

UNIT-V GENERATIVE DEEP LEARNING

Generative deep learning: Text generation – Deep dream – Neural style transfer – Generating images with variational auto encoders – Introduction to Generative Adversarial Networks

5.1 Generative Deep Learning

Introduction

Generative Deep Learning refers to a class of machine learning techniques that enable computers to generate new data that resembles the training data. Unlike discriminative models that focus on classification or regression tasks, generative models learn the underlying distribution of data and can generate new samples from this distribution.

Key Concepts

1. **Generative Models:** These models estimate the joint probability distribution $P(x,y)P(x,y)P(x,y)$ or the data distribution $P(x)P(x)P(x)$ itself.
2. **Discriminative vs Generative:** Discriminative models learn the boundary between classes ($P(y|x)P(y|x)P(y|x)$), whereas generative models learn how data is generated ($P(x|y)P(x|y)P(x|y)$ or $P(x)P(x)P(x)$).
3. **Latent Space:** Many generative models operate in a latent space, where they map random noise to data points (e.g., images).

Popular Generative Deep Learning Models

1. Generative Adversarial Networks (GANs)

- **Introduced by:** Ian Good fellow (2014)
- **Architecture:** Consists of two neural networks: Generator (G) and Discriminator (D) in a min-max game.
- **Working:**
 - The generator tries to produce realistic data from random noise.
 - The discriminator tries to distinguish between real and generated data.
 - They train adversarially until the generator produces data indistinguishable from real data.

2. Variational Autoencoders (VAEs)

- **Type:** Probabilistic generative model
- **Components:** Encoder, Decoder, and a probabilistic latent space.

- **Working:** Encodes input data into a distribution over latent space, then samples from it to reconstruct data. Uses a loss function combining reconstruction error and KL divergence.

3. Autoregressive Models

- Examples: PixelRNN, PixelCNN, GPT (for text)
- **Working:** Model data generation as a sequential process where each part of the data is conditioned on previous parts.

4. Diffusion Models

- **Used for:** High-quality image generation
- **Working:** Gradually adds noise to data (forward process), then learns to reverse the noise to generate samples.
- **Example:** DALL·E 2, Imagen

Applications

- **Image Generation:** Creating realistic faces, art (e.g., DALL·E, Midjourney)
- **Text Generation:** Language models like GPT
- **Music and Audio Synthesis**
- **Style Transfer and Image-to-Image Translation:** E.g., turning sketches into photos
- **Data Augmentation:** For training machine learning models

Challenges

- **Training Instability:** Especially in GANs
- **Mode Collapse:** Generator produces limited variety
- **Evaluation Metrics:** Hard to objectively measure quality (FID, IS scores used)
- **Ethical Concerns:** Deepfakes, fake news, content authenticity

What is Text Generation?

Text generation is a process where AI produces text that resembles natural human communication. Text generation is a process where an AI system produces written content, imitating human language patterns and styles. The process involves generating coherent and meaningful text that resembles natural human communication. Text generation has gained significant importance in various fields, including natural language processing, content creation, customer service, and coding assistance.

Text Generation Explained

Text generation works by utilizing algorithms and language models to process input data and generate output text. It involves training AI models on large datasets of text to learn patterns, grammar, and contextual information. These models then use this learned knowledge to generate new text based on given prompts or conditions.

At the core of text generation are language models, such as GPT (Generative Pre-trained Transformer) and Google's PaLM, which have been trained on vast amounts of text data from the internet. These models employ deep learning techniques, specifically neural networks, to understand the structure of sentences and generate coherent and contextually relevant text.

During the text generation process, the AI model takes a seed input, such as a sentence or a keyword, and uses its learned knowledge to predict the most probable next words or phrases. The model continues to generate text, incorporating context and coherence, until a desired length or condition is met.

Examples of Real-World Text Generation Applications

Text generation finds application in various real-world scenarios, such as:

- **Content creation.** AI-powered systems can generate articles, blog posts, and product descriptions. These systems are trained on vast amounts of data and can produce coherent content in a fraction of the time it would take a human writer.
- **Chatbots and virtual assistants.** AI-powered chatbots and virtual assistants use text generation to interact with users in a conversational manner. They can understand user queries and provide relevant responses, offering personalized assistance and information.
- **Language translation.** Text generation models can be utilized to improve language translation services. By analyzing large volumes of translated text, AI models can generate accurate translations in real time, enhancing communication across different languages.
- **Summarization.** Text summarization offers a concise version of information by identifying the most important points, which can help generate summaries of research papers, blog posts, news articles, chapters, and books.

What are the Benefits of Text Generation?

Text generation offers several advantages:

- **Increased efficiency.** AI-powered text generation can automate content creation, reducing the time and effort required for manual writing. This can enhance productivity and allows users to generate large volumes of content at scale.

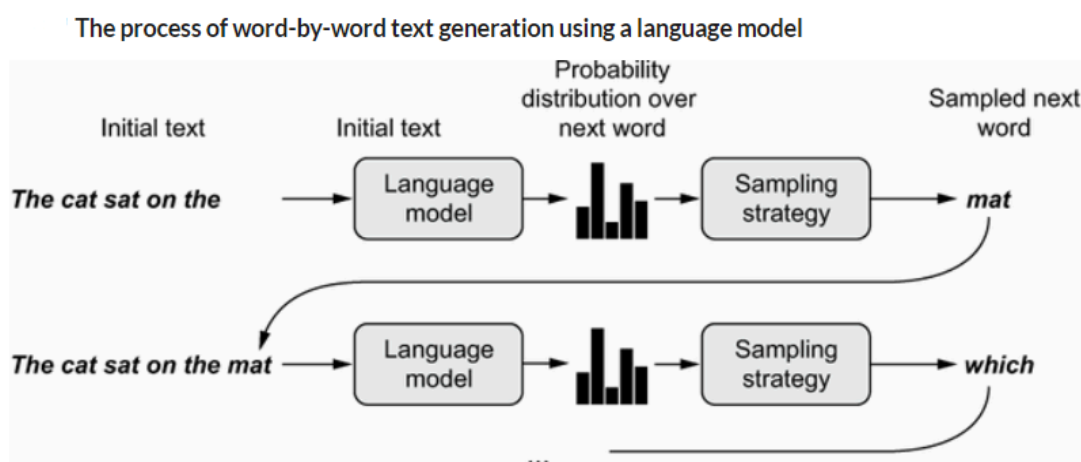
- **Improved personalization.** Text generation models can be fine-tuned to generate personalized content based on user preferences and historical data. This enables tailored recommendations, personalized marketing messages, and customized responses in customer service interactions.
- **Language accessibility.** Text generation enables translation services and speech synthesis, making information accessible to individuals who may have difficulties reading or understanding written text. It opens up new possibilities for inclusive communication and enhances accessibility.

Top Performing Text Generation Models

1. **GPT-4.** OpenAI's (and the world's) most advanced system, which generates responses that are both safe and useful.
2. **Claude.** A next-generation AI assistant developed by Anthropic, designed to be helpful, honest, and harmless.
3. **ChatGPT.** This model is a lot like InstructGPT, but it's trained to follow prompts and provide detailed responses.

