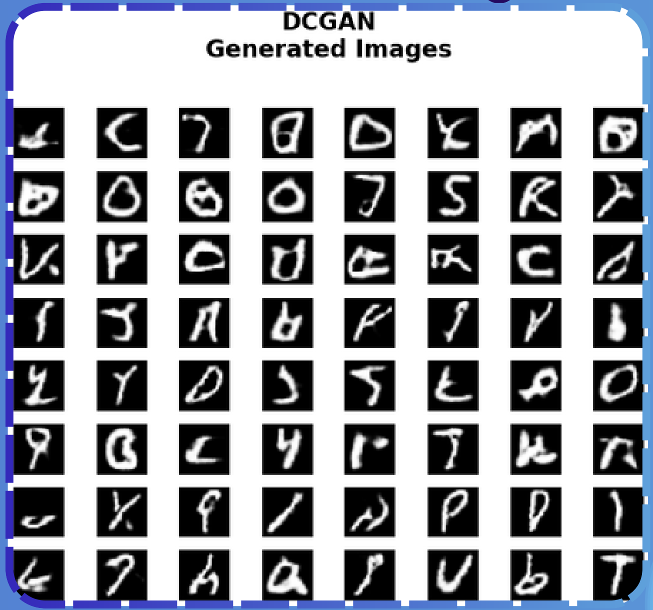
- D & G are diverging (Discriminator too strong)
- KL Divergence slowing decreasing
- FID fluctuating

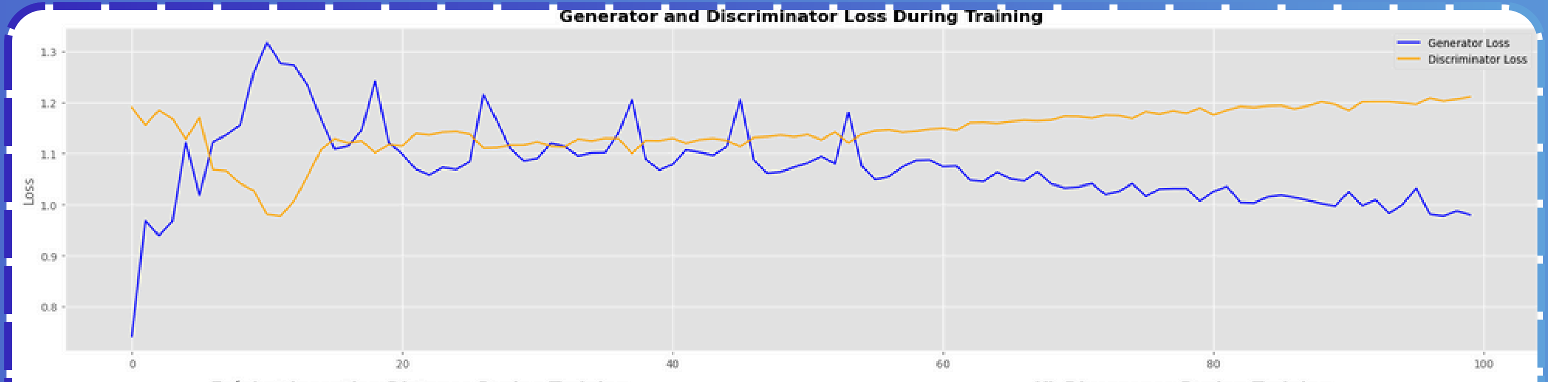
Model Architecture

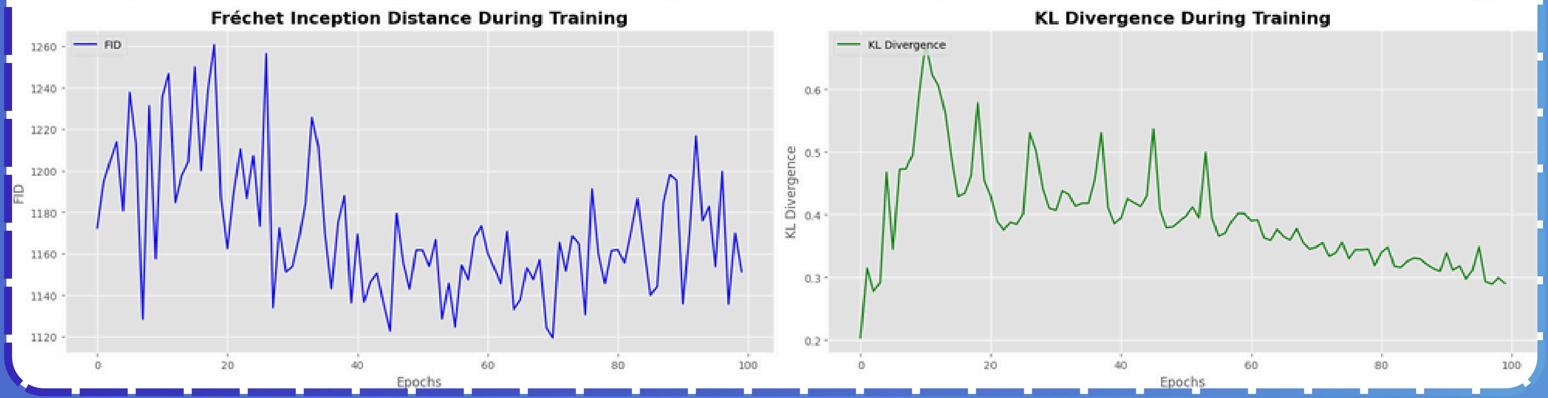
- Discriminator
  - 2 Conv Layer
- Generator
  - 3 Conv Layers



```
def make_generator_model(self):
    model = tf.keras.Sequential()
    model.add(Dense(7*7*256, use_bias=False, input_shape=(self.noise_dim,)))
    model.add(BatchNormalization())
    model.add(LeakyReLU())

    model.add(Reshape((7, 7, 256)))
    model.add(Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same', use_bias=False))
    model.add(BatchNormalization())
    model.add(LeakyReLU())

    model.add(Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use_bias=False))
    model.add(BatchNormalization())
    model.add(LeakyReLU())

    model.add(Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same', use_bias=False, activation='ta
    return model

def make_discriminator_model(self):
    model = tf.keras.Sequential()
    model.add(Conv2D(64, (5, 5), strides=(2, 2), padding='same', input_shape=[28, 28, 1]))
    model.add(LeakyReLU())
    model.add(Dropout(0.3))

    model.add(Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
    model.add(LeakyReLU())
    model.add(Dropout(0.3))

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







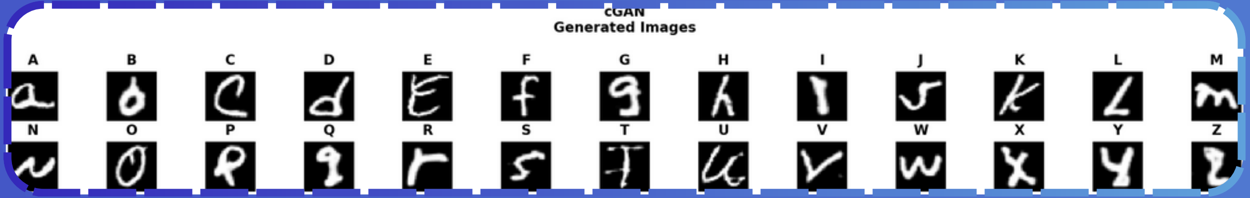
MODEL BUILDING (CGAN)

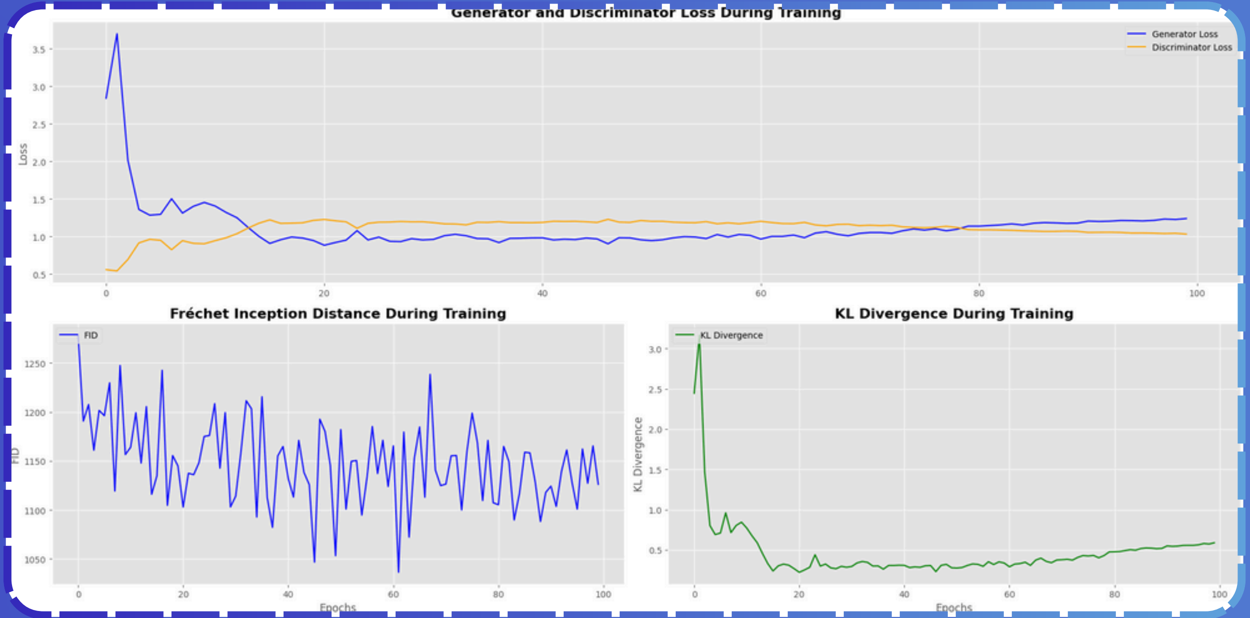
Observation

- Both losses stabilize after initial fluctuations
- KL Divergence dropped but slowly rises (Overfitted)
- FID score fluctuating

Model Architecture

- Both D & G take in the (Labels & Images) as Inputs.
- Same Architecture as Improved DCGAN






