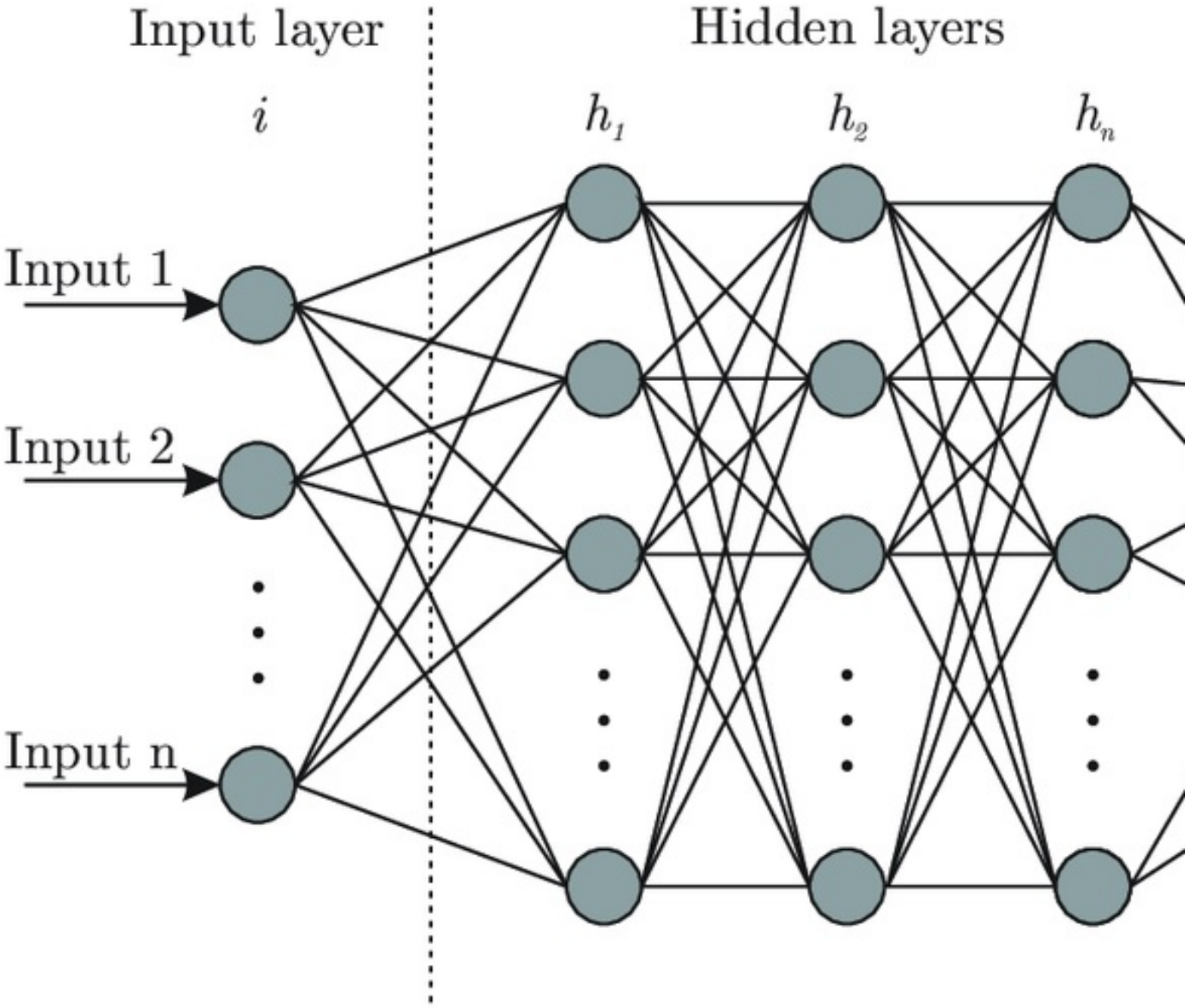
Assignment ML

ANS 1 :

Neural network

Neural networks are computational models that work like neural networks in the human brain process information. They consist of layers of neurons that transform the input data into meaningful outputs .



Types of neural networks :

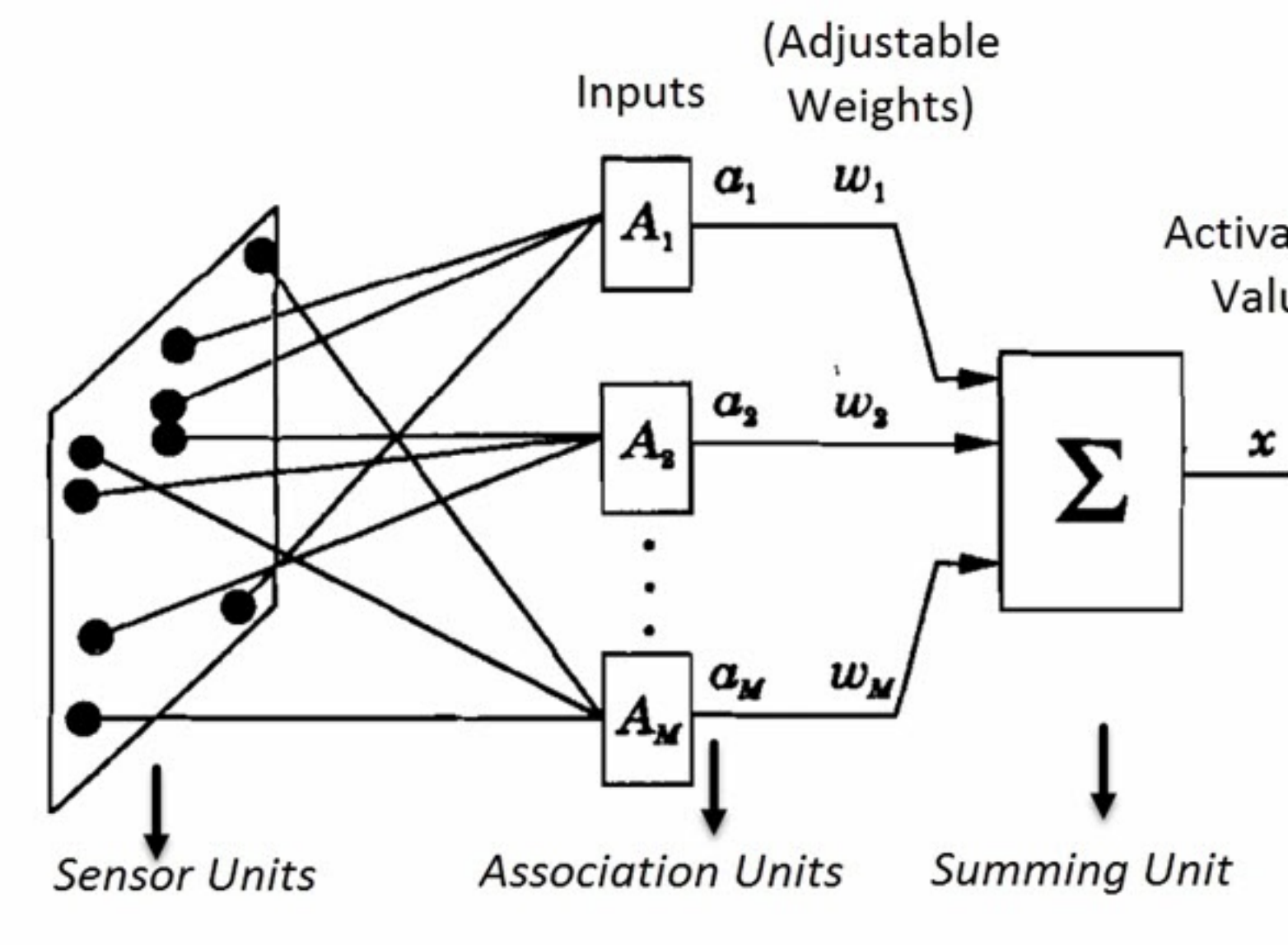
1. **ANN (Artificial Neural Network):**

Used for regression and classification

It is a type of computational model that is inspired by the structure and function of the human brain, composed of interconnected nodes or "neurons" that process and transmit information .

They are used for solving complex machine learning problems such as image classification, recommendation systems, and language-to-language translation.

They are now using in various applications , including image recognition , speech recognition , and natural language processing .