How do you generate sequence data?

The universal way to generate sequence data in deep learning is to train a model (usually a Transformer or an RNN) to predict the next token or next few tokens in a sequence, using the previous tokens as input. When working with text data, tokens are typically words or characters, and any network that can model the probability of the next token given the previous ones is called a language model. A language model captures the latent space of language: its statistical structure.



Once you have such a trained language model, you can sample from it (generate new sequences): you feed it an initial string of text (called conditioning data), ask it to generate the next character or the next word (you can even generate several tokens at once), add the

generated output back to the input data, and repeat the process many times (see figure 12.1). This loop allows you to generate sequences of arbitrary length that reflect the structure of the data on which the model was trained: sequences that look almost like human-written sentences.

The importance of the sampling strategy

When generating text, choosing the next token is crucially important. A **naive** approach is greedy sampling, consisting of always choosing the most likely next character.

Disads: But such an approach results in repetitive, predictable strings that don't look like coherent language.

Stochastic sampling: it introduces randomness in the sampling process by sampling from the probability distribution for the next character.

In such a setup, if a word has probability 0.3 of being next in the sentence according to the model, you'll choose it 30% of the time. Note that greedy sampling can also be cast as sampling from a probability distribution: one where a certain word has probability 1 and all others have probability 0.

Sampling probabilistically from the softmax output of the model is neat: it allows even unlikely words to be sampled some of the time, generating more interesting-looking sentences and sometimes showing creativity by coming up with new, realistic-sounding sentences that didn't occur in the training data.

But there's one issue with this strategy: it doesn't offer a way to control the amount of randomness in the sampling process.

Generative deep learning

Text generation

A brief history of generative deep learning for sequence generation

How do you generate sequence data?

The importance of the sampling strategy

****Reweighting a probability distribution to a different temperature****

"""

import numpy as np

def reweight_distribution(original_distribution, temperature=0.5):

distribution = np.log(original_distribution) / temperature

distribution = np.exp(distribution)

return distribution / np.sum(distribution)

"""### Implementing text generation with Keras

Preparing the data

```
**Downloading and uncompressing the IMDB movie reviews dataset**
****
```

```
!wget https://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz
!tar -xf aclImdb_v1.tar.gz
```

```
*****Creating a dataset from text files (one file = one sample)*****
```

```
import tensorflow as tf
from tensorflow import keras
dataset = keras.utils.text_dataset_from_directory(
    directory="aclImdb", label_mode=None, batch_size=256)
dataset = dataset.map(lambda x: tf.strings.regex_replace(x, "<br />", " "))
```

```
*****Preparing a `TextVectorization` layer*****
```

```
from tensorflow.keras.layers import TextVectorization
```

```
sequence_length = 100
vocab_size = 15000
text_vectorization = TextVectorization(
    max_tokens=vocab_size,
    output_mode="int",
    output_sequence_length=sequence_length,
)
text_vectorization.adapt(dataset)
```

```
*****Setting up a language modeling dataset*****
```

```
def prepare_lm_dataset(text_batch):
    vectorized_sequences = text_vectorization(text_batch)
    x = vectorized_sequences[:, :-1]
    y = vectorized_sequences[:, 1:]
    return x, y
```

```
lm_dataset = dataset.map(prepare_lm_dataset, num_parallel_calls=4)
```

```
#### A Transformer-based sequence-to-sequence model####
```

```
import tensorflow as tf
from tensorflow.keras import layers
```

```
class PositionalEmbedding(layers.Layer):
    def __init__(self, sequence_length, input_dim, output_dim, **kwargs):
        super().__init__(**kwargs)
```

```

self.token_embeddings = layers.Embedding(
    input_dim=input_dim, output_dim=output_dim)
self.position_embeddings = layers.Embedding(
    input_dim=sequence_length, output_dim=output_dim)
self.sequence_length = sequence_length
self.input_dim = input_dim
self.output_dim = output_dim

```

```

def call(self, inputs):
    length = tf.shape(inputs)[-1]
    positions = tf.range(start=0, limit=length, delta=1)
    embedded_tokens = self.token_embeddings(inputs)
    embedded_positions = self.position_embeddings(positions)
    return embedded_tokens + embedded_positions

```

```

def compute_mask(self, inputs, mask=None):
    return tf.math.not_equal(inputs, 0)

```

```

def get_config(self):
    config = super(PositionalEmbedding, self).get_config()
    config.update({
        "output_dim": self.output_dim,
        "sequence_length": self.sequence_length,
        "input_dim": self.input_dim,
    })
    return config

```

```

class TransformerDecoder(layers.Layer):
    def __init__(self, embed_dim, dense_dim, num_heads, **kwargs):
        super().__init__(**kwargs)
        self.embed_dim = embed_dim
        self.dense_dim = dense_dim
        self.num_heads = num_heads
        self.attention_1 = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=embed_dim)
        self.attention_2 = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=embed_dim)
        self.dense_proj = keras.Sequential(
            [layers.Dense(dense_dim, activation="relu"),
             layers.Dense(embed_dim),]
        )
        self.layernorm_1 = layers.LayerNormalization()
        self.layernorm_2 = layers.LayerNormalization()
        self.layernorm_3 = layers.LayerNormalization()
        self.supports_masking = True

```

```

def get_config(self):
    config = super(TransformerDecoder, self).get_config()
    config.update({
        "embed_dim": self.embed_dim,
        "num_heads": self.num_heads,
        "dense_dim": self.dense_dim,
    })
    return config

def get_causal_attention_mask(self, inputs):
    input_shape = tf.shape(inputs)
    batch_size, sequence_length = input_shape[0], input_shape[1]
    i = tf.range(sequence_length)[:, tf.newaxis]
    j = tf.range(sequence_length)
    mask = tf.cast(i >= j, dtype="int32")
    mask = tf.reshape(mask, (1, input_shape[1], input_shape[1]))
    mult = tf.concat(
        [tf.expand_dims(batch_size, -1),
         tf.constant([1, 1], dtype=tf.int32)], axis=0)
    return tf.tile(mask, mult)

def call(self, inputs, encoder_outputs, mask=None):
    causal_mask = self.get_causal_attention_mask(inputs)
    if mask is not None:
        padding_mask = tf.cast(
            mask[:, tf.newaxis, :], dtype="int32")
        padding_mask = tf.minimum(padding_mask, causal_mask)
    attention_output_1 = self.attention_1(
        query=inputs,
        value=inputs,
        key=inputs,
        attention_mask=causal_mask)
    attention_output_1 = self.layernorm_1(inputs + attention_output_1)
    attention_output_2 = self.attention_2(
        query=attention_output_1,
        value=encoder_outputs,
        key=encoder_outputs,
        attention_mask=padding_mask,
    )
    attention_output_2 = self.layernorm_2(
        attention_output_1 + attention_output_2)
    proj_output = self.dense_proj(attention_output_2)
    return self.layernorm_3(attention_output_2 + proj_output)