```python
def make_generator_model(self):
    noise = Input(shape=(self.noise_dim,))
    label = Input(shape=(self.num_classes,))
    inputs = Concatenate()([noise, label])

    # Initial dense layer
    x = Dense(7*7*256, use_bias=False)(inputs)
    x = BatchNormalization()(x)
    x = LeakyReLU(alpha=0.2)(x)
    x = Reshape((7, 7, 256))(x)

    # Residual block function
    def residual_block(input_tensor, filter_count, stride_size):
        skip_connection = input_tensor

        x = Conv2DTranspose(filter_count, (3, 3), strides=stride_size, padding='same', use_bias=False)(input_tensor)
        x = BatchNormalization()(x)
        x = LeakyReLU(alpha=0.2)(x)

        x = Conv2DTranspose(filter_count, (3, 3), strides=(1, 1), padding='same', use_bias=False)(x)
        x = BatchNormalization()(x)

        if stride_size != (1, 1) or int(skip_connection.shape[-1]) != filter_count:
            skip_connection = Conv2DTranspose(filter_count, (1, 1), strides=stride_size, padding='same', use_bias=False)(skip_connection)

        x = Add()([x, skip_connection])
        x = LeakyReLU(alpha=0.2)(x)
        return x

    # Apply residual blocks
    x = residual_block(x, 128, (2, 2))
    x = residual_block(x, 64, (2, 2))

    # Final convolution layers
    x = Conv2DTranspose(32, (3, 3), strides=(1, 1), padding='same', use_bias=False)(x)
    x = BatchNormalization()(x)
    x = LeakyReLU(alpha=0.2)(x)

    x = Conv2DTranspose(1, (5, 5), strides=(1, 1), padding='same', use_bias=False, activation='tanh')(x)

    model = Model([noise, label], x, name="Generator")
    return model

def make_discriminator_model(self):
    image = Input(shape=(28, 28, 1))
    label = Input(shape=(self.num_classes,))

    label_embedding = Dense(28*28)(label)
    label_embedding = Reshape((28, 28, 1))(label_embedding)

    inputs = Concatenate(axis=-1)([image, label_embedding])

    def conv_block(input_tensor, filter_count, stride_size):
        x = Conv2D(filter_count, (3, 3), strides=stride_size, padding='same')(input_tensor)
        x = LeakyReLU(alpha=0.2)(x)
        x = Conv2D(filter_count, (3, 3), strides=(1, 1), padding='same')(x)
        x = LeakyReLU(alpha=0.2)(x)
        return x

    # Apply convolutional blocks
    x = conv_block(inputs, 64, (2, 2))
    x = Dropout(0.3)(x)

    x = conv_block(x, 128, (2, 2))
    x = Dropout(0.3)(x)

    x = conv_block(x, 256, (2, 2))
    x = Dropout(0.3)(x)

    # Global average pooling
    x = GlobalAveragePooling2D()(x)

    # Dense layers
    x = Dense(512)(x)
    x = LeakyReLU(alpha=0.2)(x)
    x = Dropout(0.5)(x)

    x = Dense(256)(x)
    x = LeakyReLU(alpha=0.2)(x)
    x = Dropout(0.5)(x)

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

    model = Model([image, label], x, name="Discriminator")
    return model
```

12 MODEL BUILDING (IMPROVED)

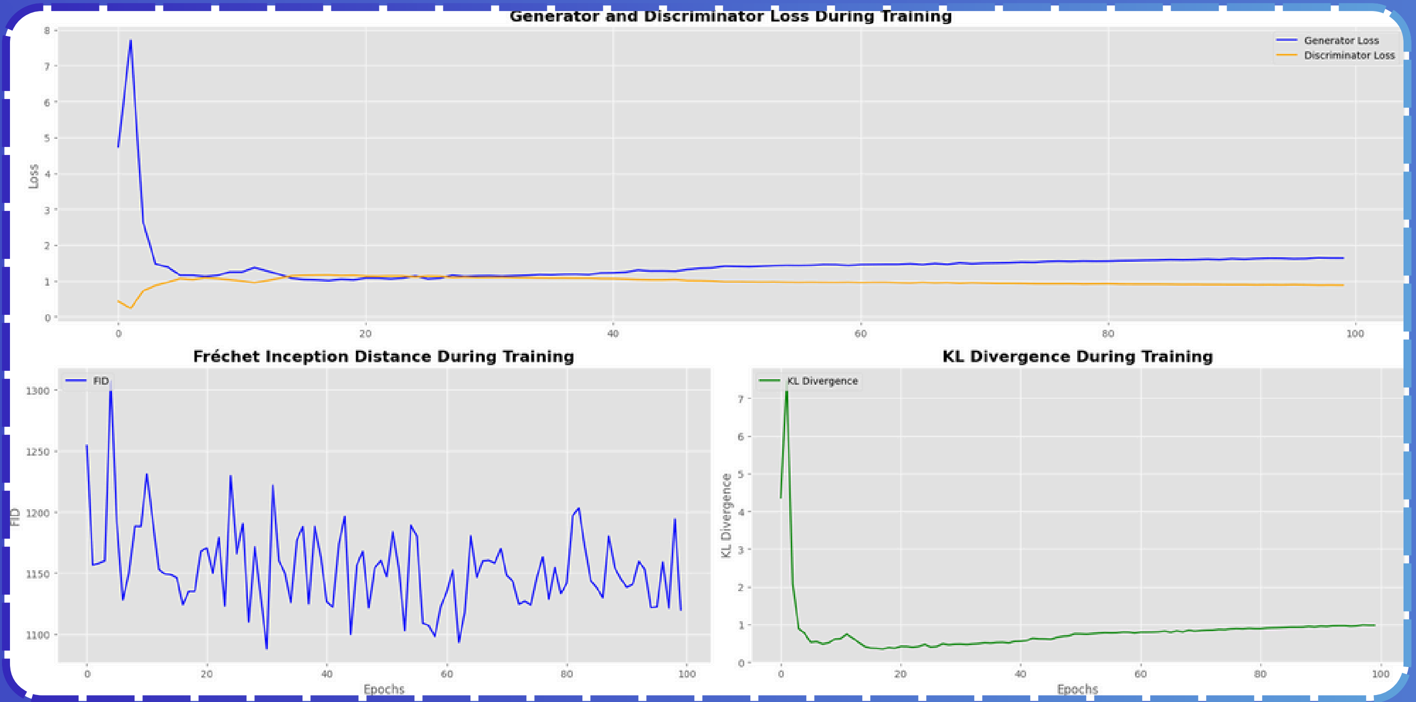
Observation

- D & G are diverging (Discriminator too strong)
- FID fluctuating
- KL Divergence shows **overfitting** (Big Drop then Rise)

Model Architecture

- Discriminator
  - 6 Conv Layer
- Generator
  - 7 Conv Layers

Added Global average pooling




```
make_generator_model(self):
    noise_input = Input(shape=(self.noise_dim,))

    # Initial dense layer
    initial_dense = Dense(7*7*256, use_bias=False)(noise_input)
    initial_bn = BatchNormalization()(initial_dense)
    initial_leaky = LeakyReLU(alpha=0.2)(initial_bn)

    reshaped_features = Reshape((7, 7, 256))(initial_leaky)