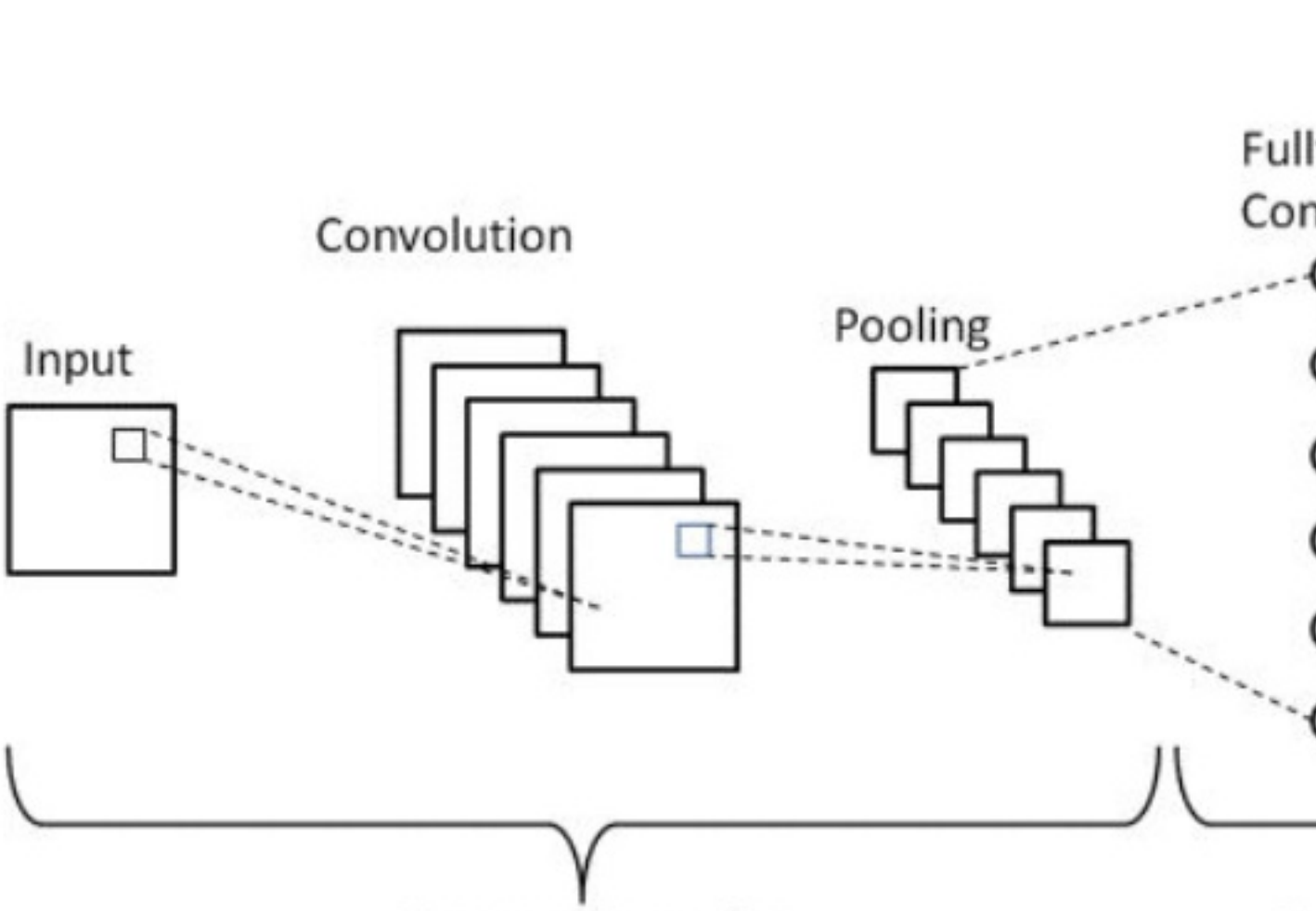
1. **CNN (Convolutional Neural Networks):**

Used for image classification

It is a type of neural network architecture that is particularly well-suited for image and video processing tasks. They are inspired by the structure and function

There are two main parts to a CNN architecture

- A convolution tool that separates and identifies th
  for analysis in a process called as Feature Extract
- The network of feature extraction consists of mar
  pooling layers.
- A fully connected layer that utilizes the output fro
  predicts the class of the image based on the featu
  stages.
- This CNN model of feature extraction aims to redu
  present in a dataset. It creates new features whic
  features contained in an original set of features. T
  shown in the basic CNN architecture with diagram

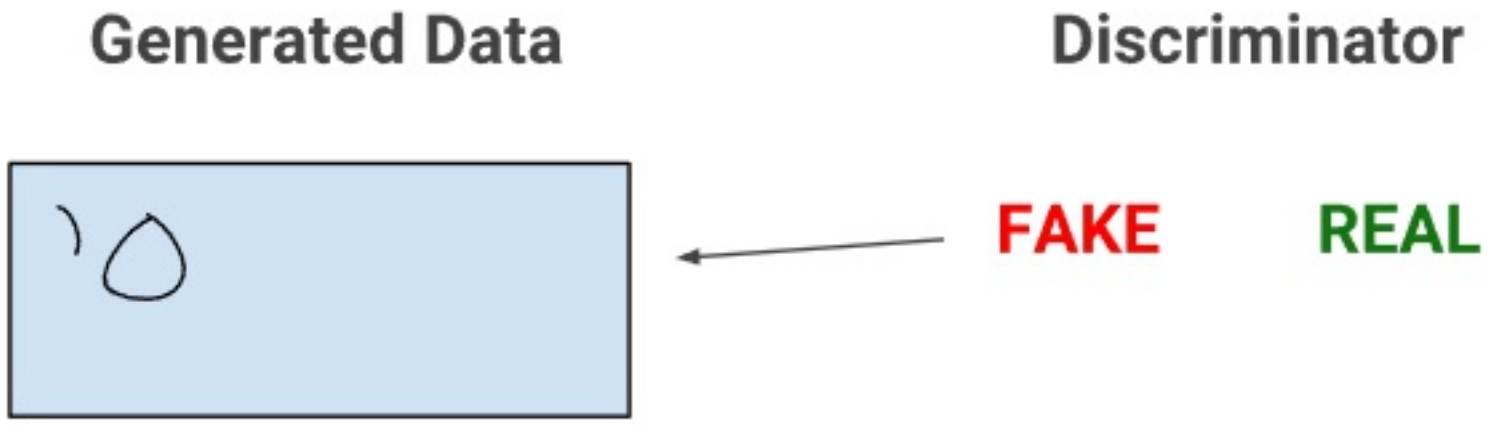


They are needed for image classification , object detection , and image segmentation tasks .

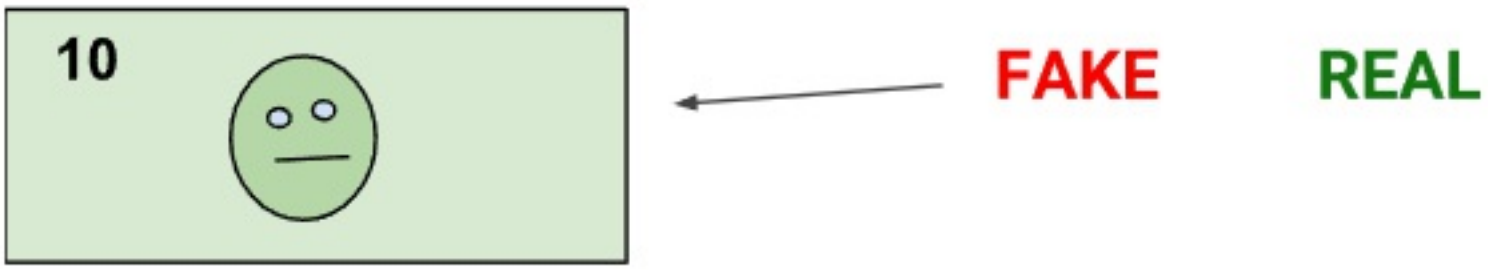
They are now used in applications like self-driving cars , facial recognition , and medical image analysis .

1. **GAN (Generative adversarial networks):**

It is a type of deep learning algorithm that consists of two neural networks: a **Generator** and a **Discriminator**. The generator creates new data samples that are similar to a given dataset, while the discriminator evaluates the generated samples and tells the generator whether they are realistic or not.



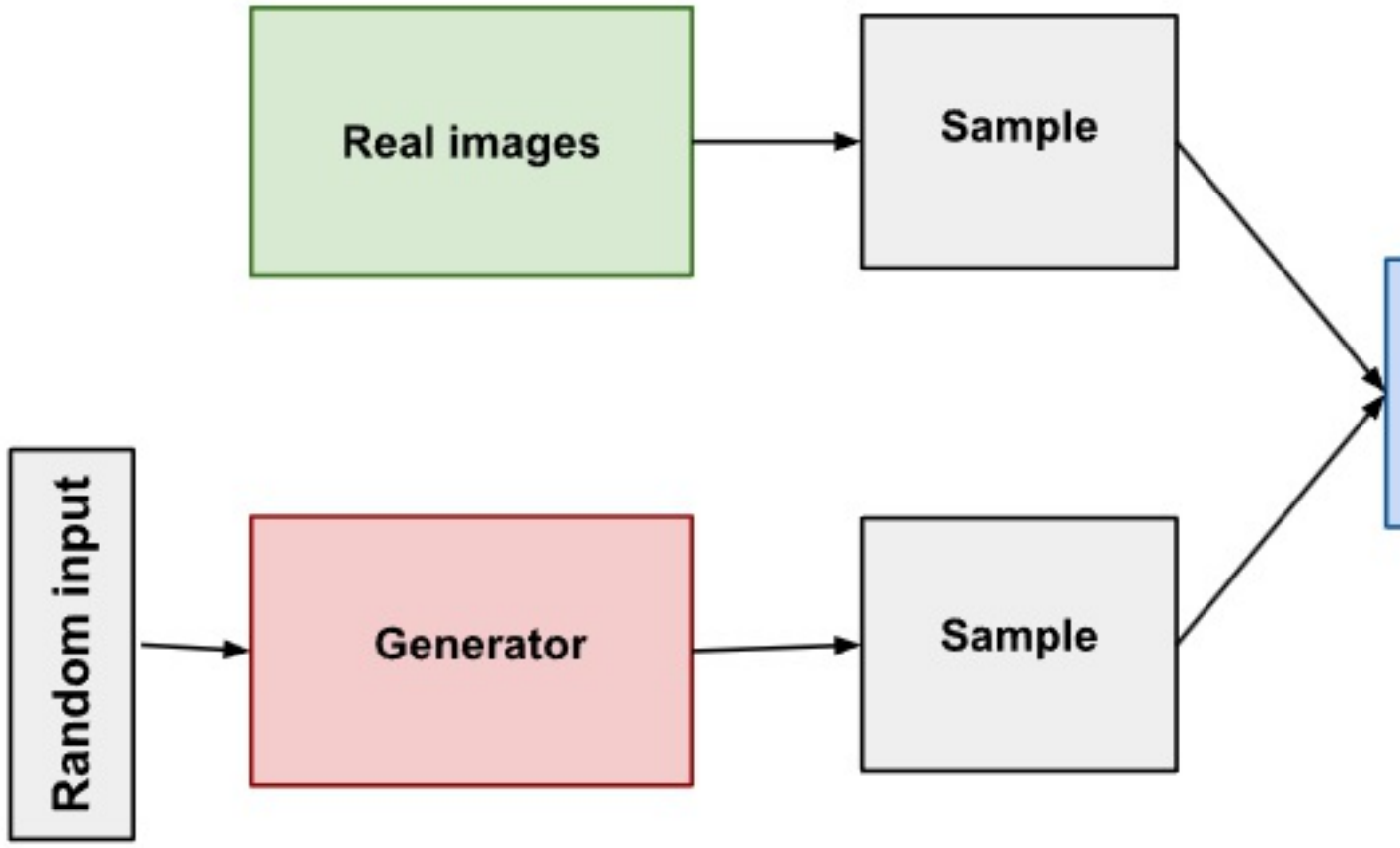
As training progresses, the generator gets closer to producing