```

*****A simple Transformer-based language model*****

```

from tensorflow.keras import layers
embed_dim = 256
latent_dim = 2048
num_heads = 2

inputs = keras.Input(shape=(None,), dtype="int64")
x = PositionalEmbedding(sequence_length, vocab_size, embed_dim)(inputs)
x = TransformerDecoder(embed_dim, latent_dim, num_heads)(x, x)
outputs = layers.Dense(vocab_size, activation="softmax")(x)
model = keras.Model(inputs, outputs)
model.compile(loss="sparse_categorical_crossentropy", optimizer="rmsprop")

"""### A text-generation callback with variable-temperature sampling

**The text-generation callback**
"""

import numpy as np

tokens_index = dict(enumerate(text_vectorization.get_vocabulary()))

def sample_next(predictions, temperature=1.0):
    predictions = np.asarray(predictions).astype("float64")
    predictions = np.log(predictions) / temperature
    exp_preds = np.exp(predictions)
    predictions = exp_preds / np.sum(exp_preds)
    probas = np.random.multinomial(1, predictions, 1)
    return np.argmax(probas)

class TextGenerator(keras.callbacks.Callback):
    def __init__(self,
                 prompt,
                 generate_length,
                 model_input_length,
                 temperatures=(1.,),
                 print_freq=1):
        self.prompt = prompt
        self.generate_length = generate_length
        self.model_input_length = model_input_length
        self.temperatures = temperatures
        self.print_freq = print_freq

    def on_epoch_end(self, epoch, logs=None):
        if (epoch + 1) % self.print_freq != 0:
            return

```

```

for temperature in self.temperatures:
    print("== Generating with temperature", temperature)
    sentence = self.prompt
    for i in range(self.generate_length):
        tokenized_sentence = text_vectorization([sentence])
        predictions = self.model(tokenized_sentence)
        next_token = sample_next(predictions[0, i, :])
        sampled_token = tokens_index[next_token]
        sentence += " " + sampled_token
    print(sentence)

prompt = "This movie"
text_gen_callback = TextGenerator(
    prompt,
    generate_length=50,
    model_input_length=sequence_length,
    temperatures=(0.2, 0.5, 0.7, 1., 1.5))

"""**Fitting the language model**"""

model.fit(lm_dataset, epochs=200, callbacks=[text_gen_callback])

```

5.2 Deep Dream

The name "Deep Dream" was inspired by the movie Inception, reflecting the recursive nature of the process where the network "dreams" within its layers, enhancing patterns iteratively.

How Deep Dream Works?

At its core, Deep Dream uses a CNN trained on a massive dataset of images. A CNN is composed of layers of interconnected nodes, or neurons, each layer responsible for recognizing different levels of features within an image—from simple edges to complex objects.

When an image is fed into Deep Dream, the program identifies and enhances the patterns it has learned to recognize. This is achieved through a process called "inceptionism," where the network is instructed to maximize the detection of features at various layers. For instance, if the network identifies a pattern resembling a dog, it will amplify this pattern, making the image appear as though it is filled with dog-like shapes and forms.



Code Implementation of Deep Dream

The code implementation of deep dream are as follows:

Image used :

Step 1: Importing the libraries

We at first import all the necessary libraries required for our analysis.

```
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
tf.__version__
```

Step 2: Loading the pre-built convolutional neural network

- **Load Pre-Trained Model:** Load the InceptionV3 model with ImageNet weights, excluding final classification layers.
- **Model Summary:** Use `base_model.summary()` to display the model's architecture and details.
- **Select Layers:** Choose intermediate layers mixed3 and mixed5 for output extraction.
- **Create Output List:** Compile a list of outputs from the selected layers.
- **Define New Model:** Create `deep_dream_model` using the same input as InceptionV3, but with outputs from the selected layers.

- **Purpose:** Utilize this new model for DeepDream experiments to enhance or visualize features learned at various network depths.

```
base_model=tf.keras.applications.InceptionV3(include_top=False, weights='imagenet')
base_model.summary()
```

```
# Relu
#names = ['mixed3', 'mixed5', 'mixed8', 'mixed9']
names = ['mixed3', 'mixed5']
layers = [base_model.get_layer(name).output for name in names]
deep_dream_model = tf.keras.Model(inputs = base_model.input, outputs = layers)
```

Step 3 : Loading and pre-processing the image

- **Load and Resize Image:** Use *tf.keras.preprocessing.image.load_img()* to load an image from a specified file path and resize it to 225x375 pixels.
- **Display Image:** Display the resized image using *plt.imshow()*.
- **Convert Image to Numpy Array:** Convert the PIL Image object to a Numpy array with *tf.keras.preprocessing.image.img_to_array()*, making it suitable for model processing.
- **Normalize Pixel Values:** Commented out line for normalizing pixel values by dividing by 255 (not executed).
- **Preprocess Image:** Use *tf.keras.applications.inception_v3.preprocess_input()* to apply necessary transformations for InceptionV3, adjusting pixel values to match the model's training format for accurate predictions or feature extraction.

```
image = tf.keras.preprocessing.image.load_img('/content/StaryNight.jpg',
                                              target_size=(225, 375))

plt.imshow(image)
image = tf.keras.preprocessing.image.img_to_array(image)
# image = image / 255
image = tf.keras.applications.inception_v3.preprocess_input(image)
```



Step 4: Getting the activations

The code expands the single image into a batch for model prediction, then retrieves activations from the mixed3 and mixed5 layers of the DeepDream model. It finally checks the shapes of these activations to confirm the dimensions of the feature maps extracted from these intermediate layers.

```
image_batch = tf.expand_dims(image, axis = 0)
activations = deep_dream_model.predict(image_batch)
activations[0].shape, activations[1].shape
```

Step 5: Calculating the loss

- **Expand Image Tensor:** Expand the image tensor into a batch of size one.
- **Pass Through Model:** Pass the image batch through the custom DeepDream model to obtain activations from intermediate layers.
- **Calculate Mean Activation:** Compute the mean activation value for each layer, representing the average intensity of features.
- **Collect Mean Values:** Gather these mean values into a list.
- **Compute Final Loss:** Sum all mean activation values to obtain the final loss.
- **Purpose:** The loss quantifies the overall intensity of features across different layers, used to optimize and enhance the image's features in the DeepDream process.

```
def calculate_loss(image, network):
    image_batch = tf.expand_dims(image, axis = 0)
    activations = network(image_batch)

    losses = []
    for act in activations:
        loss = tf.math.reduce_mean(act)
        losses.append(loss)
    #print(losses)
    #print(np.shape(losses))
    #print(tf.reduce_sum(losses))
    return tf.reduce_sum(losses)
loss = calculate_loss(image, deep_dream_model)
```

Step 6: Backpropagation

- **Define DeepDream Algorithm:** The code defines a basic DeepDream algorithm using TensorFlow and Keras.
- **DeepDream Function:**
 - o **Compute Loss:** Calculate the loss from activations of intermediate layers.
 - o **Calculate Gradients:** Compute gradients with respect to the image.
 - o **Normalize Gradients:** Normalize the gradients.
 - o **Update Image:** Adjust the image to maximize the loss, accentuating features.
- **Inverse Transform Function:**
 - o **Rescale Image:** Convert the image from the $[-1, 1]$ range to $[0, 255]$.
 - o **Convert Format:** Change the image to an 8-bit format suitable for display.
- **Run DeepDream Function:**