    # Transposed Convolution blocks with residual connections
    def residual_block(input_tensor, filter_count, stride_size):
        skip_connection = input_tensor

        conv1 = Conv2DTranspose(filter_count, (3, 3), strides=stride_size, padding='same', use_bias=False)(input_tensor)
        bn1 = BatchNormalization()(conv1)
        leaky1 = LeakyReLU(alpha=0.2)(bn1)

        conv2 = Conv2DTranspose(filter_count, (3, 3), strides=(1, 1), padding='same', use_bias=False)(leaky1)
        bn2 = BatchNormalization()(conv2)

        if stride_size != (1, 1) or int(skip_connection.shape[-1]) != filter_count:
            skip_connection = Conv2DTranspose(filter_count, (1, 1), strides=stride_size, padding='same', use_bias=False)(skip_connection)

        added = Add()([bn2, skip_connection])
        output = LeakyReLU(alpha=0.2)(added)
        return output

    block1 = residual_block(reshaped_features, 128, (2, 2))
    block2 = residual_block(block1, 64, (2, 2))

    # Final convolution
    final_conv = Conv2DTranspose(32, (3, 3), strides=(1, 1), padding='same', use_bias=False)(block2)
    final_bn = BatchNormalization()(final_conv)
    final_leaky = LeakyReLU(alpha=0.2)(final_bn)

    output_img = Conv2DTranspose(1, (5, 5), strides=(1, 1), padding='same', use_bias=False, activation='tanh')(final_leaky)

    return Model(inputs=noise_input, outputs=output_img, name="Generator")

def make_discriminator_model(self):
    img_input = Input(shape=[28, 28, 1])

    def conv_block(input_tensor, filter_count, stride_size):
        conv1 = Conv2D(filter_count, (3, 3), strides=stride_size, padding='same')(input_tensor)
        leaky1 = LeakyReLU(alpha=0.2)(conv1)
        conv2 = Conv2D(filter_count, (3, 3), strides=(1, 1), padding='same')(leaky1)
        leaky2 = LeakyReLU(alpha=0.2)(conv2)
        return leaky2

    # Convolutional blocks
    conv_block1 = conv_block(img_input, 64, (2, 2))
    dropout1 = Dropout(0.3)(conv_block1)

    conv_block2 = conv_block(dropout1, 128, (2, 2))
    dropout2 = Dropout(0.3)(conv_block2)

    conv_block3 = conv_block(dropout2, 256, (2, 2))
    dropout3 = Dropout(0.3)(conv_block3)

    # Global average pooling
    global_pool = GlobalAveragePooling2D()(dropout3)

    # Dense Layers
    dense1 = Dense(512)(global_pool)
    leaky_dense1 = LeakyReLU(alpha=0.2)(dense1)
    dropout4 = Dropout(0.5)(leaky_dense1)

    dense2 = Dense(256)(dropout4)
    leaky_dense2 = LeakyReLU(alpha=0.2)(dense2)
    dropout5 = Dropout(0.5)(leaky_dense2)

    validity = Dense(1, activation='sigmoid')(dropout5)

    return Model(inputs=img_input, outputs=validity, name="Discriminator")
```

MODEL BUILDING (TRIPLE-GAN)

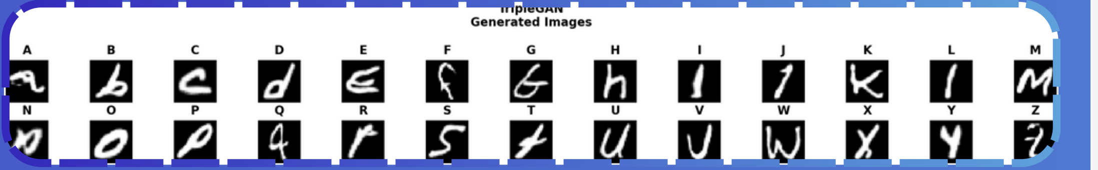
Observation

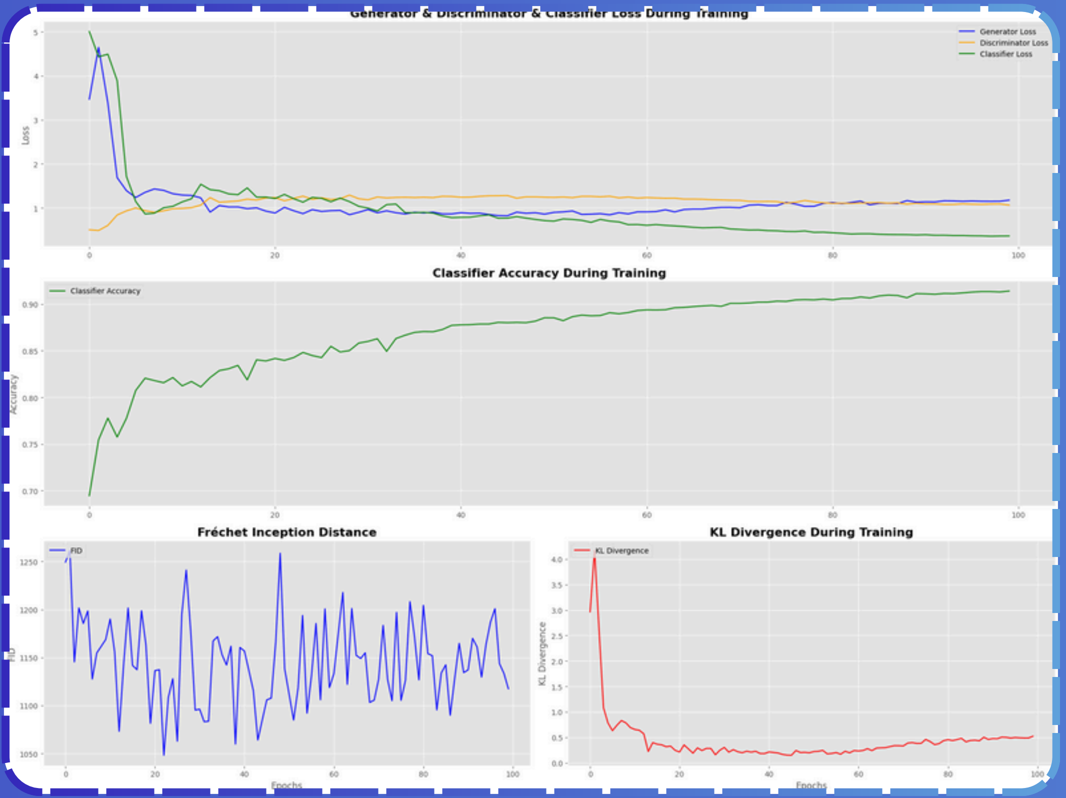
- Both losses stabilize after initial fluctuations
- KL Divergence is very very high.
- FID score fluctuating

Model Architecture

- Both D & G take in the (Labels & Images) as Inputs.
- Same Architecture as Improved DCGAN
- **Classifier** help provide additional form of regularization

```python
def make_classifier_model(self):
    image = Input(shape=(28, 28, 1))

    x = Conv2D(32, (3, 3), strides=(1, 1), padding='same')(image)
    x = LeakyReLU(alpha=0.2)(x)
    x = Conv2D(64, (3, 3), strides=(2, 2), padding='same')(x)
    x = LeakyReLU(alpha=0.2)(x)
    x = Dropout(0.3)(x)

    x = Conv2D(128, (3, 3), strides=(2, 2), padding='same')(x)
    x = LeakyReLU(alpha=0.2)(x)
    x = Dropout(0.3)(x)

    x = Flatten()(x)
    x = Dense(256)(x)
    x = LeakyReLU(alpha=0.2)(x)
    x = Dropout(0.5)(x)

    x = Dense(self.num_classes, activation='softmax')(x)

    model = Model(image, x, name="Classifier")
    return model
```






```python
def make_generator_model(self):
    noise = Input(shape=(self.noise_dim,))
    label = Input(shape=(self.num_classes,))
    inputs = Concatenate()([noise, label])

    x = Dense(7*7*256, use_bias=False)(inputs)
    x = BatchNormalization()(x)
    x = LeakyReLU(alpha=0.2)(x)
    x = Reshape((7, 7, 256))(x)

    def residual_block(input_tensor, filter_count, stride_size):
        skip_connection = input_tensor

        x = Conv2DTranspose(filter_count, (3, 3), strides=stride_size, padding='same', use_bias=False)(input_tensor)
        x = BatchNormalization()(x)
        x = LeakyReLU(alpha=0.2)(x)

        x = Conv2DTranspose(filter_count, (3, 3), strides=(1, 1), padding='same', use_bias=False)(x)
        x = BatchNormalization()(x)

        if stride_size != (1, 1) or int(skip_connection.shape[-1]) != filter_count:
            skip_connection = Conv2DTranspose(filter_count, (1, 1), strides=stride_size, padding='same', use_bias=False)(skip_connection)

        x = Add()([x, skip_connection])
        x = LeakyReLU(alpha=0.2)(x)
        return x