Finally, if generator training goes well, the discriminator gets wo
and fake. It starts to classify fake data as real, and its accuracy

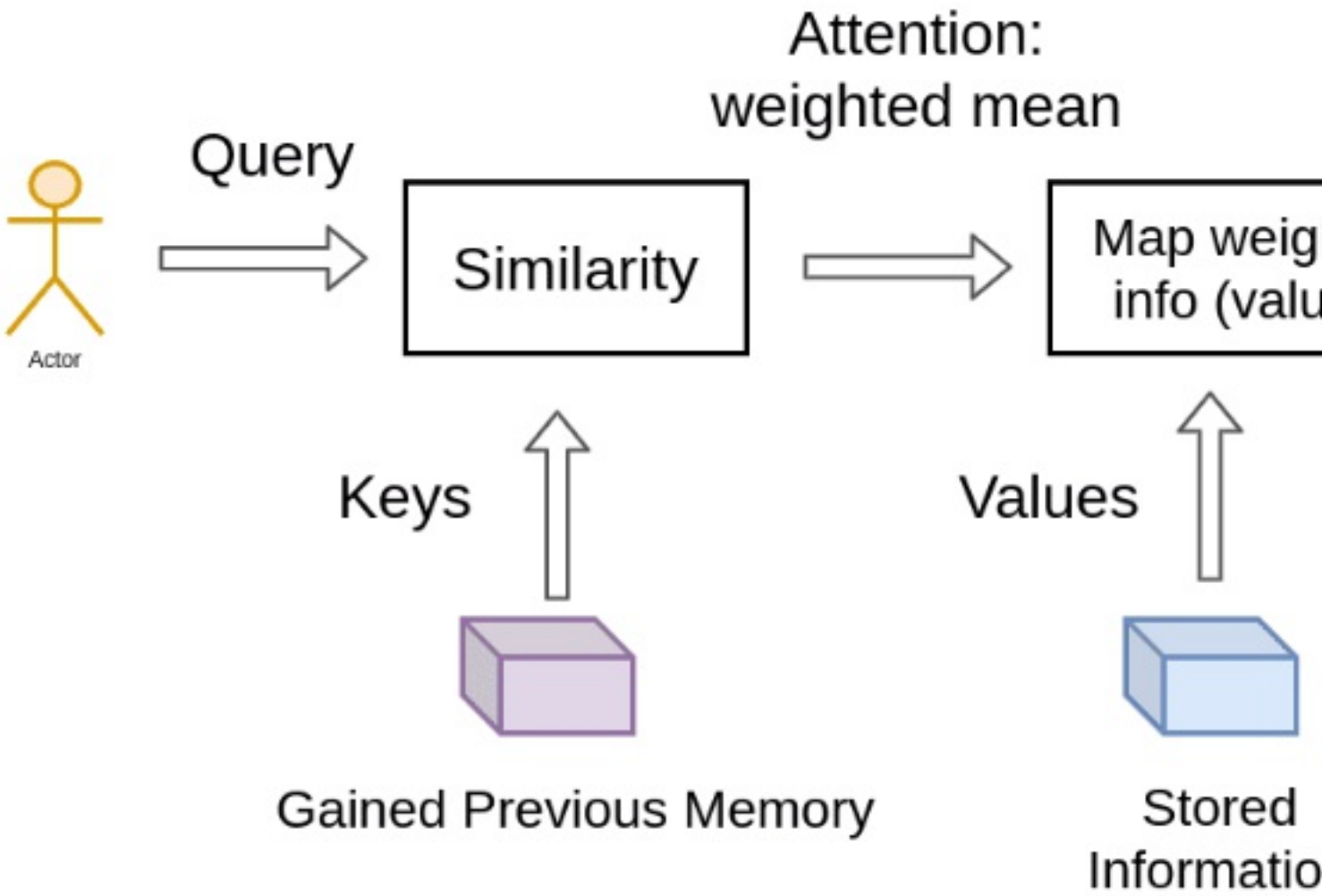
REAL REAL

Here's a picture of the whole system:



Through training , the generator improves at creating realistic samples, and the discriminator become better at distinguishing between real and fake samples and it can be used for image generation , data augmentation .

1. **Transformers**



It is also a type of neural network architecture introduced in 2017 by Vaswani et al. in the paper "Attention is All You Need". They revolutionized the field of Natural Language Processing (NLP) and have since been widely adopted in many areas of machine learning.

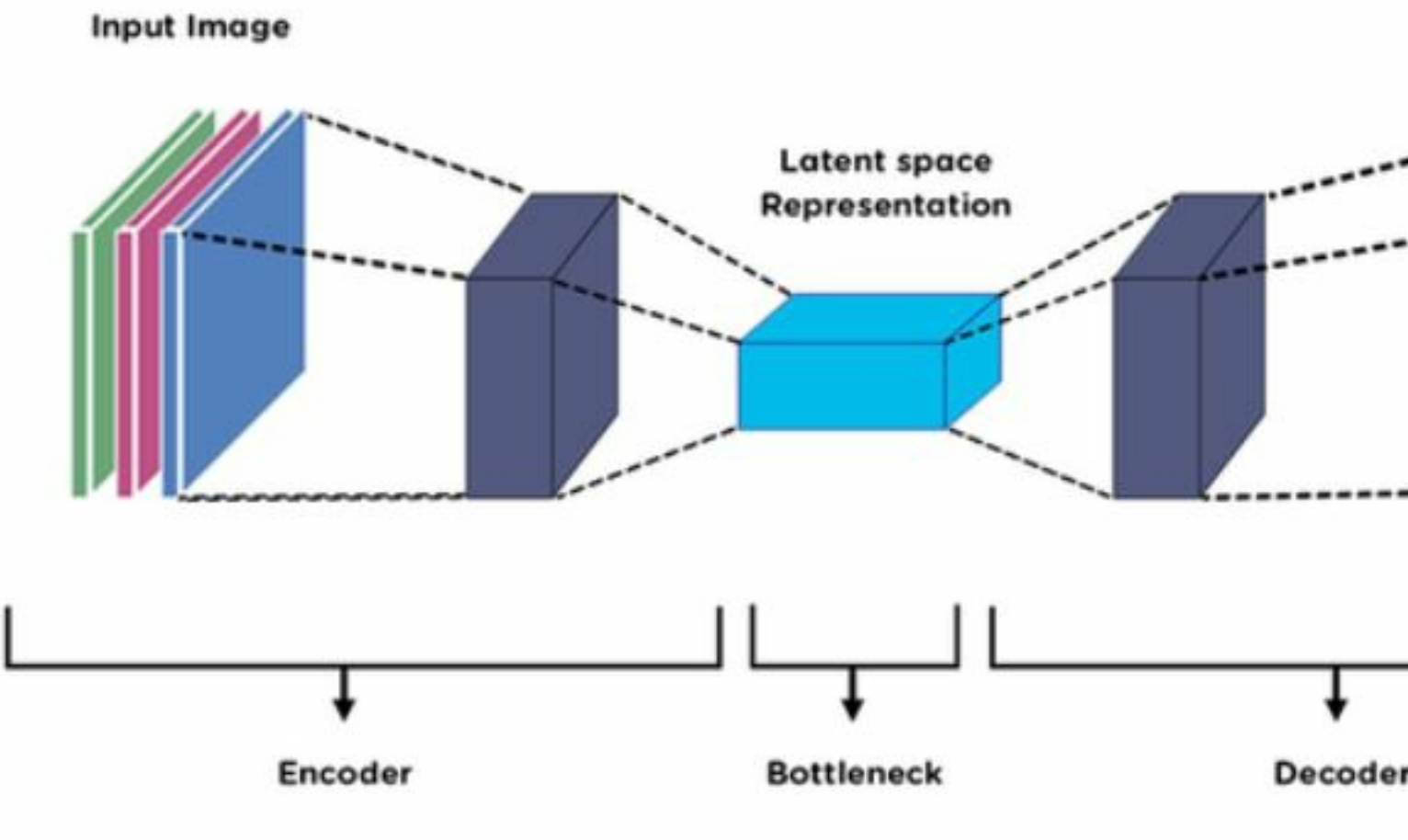
Features of transformers :

1. **Parallelization** : they can be parallelized more easily than recurrent neural networks (RNNs), making them faster to train and deploy.
2. **Scalability** : they can handle longer input sequences and larger models than RNNs.
3. **Performance** : they have achieved state-of-the-art results in many NLP tasks, such as machine translation, text classification, and language modeling.