- o **Manage Process:** Run the DeepDream optimization for a specified number of epochs.
- o **Display Progress:** Periodically display the evolving image and print the current loss value to track feature enhancement progress.

```
# Compare the activations with the pixels
# Emphasize parts of the image
# Change the pixels of the input image
```

```
@tf.function
def deep_dream(network, image, learning_rate):
    with tf.GradientTape() as tape:
        tape.watch(image)
        loss = calculate_loss(image, network)

    gradients = tape.gradient(loss, image) # Derivate
    gradients /= tf.math.reduce_std(gradients)
    image = image + gradients * learning_rate
    image = tf.clip_by_value(image, -1, 1)

    return loss, image

def inverse_transform(image):
    image = 255 * (image + 1.0) / 2.0
    return tf.cast(image, tf.uint8)

def run_deep_dream(network, image, epochs, learning_rate):
    for epoch in range(epochs):
        loss, image = deep_dream(network, image, learning_rate)

        if epoch % 200 == 0:
            plt.figure(figsize=(12,12))
            plt.imshow(inverse_transform(image))
            plt.show()
            print('Epoch {}, loss {}'.format(epoch, loss))
```

Step 7: Generating images

The **run_deep_dream** function leverages a pre-trained model to enhance features in an image through iterative optimization, adjusting the image to maximize activation loss, and producing a visually enhanced output.

```
run_deep_dream(network=deep_dream_model,    image=image,    epochs    =    8000,
learning_rate=0.000
```

Output:

```
Epoch 0, loss 0.6535599231719971
Epoch 200, loss 0.9151648283004761
```

Epoch 400, loss 1.1114864349365234

...



5.3 Neural Style Transfer with Tensor Flow:

Neural Style Transfer (NST) is a method in deeplearning where the details of one picture are combined with the artistic style of another to create a new image. This process allows you to generate an image that has the subject of the first picture but the art style, colors and textures of the second. In this guide we will explain how NST works and how to apply it using TensorFlow.

Understanding Neural Style Transfer

Neural Style Transfer or NST involves three different images:

- **Content Image:** This is the picture with the main subject or scene you want to change. It holds the basic structure and layout you want to keep
- **Style Image:** the picture with the artistic feel, like colors, textures and patterns. You want to use this style to change how your content image looks.
- **Generated Image:** The output image that combines the content of the first image with the style of the second.



Implementing Neural Style Transfer with TensorFlow

In implementation of NST we'll use the VGG19 model, pre-trained on the ImageNet dataset to extract features from our images.

Step 1: Import Necessary Libraries

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import math
```

```
from tensorflow.keras.applications.vgg19 import VGG19, preprocess_input
from tensorflow.keras.preprocessing.image import load_img, img_to_array
from tensorflow.keras.models import Model
```

Step 2: Image processing

Now, we load and process the image using Keras preprocess input in VGG 19. The *expand_dims* function adds a dimension to represent a number of images in the input. This *preprocess_input* function used in VGG 19 converts the input RGB to BGR images and centre these values around 0 according to ImageNet data.

```
def load_and_process_image(image_path):
    img = load_img(image_path)
    img = img_to_array(img)
    img = preprocess_input(img)
    img = np.expand_dims(img, axis=0)
    return img

def deprocess(img):
    img[:, :, 0] += 103.939
    img[:, :, 1] += 116.779
    img[:, :, 2] += 123.6
    img = img[:, :, ::-1]
    img = np.clip(img, 0, 255).astype('uint8')
    return img

def display_image(image):
    if len(image.shape) == 4:
        img = np.squeeze(image, axis=0)
    img = deprocess(img)

    plt.grid(False)
    plt.xticks([])
    plt.yticks([])
    plt.imshow(img)
    return
```

Now we use the above function to display the style and content images

```
content_img = load_and_process_image(content_path)
display_image(content_img)
style_img = load_and_process_image(style_path)
display_image(style_img)
```



Content Image



Style Image

Step 3: Model Initialization

Now we initialize the VGG model with ImageNet weights we will also remove the top layers and make it non-trainable. We use VGG19 to extract image features.

- **include_top=False** removes the classification head.
- We freeze the model to use it as a **fixed feature extractor**.

```
model = VGG19(  
    include_top=False,  
    weights='imagenet'  
)  
model.trainable = False  
  
model.summary()
```

Output:

Model: "vgg19"

Layer (type)	Output Shape	Param #
input_layer_2 (InputLayer)	(None, None, None, 3)	0
block1_conv1 (Conv2D)	(None, None, None, 64)	1,792
block1_conv2 (Conv2D)	(None, None, None, 64)	36,928
block1_pool (MaxPooling2D)	(None, None, None, 64)	0
block2_conv1 (Conv2D)	(None, None, None, 128)	73,856
block2_conv2 (Conv2D)	(None, None, None, 128)	147,584
block2_pool (MaxPooling2D)	(None, None, None, 128)	0
block3_conv1 (Conv2D)	(None, None, None, 256)	295,168
block3_conv2 (Conv2D)	(None, None, None, 256)	590,080
block3_conv3 (Conv2D)	(None, None, None, 256)	590,080
block3_conv4 (Conv2D)	(None, None, None, 256)	590,080
block3_pool (MaxPooling2D)	(None, None, None, 256)	0
block4_conv1 (Conv2D)	(None, None, None, 512)	1,180,160
block4_conv2 (Conv2D)	(None, None, None, 512)	2,359,808
block4_conv3 (Conv2D)	(None, None, None, 512)	2,359,808
block4_conv4 (Conv2D)	(None, None, None, 512)	2,359,808
block4_pool (MaxPooling2D)	(None, None, None, 512)	0
block5_conv1 (Conv2D)	(None, None, None, 512)	2,359,808
block5_conv2 (Conv2D)	(None, None, None, 512)	2,359,808
block5_conv3 (Conv2D)	(None, None, None, 512)	2,359,808
block5_conv4 (Conv2D)	(None, None, None, 512)	2,359,808
block5_pool (MaxPooling2D)	(None, None, None, 512)	0

Step 4: Content Model defining

Now we define the content and style model using Keras.Model API. The content model takes the image as input and output the feature map from "**block5_conv1**" from the above VGG model.