    x = residual_block(x, 128, (2, 2))
    x = residual_block(x, 64, (2, 2))

    x = Conv2DTranspose(32, (3, 3), strides=(1, 1), padding='same', use_bias=False)(x)
    x = BatchNormalization()(x)
    x = LeakyReLU(alpha=0.2)(x)

    x = Conv2DTranspose(1, (5, 5), strides=(1, 1), padding='same', use_bias=False, activation='tanh')(x)

    model = Model([noise, label], x, name="Generator")
    return model
```

```python
def make_discriminator_model(self):
    image = Input(shape=(28, 28, 1))
    label = Input(shape=(self.num_classes,))

    label_embedding = Dense(28*28)(label)
    label_embedding = Reshape((28, 28, 1))(label_embedding)

    inputs = Concatenate(axis=-1)([image, label_embedding])

    def conv_block(input_tensor, filter_count, stride_size):
        x = Conv2D(filter_count, (3, 3), strides=stride_size, padding='same')(input_tensor)
        x = LeakyReLU(alpha=0.2)(x)
        x = Conv2D(filter_count, (3, 3), strides=(1, 1), padding='same')(x)
        x = LeakyReLU(alpha=0.2)(x)
        return x

    x = conv_block(inputs, 64, (2, 2))
    x = Dropout(0.3)(x)

    x = conv_block(x, 128, (2, 2))
    x = Dropout(0.3)(x)

    x = conv_block(x, 256, (2, 2))
    x = Dropout(0.3)(x)

    x = GlobalAveragePooling2D()(x)

    x = Dense(512)(x)
    x = LeakyReLU(alpha=0.2)(x)
    x = Dropout(0.5)(x)

    x = Dense(256)(x)
    x = LeakyReLU(alpha=0.2)(x)
    x = Dropout(0.5)(x)