These transformers can be used in various applications

like Machine translation , language modelling , text classification , chatbots and conversational AI .

1. **Encoders :**



In machine learning, an encoder is a component of a neural network architecture that processes input data and transforms it into a lower-dimensional representation, called a latent representation or encoding. The encoder is typically used in conjunction with a decoder, which generates the output data from the latent representation.

It is needed for tasks like dimensionality reduction , anamoly detection , generative modelling , language modelling .

It often used as part of a large model , such as transformer or autoencoder.

1. **Large Language Models (LLMs):**

Local Linear Models (LLMs) are a type of machine learning model that combines the simplicity of linear models with the flexibility of non-linear models. It can assume that the relationship between the input features and the target variable is locally linear, meaning that the relationship can be approximated by a linear function within a small region of the input space.

It is used in many applications like regression tasks , classification tasks , time series forecasting and recommendation systems .

Here are some popular algorithms that are used in LLMs:

1. Local Linear Regression (LLR)
2. Piecewise Linear Regression(PLR)

3. Generalized Additive Models (GAMs)
4. Tree -based Models (e.g., CART , Random Forest)

ANS 2: Super-resolution using GANs

Super-resolution using GANs aims to enhance the resolution of images, making them clearer and more detailed. GANs can be used to generate high-resolution images from low-resolution inputs by learning the underlying distribution of high-resolution images, in the medical science as we now that sometime the input we want is not clear due to which is not possible give to accurate results that's why its better is use this neural network with super resolution .

Generative Adversarial Network (GAN): it's a type of neural network which Consists of two main components, the Generator and the Discriminator.

- **Generator**: it creates high-resolution images from low-resolution inputs.
- **Discriminator**: it evaluates the quality of the generated images, distinguishing between real high-resolution images and generated ones.

Implementation Steps

1. Data Preparation

- **Collecting High-Resolution and Low-Resolution Image Pairs**:
- First we gather a dataset of high-resolution images and down sample them to create corresponding low-resolution images. Remember the dataset is large and diverse for better performance.

2. Network Architecture

- **Encoder Network**: Use a convolutional neural network (CNN) to extract features from the low-resolution images. This network will compress the input image into a latent space representation.
- **Generator Network**: Design a deep network (e.g., U-Net or ResNet) to upsample the feature representation from the encoder and reconstruct the high-resolution image. Incorporate techniques like residual connections and dilated convolutions to improve quality.
- **Discriminator Network**: Implement a CNN-based network that classifies images as either real (from the high-resolution dataset) or fake (generated by the Generator). This helps the Generator improve its output quality over time

3. Training

- **Loss Functions**: it use a combination of loss functions:
- **Adversarial Loss**: it measures how well the Generator can fool the Discriminator.
- **Content Loss**: it ensure the generated image is similar to the ground truth high-resolution image. Often implemented as Mean Squared Error (MSE) or L1 loss.
- **Perceptual Loss**: it measures the difference between high-level features of the generated and ground truth images, often using pre-trained networks

like VGG.
- **Training Procedure**:
- There is alternate between training the Generator and the Discriminator.
- It uses a batch of low-resolution images to generate high-resolution images and compare them to the actual high-resolution images.
- Optimize the networks using gradient descent methods like Adam or RMSprop.

4. Evaluation

- **Quantitative Metrics**: it evaluate the performance using metrics such as Peak Signal-to-Noise Ratio (PSNR) and Structural Similarity Index (SSIM) to assess image quality.
- **Qualitative Assessment**: Inspect the generated images to ensure they are visually appealing and free from artifacts.

5. Deployment

- **Medical Field**: In medical imaging, implement the trained GAN model to enhance the resolution of diagnostic images such as MRI or CT scans. This can help in detecting finer details and improving diagnostic accuracy.
- **Camera Surveillance**: it apply the GAN model to enhance surveillance footage, making it easier to identify objects or individuals in low-resolution video feeds.

Ans 3 . Research problems involving Generative Adversarial Networks (GANs) for image super-resolution are broad and multifaceted. Below are some significant research problems that address current limitations and explore new frontiers in this area:

1. Improving Image Quality and Realism

Problem: GANs is used for generating high-resolution images with artifacts or unnatural textures, which can lower the perceived quality and orginality of the images.

- **Research Questions**:
  - How can the architecture of GANs be improved to minimize artifacts in super-resolved images?
  - What novel loss functions or regularization techniques can be introduced to enhance the visual quality and fidelity of the generated images?
- **Potential Approaches**:
  - Explore advanced GAN architectures, such as those incorporating self-attention mechanisms or novel residual connections.
  - Develop new perceptual or adversarial loss functions that better capture high-level visual features.

2. Data Requirements and Generalization

Problem: Training GANs for super-resolution often requires large amounts of high-quality paired data (high-resolution and low-resolution image pairs). In

practice, such datasets may not be readily available.

- **Research Questions**:
  - How can GANs be trained effectively with limited or unpaired datasets?
  - Can techniques like transfer learning or domain adaptation improve performance when high-resolution data is scarce?
- **Potential Approaches**:
  - Investigate methods for semi-supervised or unsupervised learning using unpaired datasets.

Implementation of Encoders for super-resolution GANs (SRGANs) in the medical field and camera surveillance

Autoencoders are neural network-based data compression algorithms that are data-specific lossy and learn from examples. They consist of three key components:

1. **Encoding Function:** Compresses input data into a lower-dimensional representation.
2. **Decoding Function:** Reconstructs the original data from the compressed representation.
3. **Loss Function:** Measures the difference between the original and reconstructed data to minimize information loss during training.

Autoencoders are tailored to specific data types (e.g., faces) and may not generalise well to unrelated data (e.g., trees). They are lossy, meaning the decompressed data is slightly degraded. However, they are easy to train for specialised tasks with the right data.

- **Encoder**: The encoder is responsible for extracting key features from the low-resolution image. It typically consists of several convolutional layers that progressively reduce the spatial dimensions while increasing the depth of feature maps.
- **Decoder**: The decoder reconstructs the high-resolution image from the compressed feature representation. It uses techniques like deconvolution (transposed convolution) to increase the spatial dimensions back to the original resolution.

Autoencoders are great for:

1. **Data Denoising:** They clean up noisy data, making it clearer.
2. **Dimensionality Reduction:** They shrink data to fewer dimensions, which helps in visualizing it better. Autoencoders can find patterns that basic methods like PCA might miss.

1. Problem Definition

- **Medical Field**: Enhance the resolution of medical images like MRI, CT scans, or X-rays to improve diagnostic accuracy by revealing finer details that may not be visible in low-resolution images.
- **Camera Surveillance**: Upscale low-resolution footage to better identify

individuals, objects, or events, aiding in security and forensic investigations

2. Data Collection and Preprocessing

- **Medical Field**:
  - Collect a dataset of high-resolution medical images and their corresponding low-resolution counterparts. This can involve downsampling high-resolution images to create training pairs.
- **Camera Surveillance**:
  - Gather surveillance footage or images in both high and low resolution. Downsample high-resolution frames to create pairs.

3. Encoder-Decoder Architecture

- **Encoder**: The encoder is responsible for extracting key features from the low-resolution image. It typically consists of several convolutional layers that progressively reduce the spatial dimensions while increasing the depth of feature maps.
- **Decoder**: The decoder reconstructs the high-resolution image from the compressed feature representation. It uses techniques like deconvolution (transposed convolution) to increase the spatial dimensions back to the original resolution.

4. GAN Integration

- **Generator**: The encoder-decoder acts as the generator in the GAN framework, attempting to generate high-resolution images from low-resolution inputs.
- **Discriminator**: The discriminator is trained to differentiate between real high-resolution images and the generated ones.
- **Loss Functions**:
  - **Content Loss**: Usually based on the difference between the generated high-resolution image and the ground truth, measured using pixel-wise loss (e.g., MSE or L1 loss).

5. Training Strategy

- **Dataset Splitting**: Divide the dataset into training, validation, and test sets. Ensure that the model is not overfitting by using a diverse set of images.
- **Training Process**:
  - Alternate between training the generator and the discriminator.
  - Use a learning rate scheduler and early stopping to prevent overfitting.
  - Fine-tune the model by using techniques like transfer learning, especially in the medical field where annotated data is often scarce.

6. Post-processing

- **Image Enhancement**: After generating the high-resolution images, apply further enhancement techniques like sharpening, denoising, and contrast adjustment to improve visual quality.

7. Evaluation Metrics

- **PSNR (Peak Signal-to-Noise Ratio)**: Measures the quality of the reconstructed image compared to the original high-resolution image.
- **SSIM (Structural Similarity Index)**: Assesses the similarity between the generated and real images, focusing on structural information.
- **Qualitative Assessment**: In medical applications, have experts (e.g., radiologists) evaluate the quality and diagnostic utility of the enhanced images.

8. Deployment

- **Medical Field**: Integrate the model into medical imaging software, ensuring that it can process images in real-time or near-real-time.
- **Camera Surveillance**: Deploy the model in surveillance systems, where it can enhance images on the fly or process stored footage for analysis.