```
content_layer = 'block5_conv1'
content_model = Model(
    inputs=model.input,
    outputs=model.get_layer(content_layer).output
)
content_model.summary()
```

Output:

Model: "functional_5"

Layer (type)	Output Shape	Param #
input_layer_2 (InputLayer)	(None, None, None, 3)	0
block1_conv1 (Conv2D)	(None, None, None, 64)	1,792
block1_conv2 (Conv2D)	(None, None, None, 64)	36,928
block1_pool (MaxPooling2D)	(None, None, None, 64)	0
block2_conv1 (Conv2D)	(None, None, None, 128)	73,856
block2_conv2 (Conv2D)	(None, None, None, 128)	147,584
block2_pool (MaxPooling2D)	(None, None, None, 128)	0
block3_conv1 (Conv2D)	(None, None, None, 256)	295,168
block3_conv2 (Conv2D)	(None, None, None, 256)	590,080
block3_conv3 (Conv2D)	(None, None, None, 256)	590,080
block3_conv4 (Conv2D)	(None, None, None, 256)	590,080
block3_pool (MaxPooling2D)	(None, None, None, 256)	0
block4_conv1 (Conv2D)	(None, None, None, 512)	1,180,160
block4_conv2 (Conv2D)	(None, None, None, 512)	2,359,808
block4_conv3 (Conv2D)	(None, None, None, 512)	2,359,808
block4_conv4 (Conv2D)	(None, None, None, 512)	2,359,808
block4_pool (MaxPooling2D)	(None, None, None, 512)	0
block5_conv1 (Conv2D)	(None, None, None, 512)	2,359,808
block5_conv2 (Conv2D)	(None, None, None, 512)	2,359,808

Total params: 15,304,768 (58.38 MB)
Trainable params: 0 (0.00 B)
Non-trainable params: 15,304,768 (58.38 MB)

Step 5: Style Model defining:

Now, we define the content and style model using Keras.Model API. The style model takes an image as input and output the feature map from "**block1_conv1, block3_conv1 and block5_conv2**" from the above VGG model. We use multiple layers for style to capture texture patterns at different scales. Each layer captures style at different levels of abstraction like edges, patterns and textures.

```
style_layers = [
    'block1_conv1',
    'block3_conv1',
    'block5_conv1'
]
style_models = [Model(inputs=model.input,
                      outputs=model.get_layer(layer).output) for layer in style_layers]
```

Step 6: Training Function

In this step we define the `training_loop` function that optimizes the generated image using both content and style losses. It begins by loading and preprocessing the content and style images then initializes the generated image as a trainable variable.

- **GradientTape** watches changes in the generated image.
- **J_content**: difference in content features.
- **J_style**: difference in style (Gram matrices).
- **J_total**: weighted combination using `a` and `b`.

```
generated_images = []
```

```
def training_loop(content_path, style_path, iterations=50, a=10, b=1000):
```

```
    content = load_and_process_image(content_path)
    style = load_and_process_image(style_path)
    generated = tf.Variable(content, dtype=tf.float32)
```

```
    opt = tf.keras.optimizers.Adam(learning_rate=0.7)
```

```
    best_cost = math.inf
```

```
    best_image = None
```

```
    for i in range(iterations):
```

```
        start_time_cpu = time.process_time()
```

```
        start_time_wall = time.time()
```

```
        with tf.GradientTape() as tape:
```

```
            J_content = content_cost(style, generated)
```

```
            J_style = style_cost(style, generated)
```

```
            J_total = a * J_content + b * J_style
```

```
        grads = tape.gradient(J_total, generated)
```

```
        opt.apply_gradients([(grads, generated)])
```

```
    end_time_cpu = time.process_time()
```

```
    end_time_wall = time.time()
```

```
    cpu_time = end_time_cpu - start_time_cpu
```

```
    wall_time = end_time_wall - start_time_wall
```

```
    if J_total < best_cost:
```

```
        best_cost = J_total
```

```
        best_image = generated.numpy()
```

```
    print("CPU times: user {} μs, sys: {} ns, total: {} μs".format(
        int(cpu_time * 1e6),
        int((end_time_cpu - start_time_cpu) * 1e9),
        int((end_time_cpu - start_time_cpu + 1e-6) * 1e6))
    )
```

```

print("Wall time: {:.2f} μs".format(wall_time * 1e6))
print("Iteration :{}".format(i))
print("Total Loss {:.e}.".format(J_total))
generated_images.append(generated.numpy())

```

```

return best_image

```

Step 7: Model Training

```

final_img = training(content_path, style_path)

```

```

CPU times: user 284656 μs, sys: 284656413 ns, total: 284657 μs
Wall time: 289904.12 μs
Iteration :0
Total Loss 1.761738e+12.
CPU times: user 276058 μs, sys: 276058034 ns, total: 276059 μs
Wall time: 277400.49 μs
Iteration :1
Total Loss 1.710360e+12.
CPU times: user 275056 μs, sys: 275056399 ns, total: 275057 μs
Wall time: 275687.46 μs
Iteration :2
Total Loss 1.654856e+12.
CPU times: user 273576 μs, sys: 273576047 ns, total: 273577 μs
Wall time: 274424.31 μs
Iteration :3
Total Loss 1.598882e+12.
CPU times: user 282772 μs, sys: 282772656 ns, total: 282773 μs
Wall time: 284516.57 μs
Iteration :4
Total Loss 1.542978e+12.
CPU times: user 284965 μs, sys: 284965087 ns, total: 284966 μs
Wall time: 284451.25 μs
Iteration :5
Total Loss 1.487192e+12.
CPU times: user 278286 μs, sys: 278286268 ns, total: 278287 μs
Wall time: 279321.91 μs
Iteration :6
Total Loss 1.431401e+12.
CPU times: user 292246 μs, sys: 292246131 ns, total: 292247 μs
Wall time: 290832.04 μs
Iteration :7
Total Loss 1.375675e+12.
CPU times: user 289826 μs, sys: 289826715 ns, total: 289827 μs
Wall time: 290276.53 μs
Iteration :8
Total Loss 1.320012e+12.
CPU times: user 266069 μs, sys: 266069423 ns, total: 266070 μs
Wall time: 266137.60 μs
Iteration :9

```

Step 8: Model Prediction

In the final step, we plot the final and intermediate results.

```

plt.figure(figsize=(12, 12))

```

```

for i in range(10):
    plt.subplot(4, 3, i + 1)
    display_image(generated_images[i+39])
plt.show()

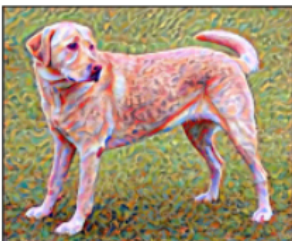
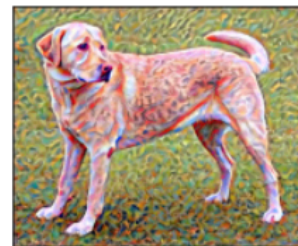
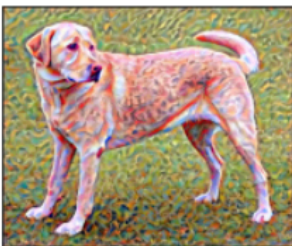
```