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

    model = Model([image, label], x, name="Discriminator")
    return model
```

OVERVIEW

Fréchet Inception Distance (FID)

- FID measures the distance between feature vectors of real and generated images, extracted using a pre-trained Inception network.
- FID range (0 - ∞), where lower is more alike

$$FID = \|\mu - \mu_w\|^2 + tr(\Sigma + \Sigma_w - 2(\Sigma\Sigma_w)^{\frac{1}{2}})$$

However, the Inception Model is trained on a different dataset. Hence, it may not be the best evaluation matrix

Kullback–Leibler (KL) Divergence

- It measure how the distribution of generated samples diverges from the distribution of real samples.

$$D_{KL}(p|q) = \Sigma_x p(x) \log \frac{p(x)}{q(x)}$$

	Model Type	Final FID	Final KL Divergence
0	DCGAN	1151.201036-0.000001j	0.290436
1	Enhanced DCGAN	1119.844249-0.000001j	0.975994
2	cGAN	1126.058124-0.000001j	0.587908
3	TripleGAN	1117.314748-0.000001j	0.522372

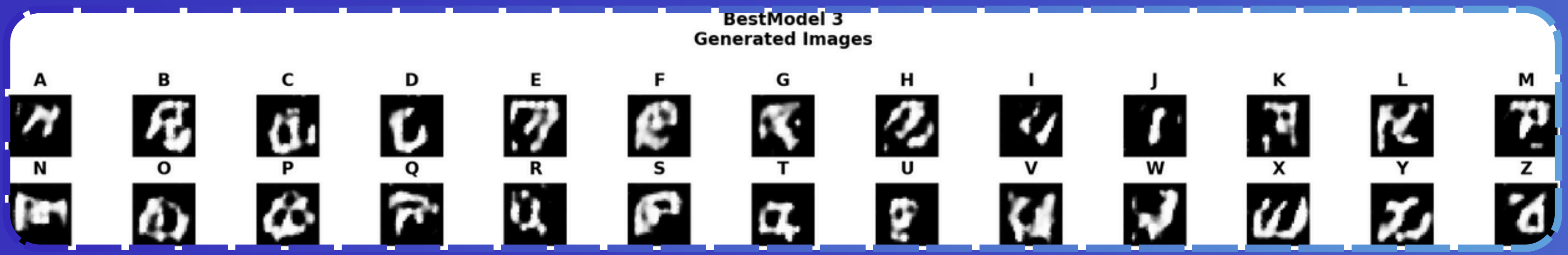
MODEL IMPROVEMENT

MODEL IMPROVEMENT

Spectral Normalization

- Spectral Normalization (SN) is a regularization technique used to stabilize the training of deep neural networks. It basically addresses the most critical features of the generated images.

However, our dataset was too simple. Hence, when we implemented SN, our model focus on the obvious errors that it did not create high quality images.

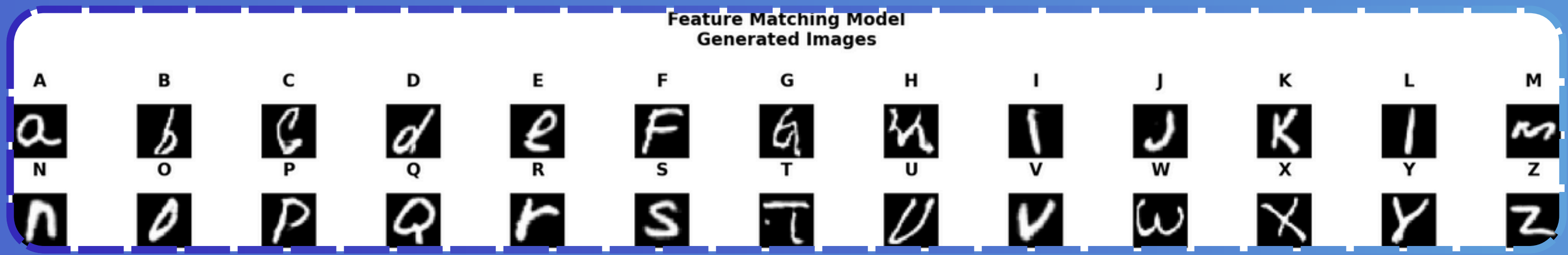


Feature Matching

- Feature Matching is a technique used to improve the stability and quality of generated images.
- It basically points out the mistakes of the images, instead of just the discriminator saying a binary output and the generator randomly changes.

Feature Extraction:

- A pre-trained feature extractor is used to extract features from both real and generated images



SUMMARY

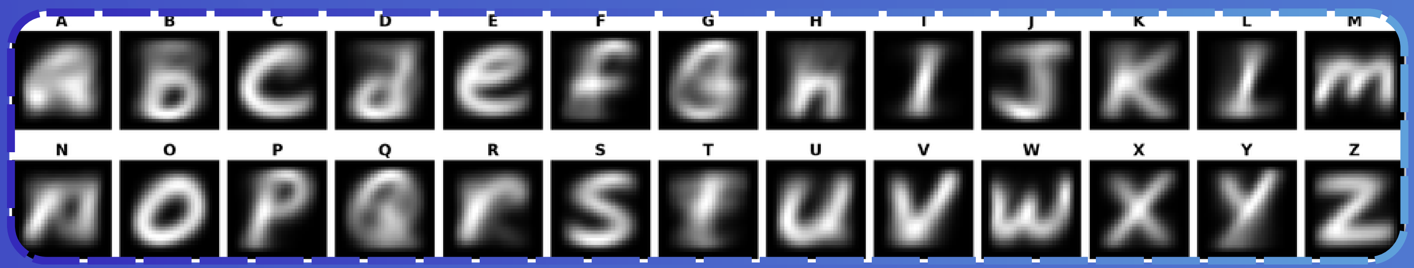
Generating Images of a Specific Class

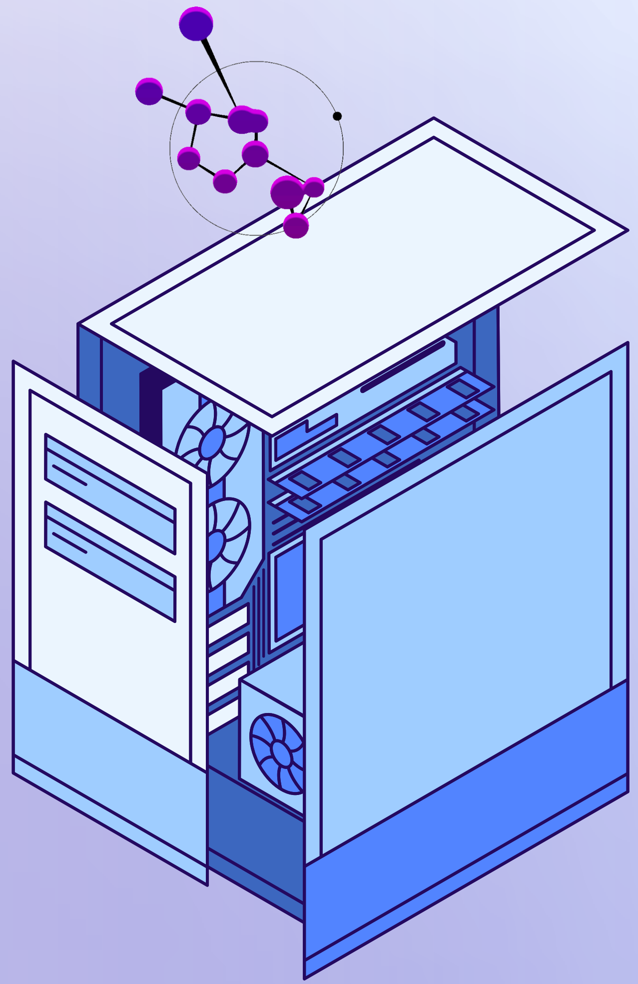
- Use of TripleGAN / cGAN as they can take in labels of the images
- Train the DCGAN on each class seperately

Classes That Are Relatively Easier/Harder to Generate

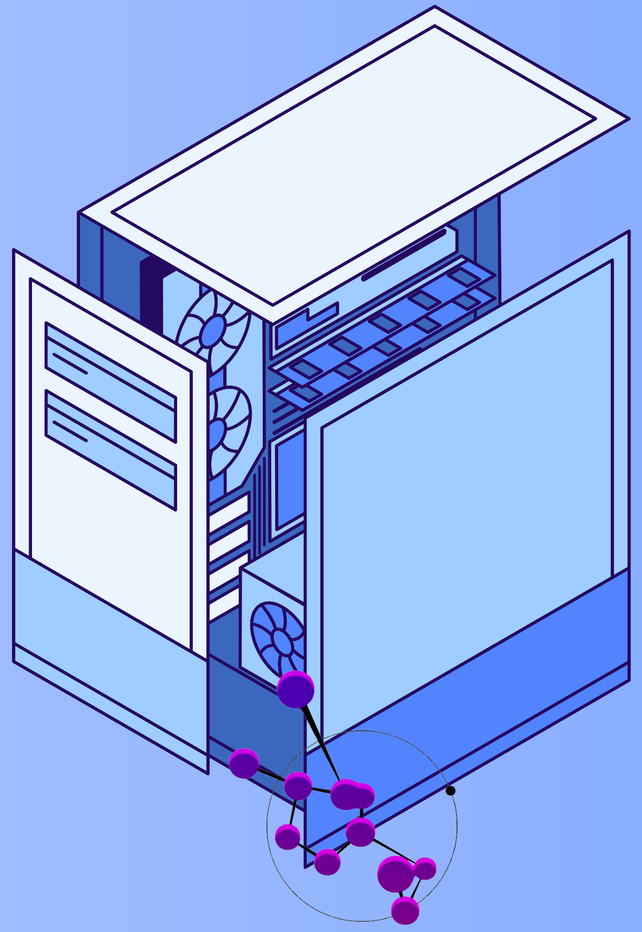