.

Implementation

```
#importing libraries
!pip install tensorflow
import tensorflow as tf
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import os
```

Show hidden output

```
[ ] #dataset loading
    def load_data(path):
        with np.load(path) as f:
            x_train, y_train = f['x_train'], f['y_train']
            x_test, y_test = f['x_test'], f['y_test']
            return (x_train, y_train), (x_test, y_test)
```

```
ENCODING_DIM = 32

# input placeholder
input_img = tf.keras.layers.Input(shape=(784,))

# this is the encoded representation of the input
encoded = tf.keras.layers.Dense(ENCODING_DIM, activatio

# this is the loss reconstruction of the input
decoded = tf.keras.layers.Dense(784, activation='sigmoi

# this model maps an input to its recommendation
autoencoder = tf.keras.models.Model(input_img, decoded)
```

```
[ ] encoder = tf.keras.models.Model(input_img, encoded)
```

```
[ ] # as well as decoder model
    # create a placeholder for an encoded (32-dimensional)
    encoded_input = tf.keras.layers.Input(shape=(ENCODING_D

    # retrieve the last layer of the autoencoder model
    decoder_layer = autoencoder.layers[-1]

    # create the decoder model
    decoder = tf.keras.models.Model(encoded_input, decoder_
```

```
# Now let's train our autoencoder to reconstruct MNIST
# first we will configure our model to use a per-pixel
autoencoder.compile(optimizer='adadelta', loss='binary
```

```
# let's prepare our input data. We are using MNIST dig
# load the data
(x_train, _), (x_test, _) = tf.keras.datasets.mnist.lo
```

```
Downloading data from https://storage.googleapis.com/t
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

```
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255
x_train = x_train.reshape((len(x_train), np.prod(x_trai
x_test = x_test.reshape((len(x_test), np.prod(x_test.sh
print(x_train.shape)
print(x_test.shape)
```

```
(60000, 784)
(10000, 784)
```

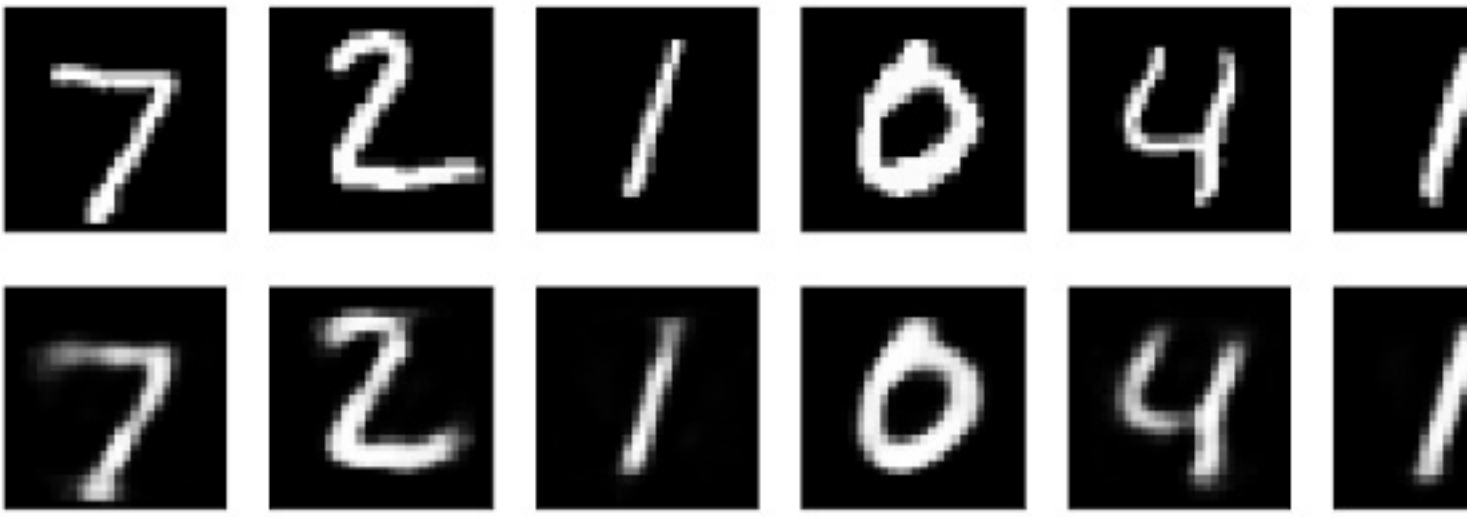
```
# now let's train our autoencoder for 50 epochs
autoencoder.fit(x_train, x_train, epochs=50, batch_size
```

Show hidden output

```
# after 50 epochs the autoencoder seems to reach a stab
# encode and decode some digits
# note that we take them from the "test" set
encoded_imgs = encoder.predict(x_test)
decoded_imgs = decoder.predict(encoded_imgs)
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
```

```
# now using Matplotlib to plot the images
n = 10 # how many images we will display
plt.figure(figsize=(20, 4))
for i in range(n):
    # display original
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # display reconstruction
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

plt.show()
```

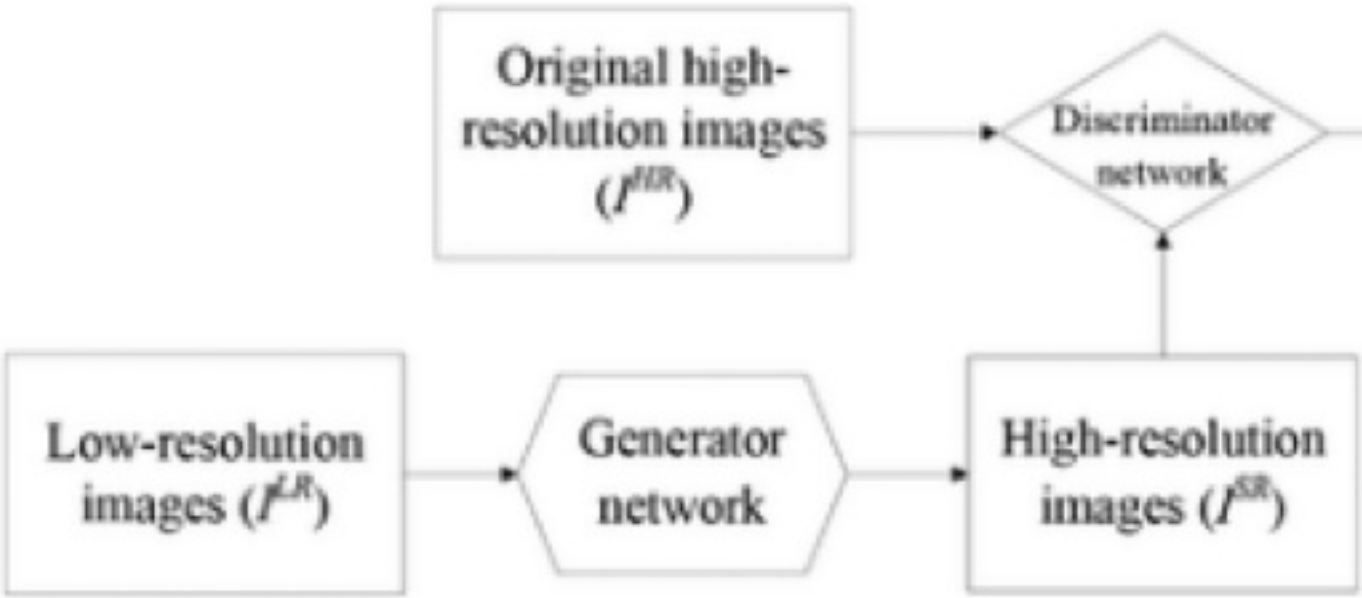


```
ENCODING_DIM = 32

input_img = tf.keras.layers.Input(shape=(784,))

# add a dense layer with L1 activity regularizer
encoded = tf.keras.layers.Dense(ENCODING_DIM, activation='re
decoded = tf.keras.layers.Dense(784, activation='sigmoid')(e
autoencoder = tf.keras.models.Model(input_img, decoded)
autoencoder.compile(optimizer='adadelta', loss='binary_cross