```

display_image(final_img)

```

Output:



5.4 Last 10 Generated images

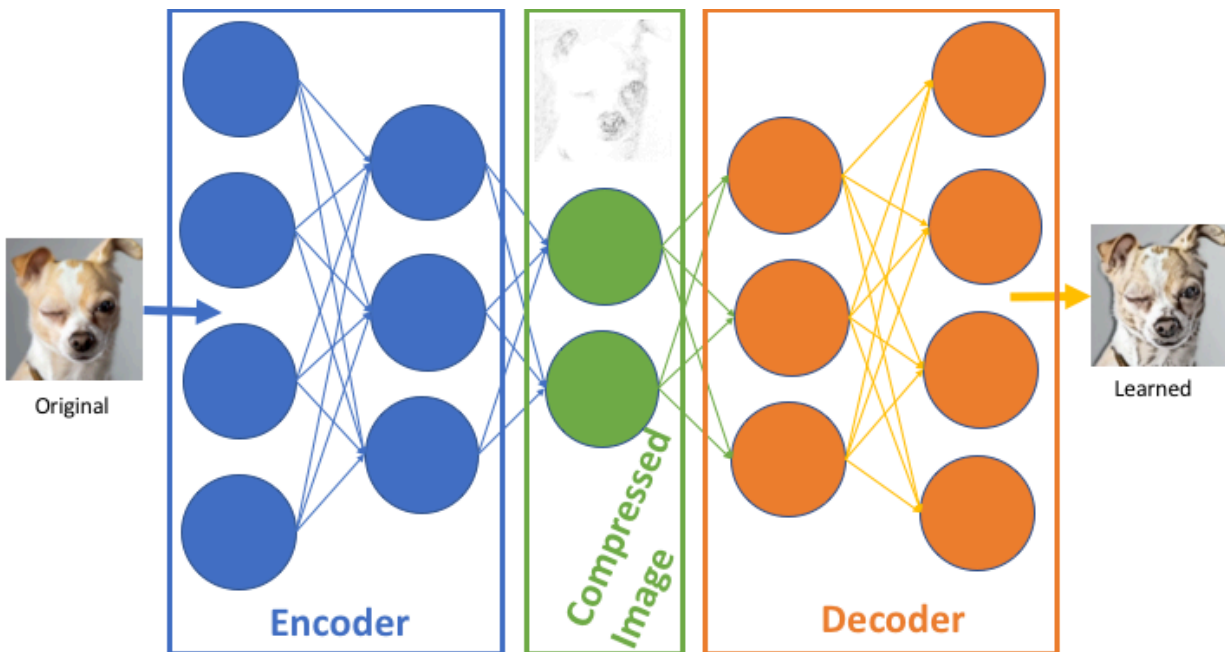


Best Generated image

This output shows the last 10 generated images during Neural Style Transfer where the content (dog) is preserved and style patterns are gradually applied. Each image reflects slight improvements in blending style with content across iterations.

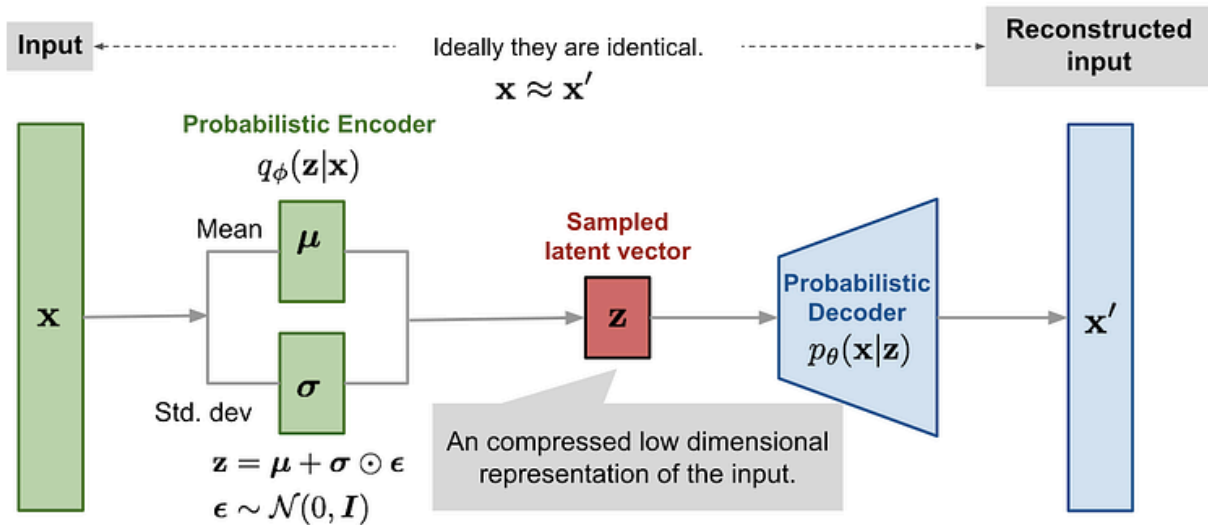
Autoencoder

Autoencoder is a very simple neural network structure that consists of two parts, the encoder, and the decoder. Encoder is the one that learns to compress the original input into a compressed form in much smaller vector space and the decoder it the one learns to reconstruct the compressed data into the original input with some loss.



Variational Encoder (VAE)

We observed that traditional autoencoders learn to compress and reconstruct data but not really help with generating new data. This is where Variational Encoder (VAE) came in handy. VAE learns the distribution of the data instead of just a compressed image, and by using the distribution, we can decode and generate new data. The encoder is trying to learn the parameters ϕ to compress data input x to a latent vector z , and the output encoding z is drawn from Gaussian density with parameters ϕ . As for the decoder, its input is encoding z , the output from the encoder. It parametrizes the reconstructed x' over parameters θ , and the output x' is drawn from the distribution of the data.



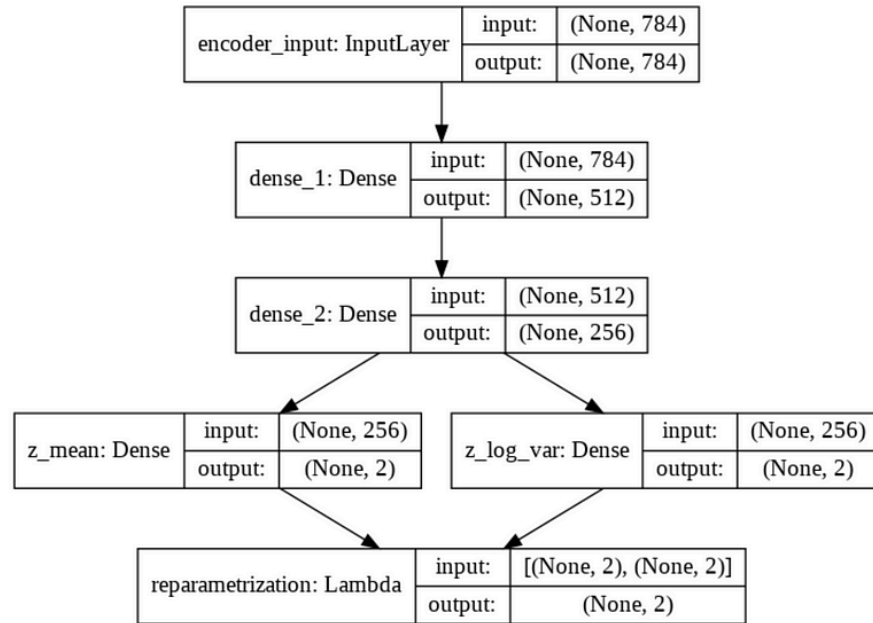
Build Encoder

We used two dense layers to construct the encoder