- Easier Classes are like simple, well-defined characters with fewer strokes, such as 'I', 'L', 'O', and 'T'. While more complex characters with multiple strokes and curves, such as 'M', 'W', 'K', and 'R' have more intricate details, making it harder.

However, its not just about the strokes, our dataset has both uppercase & lowercase letters. The blurier the images, the more complex the class is.





REINFORCEMENT LEANRING



BACKGROUND INFO

Balance a Pendulum

Use Cases

- This task are critical in robotics, such as for self-balancing robots and robotic manipulators, as well as in aerospace engineering for stabilizing rockets and maintaining satellite orientation.

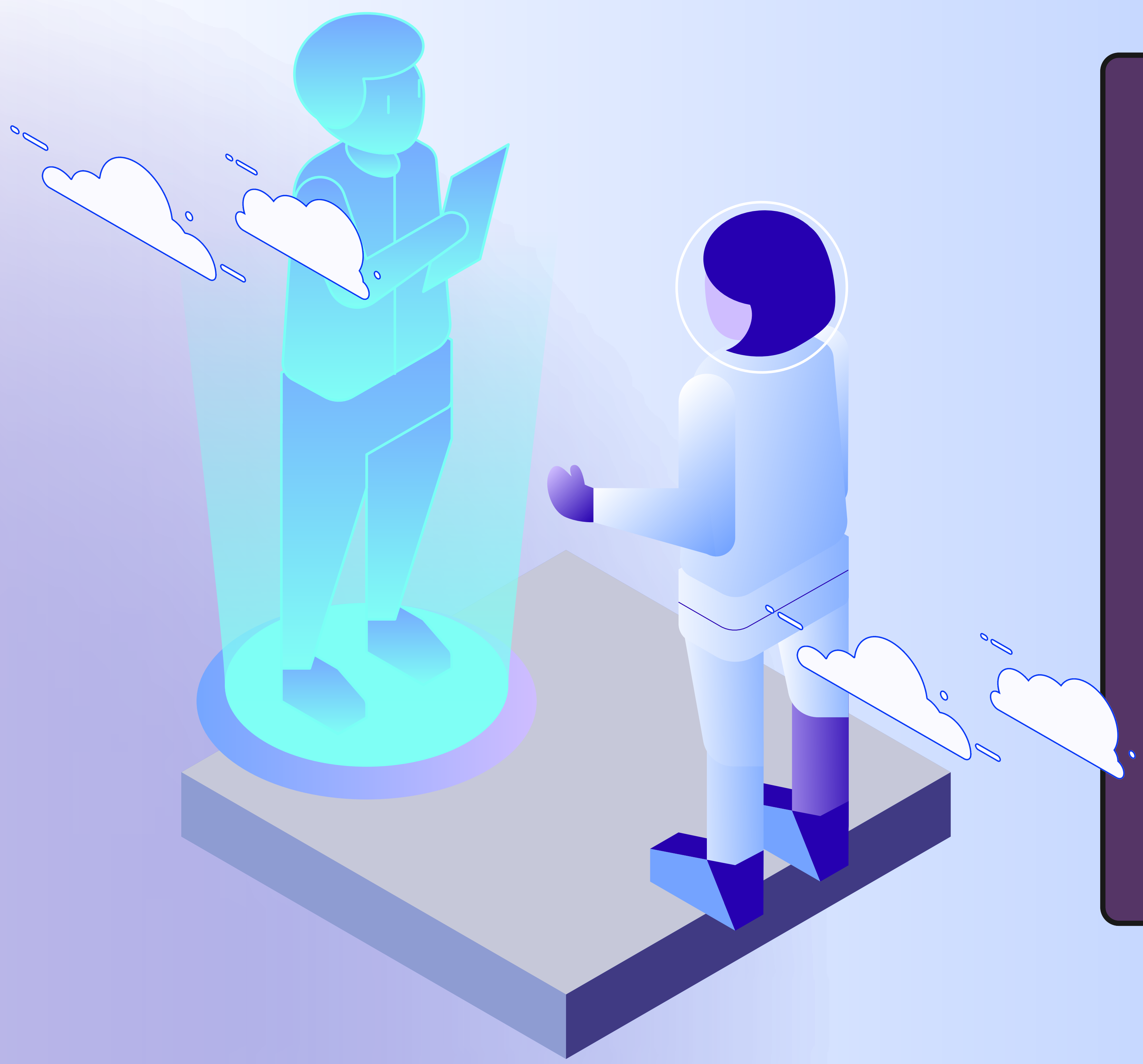
Reinforcement Learning

- Challenges we may face
    - Exploration vs. Exploitation (Stability)

Gymnasium

- We will be using a virtual environment from the gymnasium (Pendulum v0) to train our model.

TABLE OF CONTENTS



- Environment Exploration
- Model Selection
- Model Improvement

ENVIRONMENT EXPLORATION

ENVIRONMENT INFORMATION

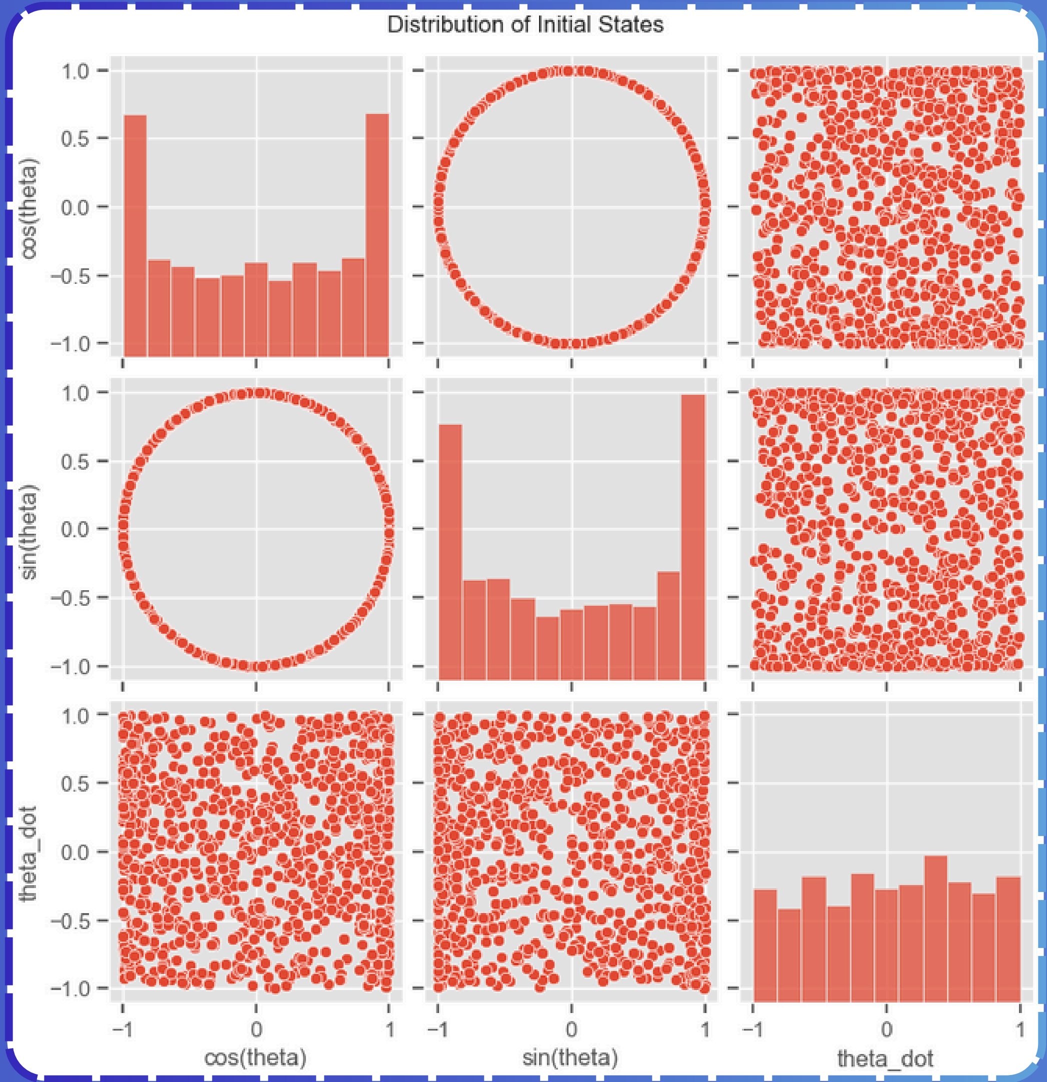
Pendulum v0

- **x-y:**
  - cartesian coordinates of the pendulum's end in meters.
- **theta :**
  - angle in radians.
- **tau:**
  - torque in N m. Defined as positive counter-clockwise.

Objective

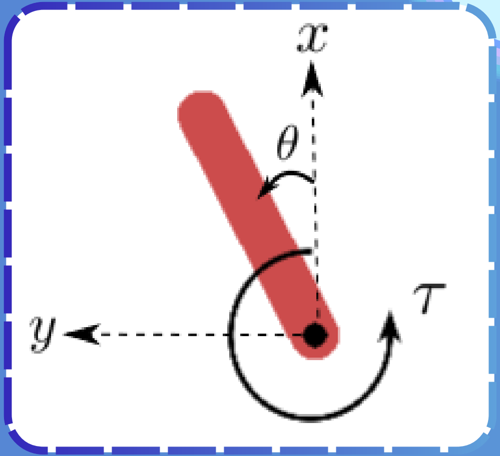
Swing up and balance a pendulum and Maximize cumulative reward by maintaining the pendulum upright.

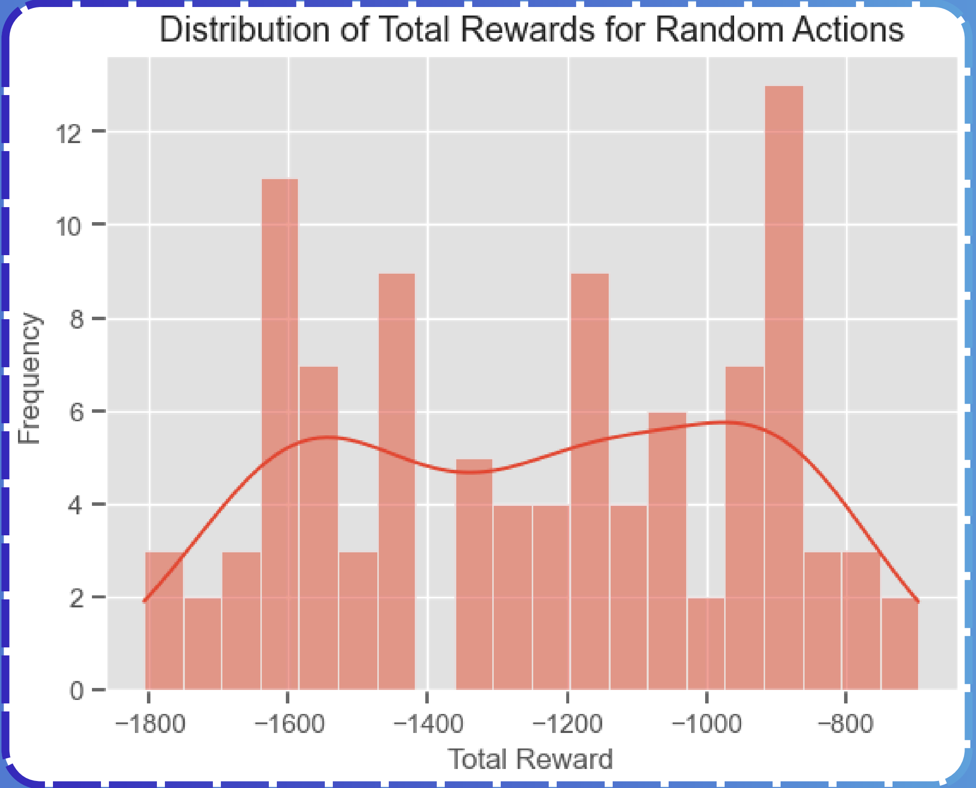
Initial States



Action	Min	Max
Torque	-2.0	2.0

Observation	Min	Max
x = cos(theta)	-1.0	1.0
y = sin(angle)	-1.0	1.0
Angular Velocity	-8.0	8.0





MODEL SELECTION

MODEL BUILDING (BASELINE)

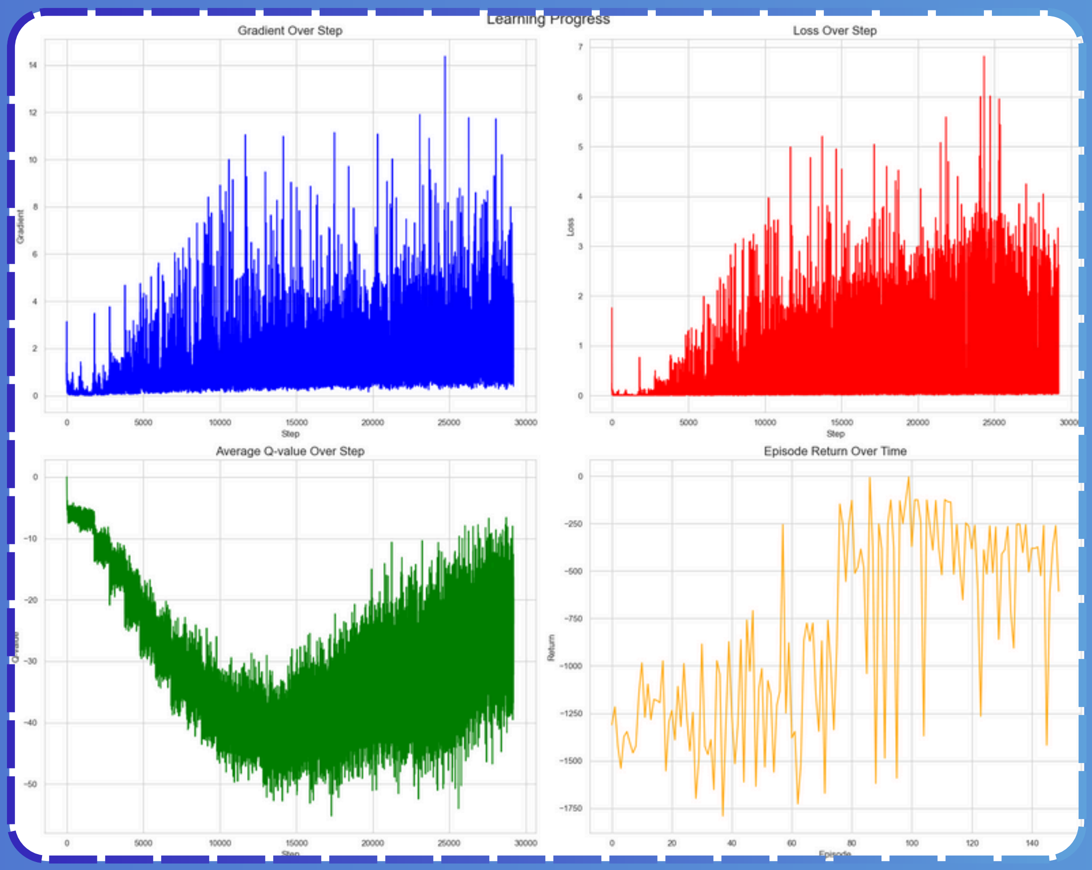
Key Components of DQN

- **Q-Network**
  - Input Layer: The current state of the environment.
  - Hidden Layers: Several layers to extract features from the state.
  - Output Layer: Q-values for each possible action.
- **Experience Replay**
  - Replay Buffer: Stores agent's experiences (s, a, r, s').
  - Batch Training: Random mini-batches sampled from the buffer for training.
- **Target Network**
  - Purpose: Provides stable Q-value targets.