# now let's train this for 100 epochs (with added regulariza
autoencoder.fit(x_train, x_train, epochs=100, batch_size=256
```

Q. Why we use GAN for super resolution of images?

A. Generative Adversarial Networks (GANs) are used for super-resolution of images because they excel at generating high-quality, realistic images, which makes them well-suited for enhancing the resolution of low-resolution images.

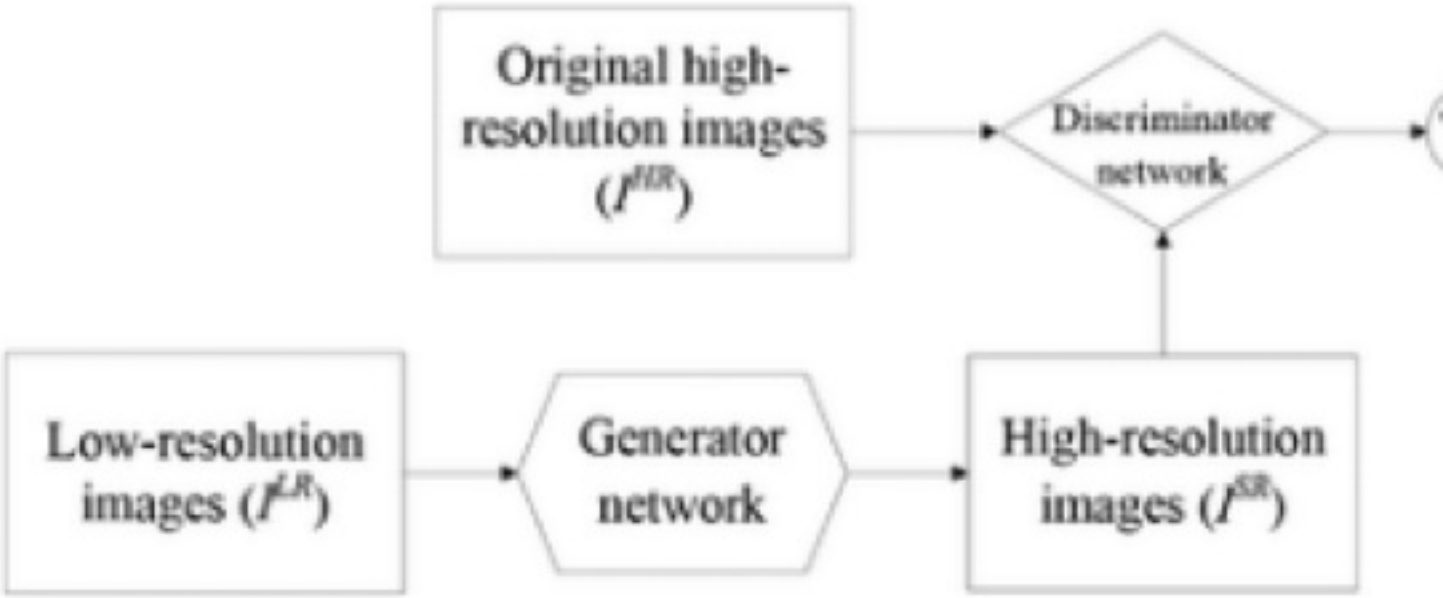
High-Quality Image Generation: GANs consist of two networks, a generator and a discriminator. The generator creates high-resolution images from low-resolution inputs, while the discriminator evaluates whether these images are realistic or not. This adversarial process helps the generator produce images that look more natural and detailed.



Perceptual Loss: In traditional methods, super-resolution focuses on minimizing pixel-wise differences (e.g., mean squared error) between the high-resolution and

low-resolution images, which often leads to blurry results. GANs, however, can use a perceptual loss function that focuses on high-level features (e.g., edges, textures) extracted from a pre-trained network, leading to sharper and more visually pleasing images.

Perpetual loss function (LSR), which is used by the SRGAN, is the weighted sum of two types of loss: content loss and adversarial loss. For the generator architecture's performance, this loss is crucial.



Learning Fine Details: GANs are particularly good at learning and recreating fine textures and details in images, which are crucial for high-quality super-resolution. This makes the output images look more realistic compared to outputs from other methods.

Adaptive Upscaling: GANs can learn complex mappings from low-resolution to high-resolution images, adapting to different image types and content. This adaptability allows GANs to produce better results across a wide range of images, including faces, landscapes, and text.

SRGAN

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import sys, os

from keras.layers import Input, Dense
from keras.models import Model
from keras.optimizers import SGD, Ada
```

```
# Load the MNIST Data

mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test)

# Scale the inputs in range of (-1, +
x_train, x_test = x_train / 255.0 * 2
```

```
print ("x_train.shape:" ,x_train.sha
print ("x_test.shape:" ,x_test.shape
```

```
x_train.shape: (60000, 28, 28)
x_test.shape: (10000, 28, 28)
```

```
# Flatten the data
N, H, W = x_train.shape
D = H * W
x_train = x_train.reshape(-1, D)
x_test = x_test.reshape(-1, D)
```

```
print ("x_train.shape:" ,x_train.sha
print ("x_test.shape:" ,x_test.shape
```

```
x_train.shape: (60000, 784)
x_test.shape: (10000, 784)
```

```
# Dimensionality of the latent space
latent_dim = 100
```

```
# Defining the generator model
def build_generator(latent_dim):
  i = Input(shape=(latent_dim,))
  x = Dense(256, activation=LeakyReL
  x = BatchNormalization(momentum=0.
  x = Dense(512, activation=LeakyReL
  x = BatchNormalization(momentum=0.
  x = Dense(1024, activation=LeakyRe
  x = BatchNormalization(momentum=0.
  x = Dense(D, activation='tanh')(x)

  model = Model(i, x)
  return model
```

```
# Defining the discriminator model
def build_discriminator(img_size):
  i = Input(shape=(img_size,))
  x = Dense(512, activation=LeakyReL
  x = Dense(256, activation=LeakyReL
  x = Dense(1, activation='sigmoid')

  model = Model(i, x)
  return model
```

```
# Compile both models in preparation

# Build and compile the discriminator
discriminator = build_discriminator(D
discriminator.compile ( loss='binary_

# Build and compile the combined mode
generator = build_generator(latent_di
```

```
generator.summary()
```

```
Model: "functional_1"
```

Layer (type)
input_layer_1 (InputLayer)
dense_3 (Dense)
batch_normalization (BatchNormalization)
dense_4 (Dense)
batch_normalization_1 (BatchNormalization)
dense_5 (Dense)
batch_normalization_2 (BatchNormalization)
dense_6 (Dense)

Total params: 1,493,520 (5.70 MB)

Trainable params: 1,489,936 (5.68

Non-trainable params: 3,584 (14.00

Non-trainable params: 3,584 (14.00

```
discriminator.summary()
```

Model: "functional"

Layer (type)
input_layer (InputLayer)
dense (Dense)
dense_1 (Dense)
dense_2 (Dense)

Total params: 533,505 (2.04 MB)

Trainable params: 533,505 (2.04 MB

Non-trainable params: 0 (0.00 B)

```
# Create an input to represent noise
z = Input(shape=(latent_dim,))
z.shape
```

```
(None, 100)
```

```
# Pass noise through generator to get
img = generator(z)
```

```
img
```

```
<KerasTensor shape=(None, 784), dtyp
```

```
# Make sure only the generator is tra
discriminator.trainable = False
```

```
# The true output is fake, but we lab
# Passing the output of Generator to

fake_pred = discriminator(img)
```

```
# Create the combined model object
combined_model_gen = Model(z, fake_p
```

```
# Compile the combined model
combined_model_gen.compile(loss='bin
```

```
# Config

batch_size = 32
epochs = 10000
sample_period = 200 # every `sample_
```

```
# Create batch labels to use when ca
ones = np.ones(batch_size)
zeros = np.zeros(batch_size)

# Store the losses
d_losses = []
g_losses = []

# Create a folder to store generated
if not os.path.exists('gan_images'):
  os.makedirs('gan_images')
```

```
# A function to generate a grid of ra

def sample_images(epoch):
    rows, cols = 5, 5
    noise = np.random.randn(rows * cols
    imgs = generator.predict(noise)

    # Rescale images 0 - 1
    imgs = 0.5 * imgs + 0.5

    fig, axs = plt.subplots(rows, cols)
    idx = 0
    for i in range(rows):
        for j in range(cols):
            axs[i,j].imshow(imgs[idx].resha
            axs[i,j].axis('off')
            idx += 1
    fig.savefig("gan_images/%d.png" % e
    plt.close()
```

```
### Train the GAN
```

```
# Main training loop
for epoch in range(epochs):
    ##############################
    ### Train discriminator ###
    ##############################

    # Select a random batch of images
    idx = np.random.randint(0, x_train.s
    real_imgs = x_train[idx]

    # Generate fake images
    noise = np.random.randn(batch_size,
    fake_imgs = generator.predict(noise)

    # Train the discriminator
    # both loss and accuracy are returne
    d_loss_real, d_acc_real = discrimina
    d_loss_fake, d_acc_fake = discrimina
    d_loss = 0.5 * (d_loss_real + d_loss
    d_acc  = 0.5 * (d_acc_real + d_acc_f


    ##########################
    ### Train generator ###
    ##########################
```

```
    noise = np.random.randn(batch_size,
    g_loss = combined_model_gen.train_on

    # Repeat the training for generator
    noise = np.random.randn(batch_size,
    g_loss = combined_model_gen.train_on
```

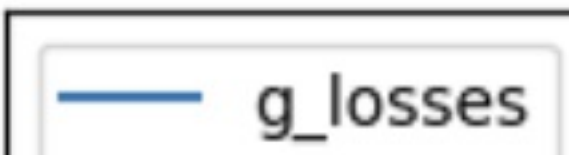
```
        # Save the losses
        d_losses.append(d_loss)
        g_losses.append(g_loss)

        # Ensure d_loss, d_acc, and g
        if isinstance(d_loss, list):
            d_loss = d_loss[0]
        if isinstance(d_acc, list):
            d_acc = d_acc[0]
        if isinstance(g_loss, list):
            g_loss = g_loss[0]

        if epoch % 100 == 0:
            print(f"epoch: {epoch+1}/

        if epoch % sample_period == 0
            sample_images(epoch)
print("ok")
```