```

1  # build encoder model
2  inputs = Input(shape=input_shape, name='encoder_input')
3  x = Dense(intermediate_dim, activation='relu')(inputs)
4  x = Dense(second_intermediate_dim, activation='relu')(x)
5  z_mean = Dense(latent_dim, name='z_mean')(x)
6  z_log_var = Dense(latent_dim, name='z_log_var')(x)
7
8  # use reparameterization trick to push the sampling out as input
9  # note that "output_shape" isn't necessary with the TensorFlow backend
10 z = Lambda(sampling, output_shape=(latent_dim,), name='reparametrization')([z_mean, z_log_var])
11
12 # instantiate encoder model
13 encoder = Model(inputs, [z_mean, z_log_var, z], name='encoder')

```



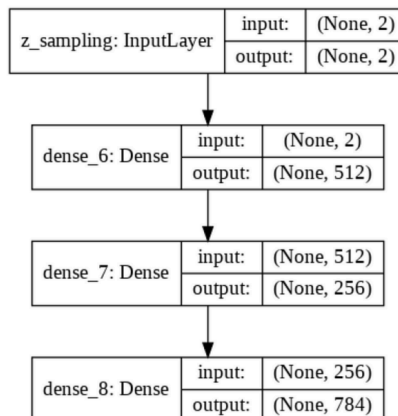
Build Decoder

We use the same layers as the encoder to reconstruct images