  - Target Update: Updated periodically with the Q-Network's weights.

Observation

- Gradient shows high variability, with spikes up to 14.
- Loss gradually increases and stabilizes around 2-3.
- Average Q-value initially decreases, then increases
- Episode Return is highly variable, but showed an overall upward trend.

Gradient → Blue
Loss → Red
Q-value → Green
Return → Yellow



MODEL BUILDING (NOISY DQN)

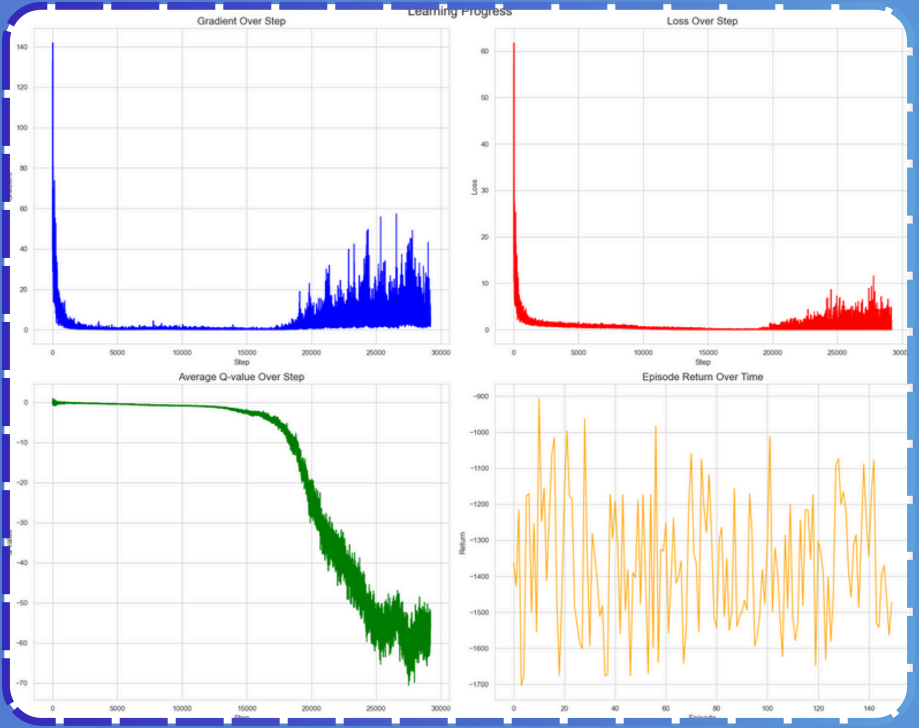
Key Components of Noisy DQN

- **Noisy Q-Network**
  - Input Layer: The current state of the environment.
  - Hidden Layers: Several layers to extract features from the state.
  - Output Layer: Q-values for each possible action, with added noise to encourage exploration.
- **Noisy Layers**
  - Noise Injection: Gaussian noise is added to the weights of the neural network to create a stochastic policy.
  - Learnable Parameters: The parameters of the noise distribution are learned during training, enabling adaptive exploration.

Observation

- Gradient initially high, then stabilizes at a low level before increasing again.
- Loss starts very high, quickly drops and remains low.
- Average Q-valueremains stable until around step 15000, then drops sharply.
- Episode Return fluctuates but shows no clear improvement trend.

Gradient → Blue
Loss → Red
Q-value → Green
Return → Yellow



MODEL BUILDING (DDPG)

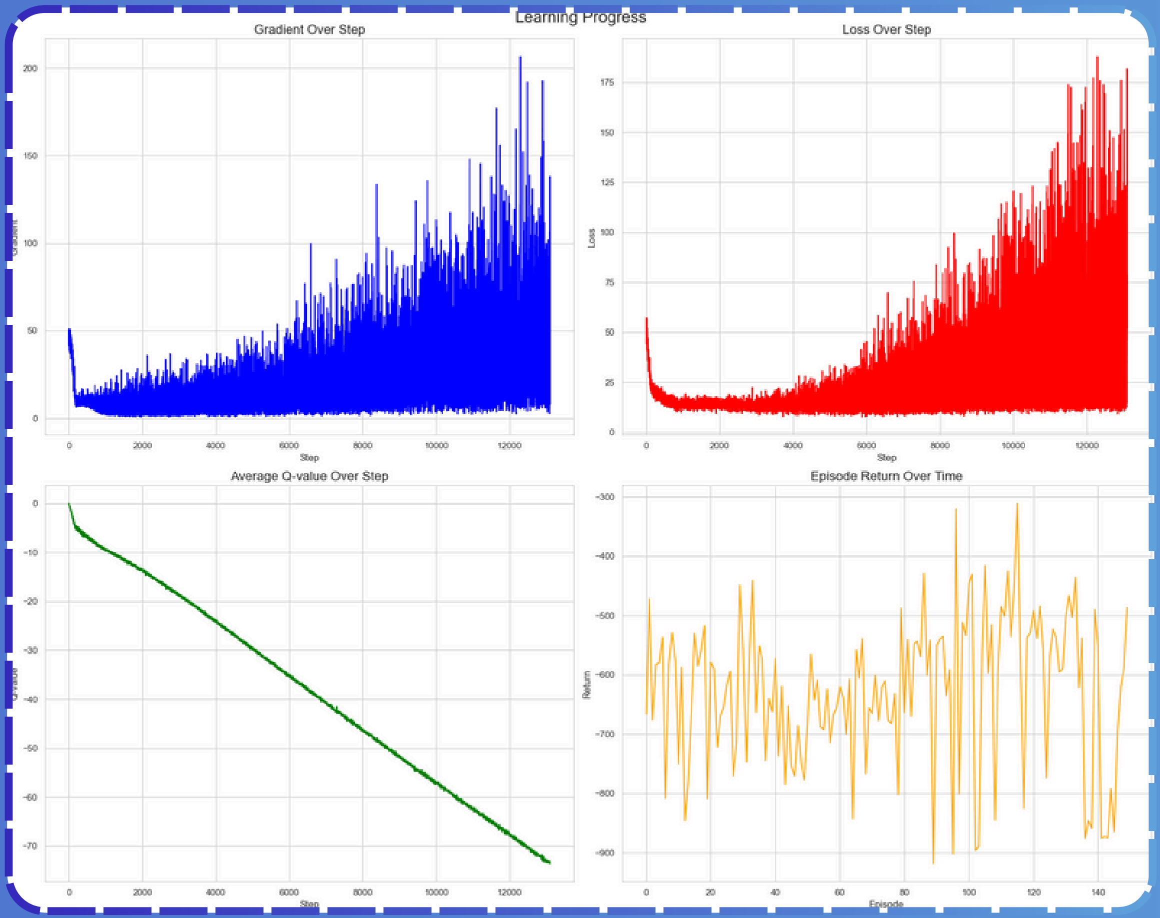
Key Components of DDPG

- **Actor Network**
  - Input Layer: Receives the current state.
  - Hidden Layers: Extract features from the state.
  - Output Layer: Outputs a specific action for the given state.
- **Critic Network**
  - Input Layer: Receives the state and action.
  - Hidden Layers: Processes state-action pairs.
  - Output Layer: Outputs the Q-value representing the value of the action-state pair.

Observation

- Gradient gradually increases with high variability.
- Loss initially drops, then steadily increases.
- Average Q-value consistently decreases linearly.
- Episode Return fluctuates but shows a slight upward trend.

Gradient → Blue
Loss → Red
Q-value → Green
Return → Yellow



OVERVIEW

Evaluation:

- Gradient (How quick it learns)
- Average Return (How well it does)
- Q-value (How well its learning)

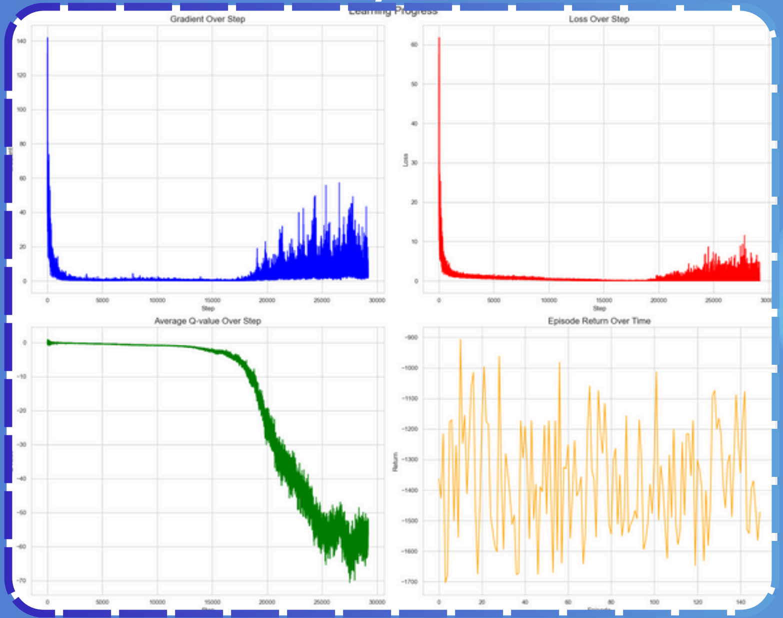
#1st DDPG seems to be have the lowest average return
#2nd Baseline is the most stable
#3rd NoisyDQN is the worst (Explored too much)

Although DDPG isn't very stable, with the right hypertuning & modifications to it, it can be more stable.

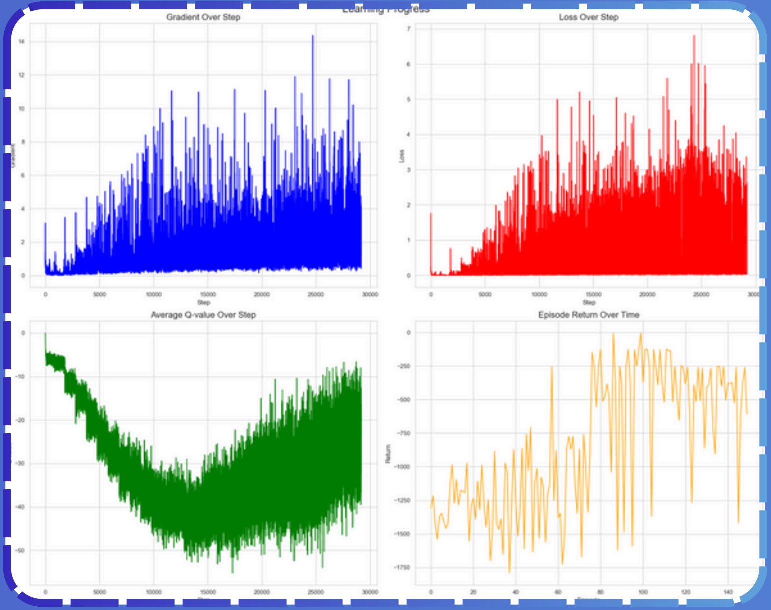
Average Return

	Model	Average Return
0	Baseline	-838.130000
1	NoisyDQN	-1367.530000
2	DDPG	-627.460000

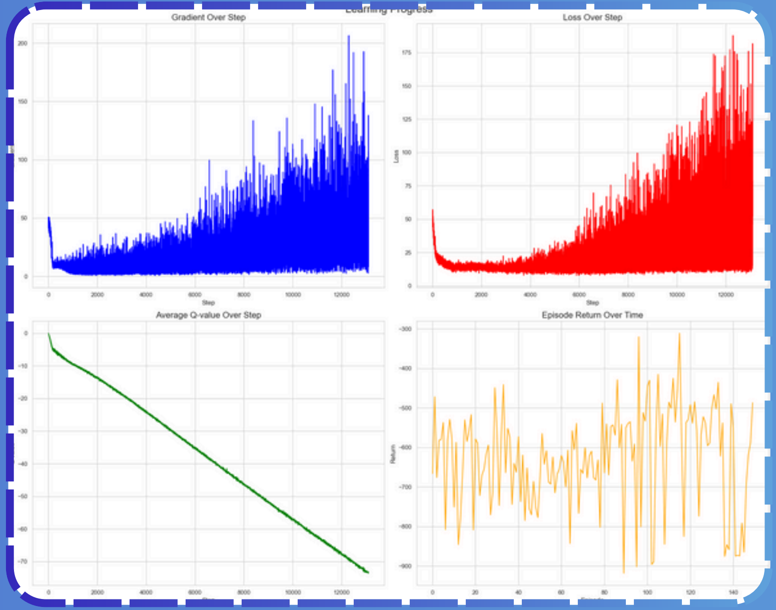
Noisy DQN



Baseline



DDPG



MODEL IMPROVEMENT

MODEL IMPROVEMENT

Hyperparameter Tuning

Tuning the default parameters of the model to best fit the data given

Parameter Type	Parameter Value	Average Return
Learning Rate	0.001	-625.660183
Learning Rate	0.010	-728.164443
Learning Rate	0.100	-676.745119
Gamma	0.900	-588.597900
Gamma	0.950	-608.660076
Gamma	0.990	-665.686798
Batch Size	16.000	-586.760901
Batch Size	32.000	-631.070504
Batch Size	64.000	-616.376262
Intermediate Size	32.000	-668.598781
Intermediate Size	64.000	-683.445408
Intermediate Size	128.000	-676.375155

Parameters:

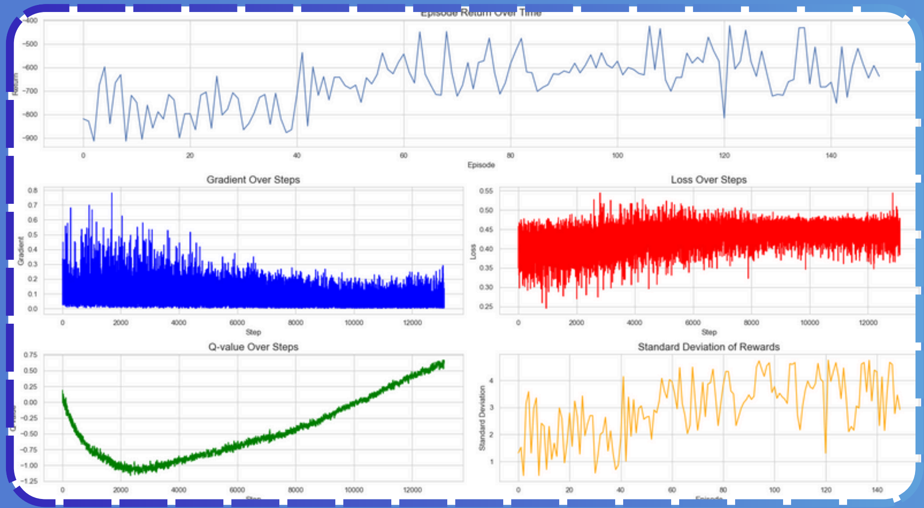
- Learning Rate
- Gamma
- Batch Size
- Intermediate Size

Ornstein-Uhlenbeck Process (OU Noise)

The OU process is a type of noise that is temporally correlated. It adds noise to the action outputs. This helps to **improve exploration** of the action space by the agent.

$$dx_t = \theta(\mu - x_t)dt + \sigma dW_t$$

However, looking at the results, it still seems to be exploiting the environment still.



SUMMARY

Systematically Optimize Solutions:

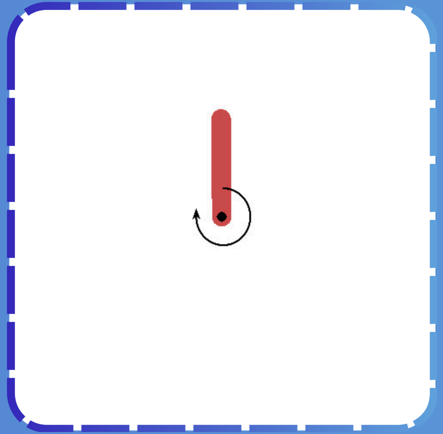
- Implement a structured approach such as grid search & random search to explore the hyperparameter space.

Hyper Parameters

- Learning Rate
- Gamma (Discount Factor)
- Batch Size
- Intermediate Size

Criteria Used

- Gradient Magnitude
- Standard Deviation of return



In reinforcement learning, Its hard to determine a good model between the best model. Due to the randomness of the environment. However, a good model should average around -100 return

THANK YOU