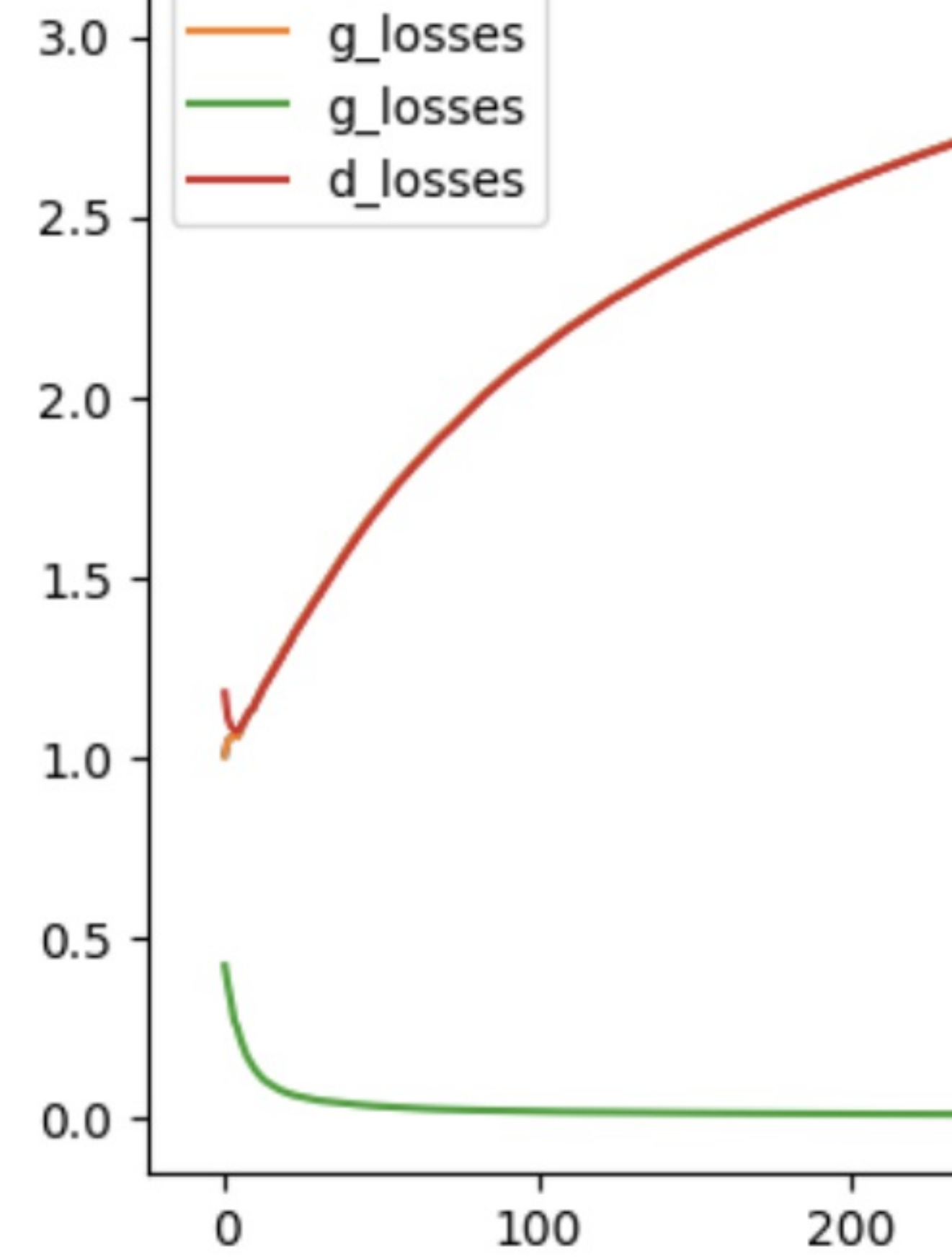
```
print('ok')
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 21ms/
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/
epoch: 1801/2200, d_loss: 4.36, g
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms
```

[58]:
```
plt.plot(g_losses, label='g_losses
plt.plot(d_losses, label='d_losses
plt.legend()
```

[58]:
```
<matplotlib.legend.Legend at 0x44d
```




g_losses
g_losses
d_losses
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
100
200

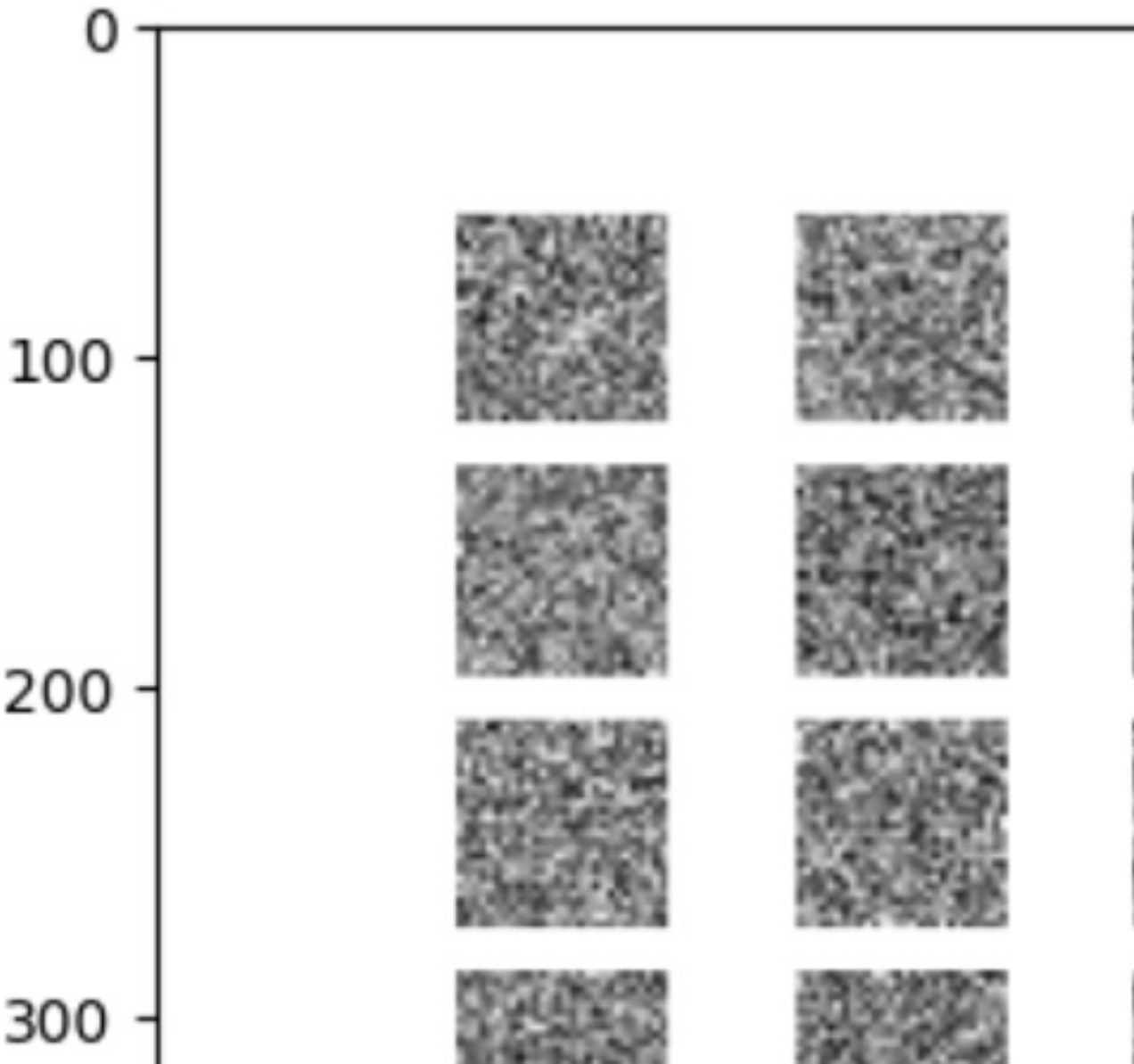
```
]: !ls gan_images
```

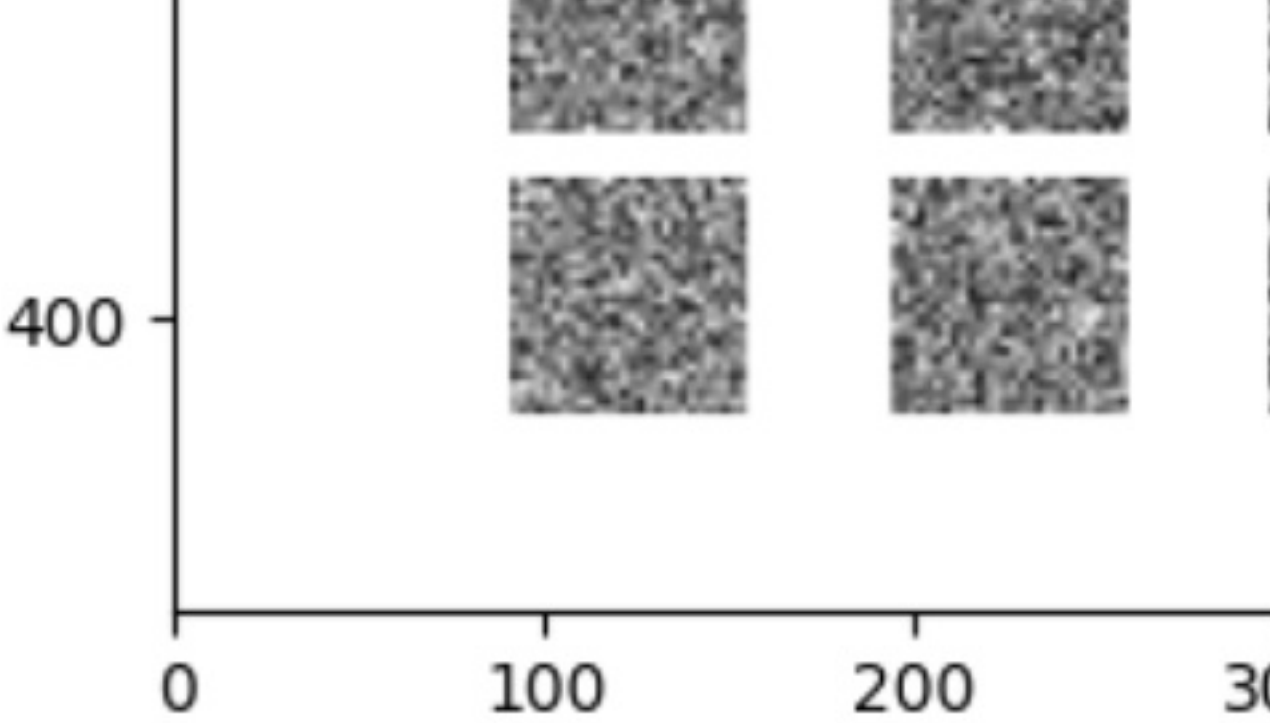
```
0.png    1200.png 1600.png 200.png  220
1000.png 1400.png 1800.png 2000.png 240
```

1000.png 1400.png 1800.png 2000.pn

```
[62]: from skimage.io import imread
      a = imread('gan_images/0.png')
      plt.imshow(a)
```

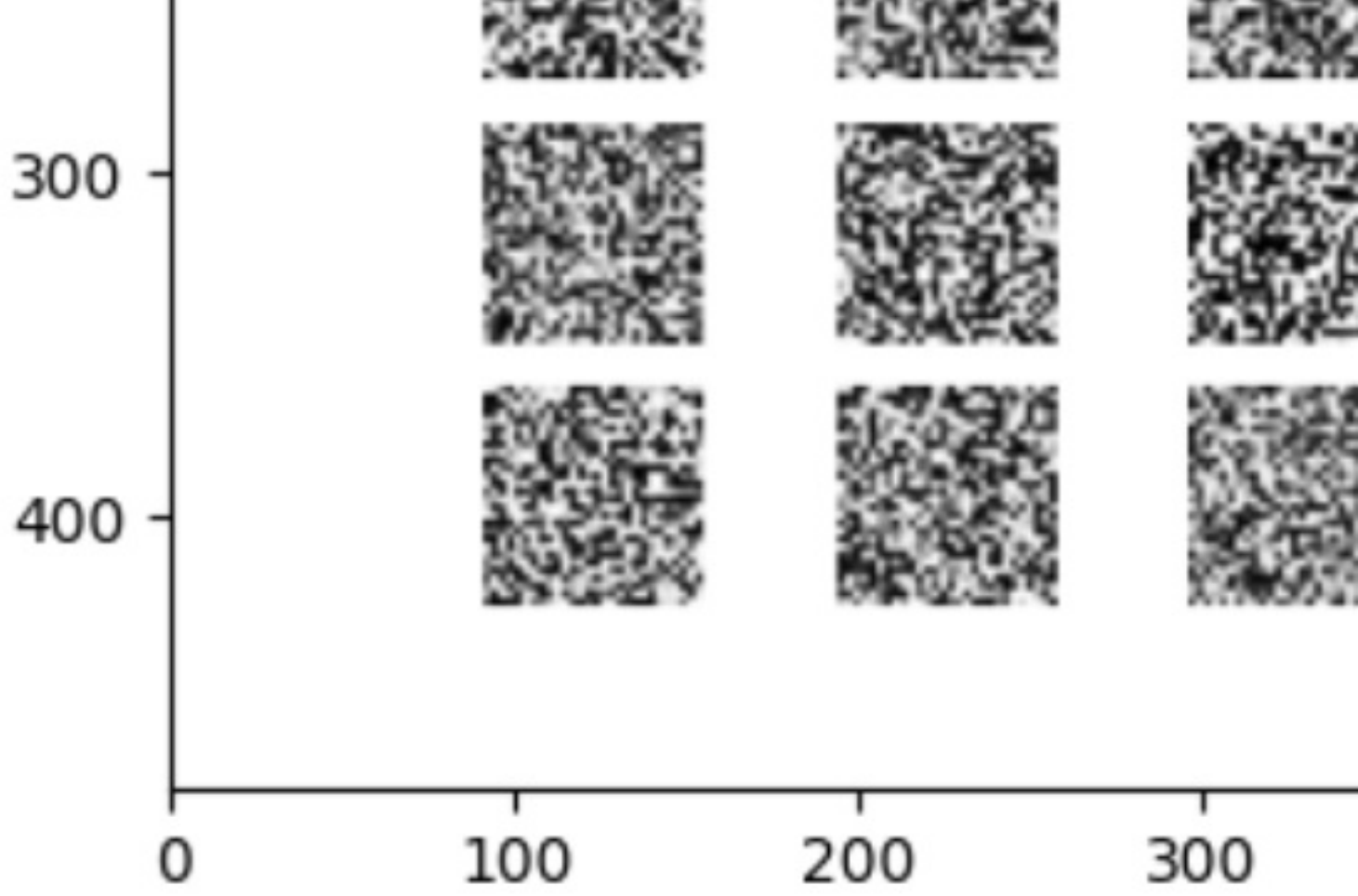
```
[62]: <matplotlib.image.AxesImage at 0x4
```





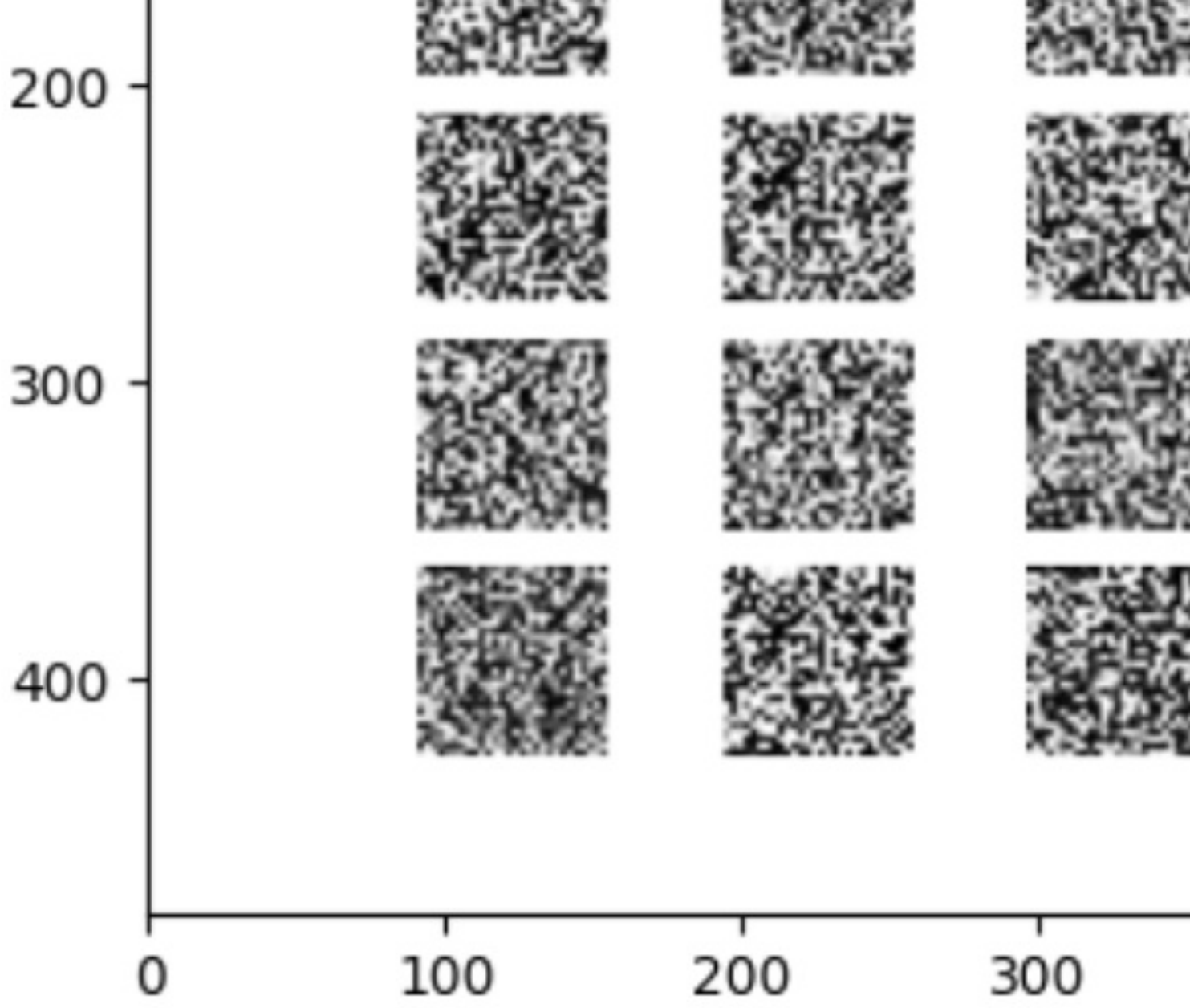

```
a = imread('gan_images/200.png')
plt.imshow(a)
```

```
<matplotlib.image.AxesImage at 0x457b12
```




```
a = imread('gan_images/400.png')
plt.imshow(a)
```

```
<matplotlib.image.AxesImage at 0x457e148
```


200
300
400
0
100
200
300