```

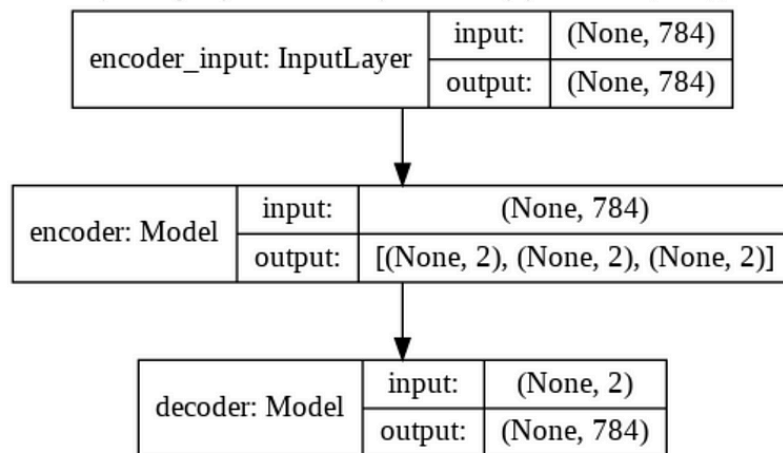
1  # build decoder model
2  latent_inputs = Input(shape=(latent_dim,), name='z_sampling')
3  x = Dense(intermediate_dim, activation='relu')(latent_inputs)
4  x = Dense(second_intermediate_dim, activation='relu')(x)
5  outputs = Dense(original_dim, activation='sigmoid')(x)
6
7  # instantiate decoder model
8  decoder = Model(latent_inputs, outputs, name='decoder')

```



Instantiate VAE

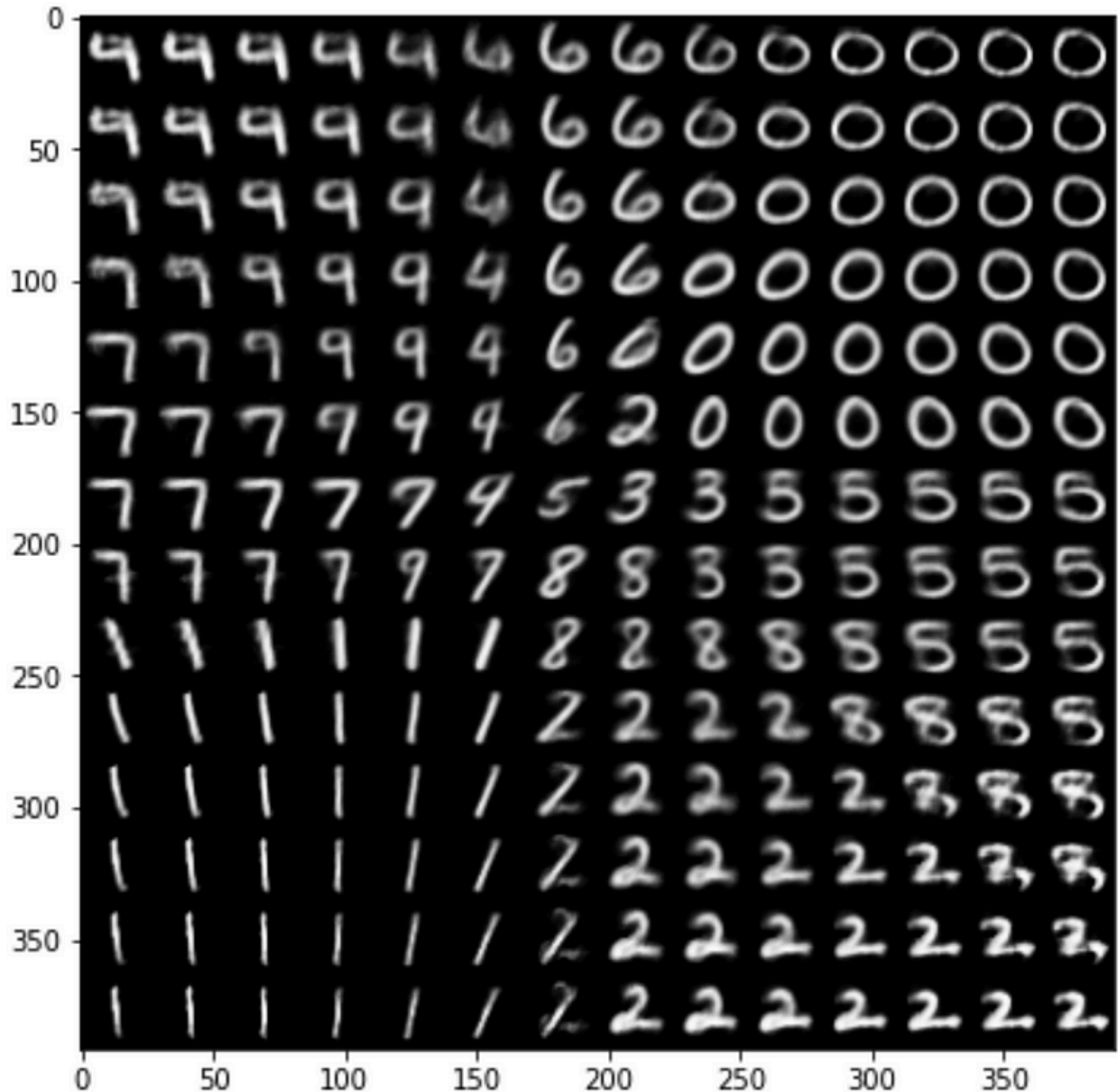
```
1 # instantiate VAE model
2 outputs = decoder(encoder(inputs)[2])
3 vae = Model(inputs, outputs, name='vae_mlp')
```



Generating Handwritten Images

After we have trained the VAE, we use only the decoder to generate images by providing it a vector of size two. We generate an interpolation graph by providing the decoder two numbers from -3 to 4, and we generate 196 of those images.

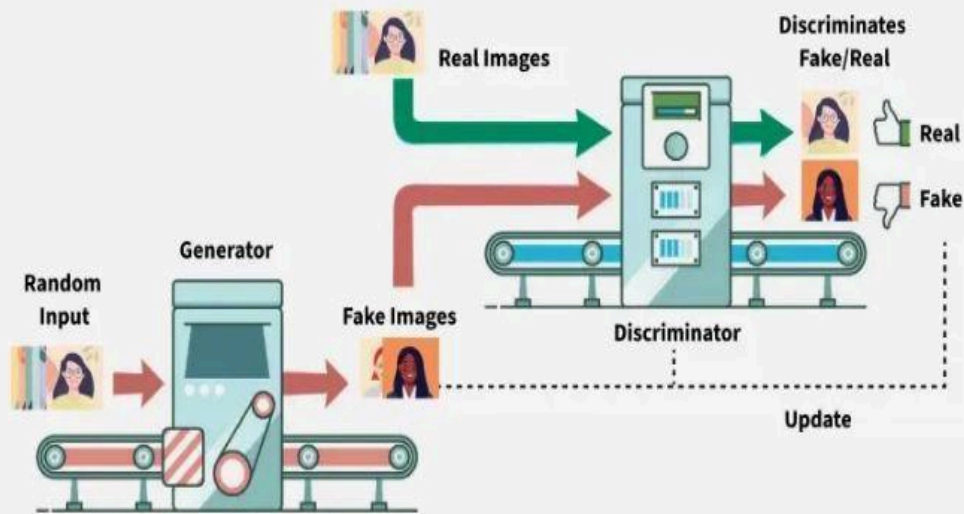
```
1 values = np.arange(-3, 4, .5)
2 xx, yy = np.meshgrid(values, values)
3 input_holder = np.zeros((1, 2))
4 # Matrix that will contain the grid of images
5 container = np.zeros((image_size * len(values), image_size * len(values)))
6
7 for row in range(xx.shape[0]):
8     for col in range(xx.shape[1]):
9         input_holder = np.array([[xx[row, col], yy[row, col]]])
10        output = decoder.predict(input_holder)
11        artificial_image = output.reshape((image_size, image_size))
12        container[row * image_size: (row + 1) * image_size, col * image_size: (col + 1) * image_size] = artificial_image
13
14 plt.figure(figsize=(50,7))
15 plt.imshow(container, cmap='gray')
```



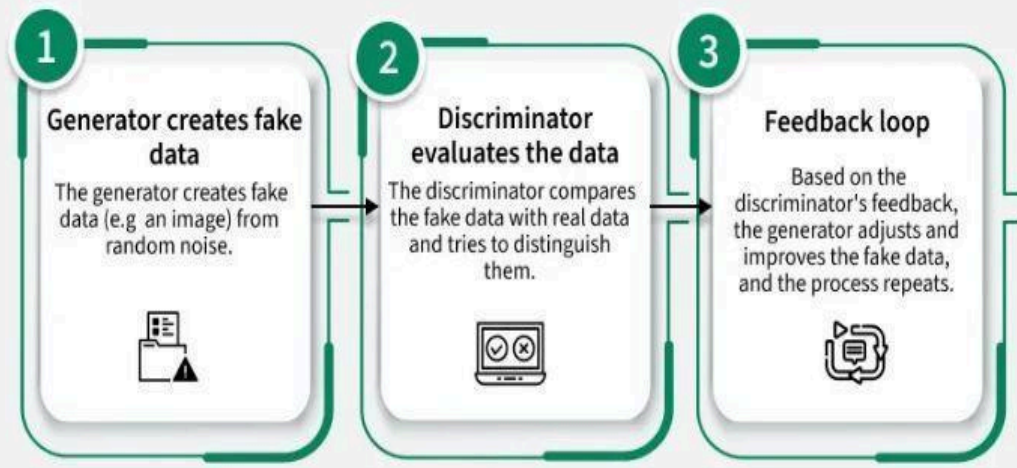
5.5 Generative Adversarial Network (GAN)

Generative Adversarial Networks (GANs) help machines to create new, realistic data by learning from existing examples. It is introduced by Ian Goodfellow and his team in 2014 and they have transformed how computers generate images, videos, music and more. Unlike traditional models that only recognize or classify data, they take a creative way by generating entirely new content that closely resembles real-world data. This ability helped various fields such as art, gaming, healthcare and data science.

Generative Adversarial Network (GANs)



How GANs Work ?



Generative Adversarial Network vs Traditional Machine Learning



Synthetic Data



Generator vs Discriminator



Adversarial Training



Realistic Output



Predict Outcomes



Supervised Learning



Minimize Error



Regression & Classification

Applications of GANs

Image Generation

GANs are widely used in creating realistic images (e.g faces, art).



Super Resolution

GANs improve the resolution of images or videos.



Data Augmentation

GANs generate more data for training machine learning models.



Text-to-Image

GANs convert textual descriptions into corresponding images



Architecture of GANs

GANs consist of two main models that work together to create realistic synthetic data which are as follows:

1. Generator Model

The generator is a deep neural network that takes random noise as input to generate realistic data samples like images or text. It learns the underlying data patterns by adjusting its internal parameters during training through backpropagation. Its objective is to produce samples that the discriminator classifies as real.

Generator Loss Function: The generator tries to minimize this loss:

$$J_G = -\frac{1}{m} \sum_{i=1}^m \log D(G(z_i))$$

where

- J_G measure how well the generator is fooling the discriminator.
- $G(z_i)$ is the generated sample from random noise z_i
- $D(G(z_i))$ is the discriminator's estimated probability that the generated sample is real.

The generator aims to maximize $D(G(z_i))$ meaning it wants the discriminator to classify its fake data as real (probability close to 1).

2. Discriminator Model

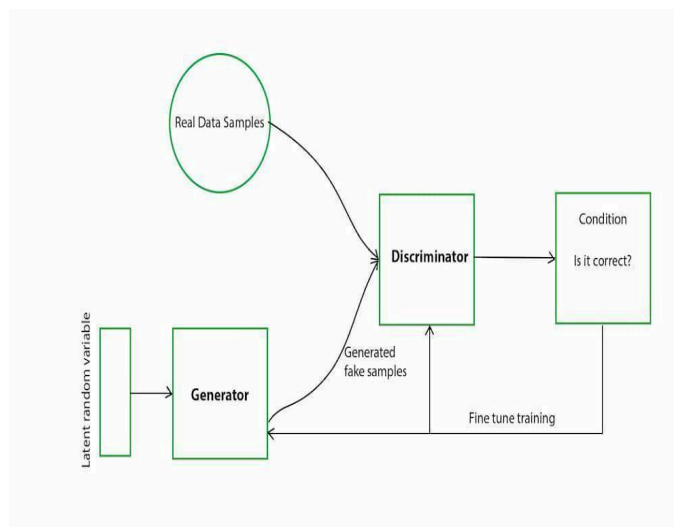
The discriminator acts as a binary classifier helps in distinguishing between real and generated data. It learns to improve its classification ability through training, refining its parameters to detect fake samples more accurately. When dealing with image data, the discriminator uses convolutional layers or other relevant architectures which help to extract features and enhance the model's ability.

Discriminator Loss Function: The discriminator tries to minimize this loss:

$$J_D = -\frac{1}{m} \sum_{i=1}^m \log D(x_i) - \frac{1}{m} \sum_{i=1}^m \log(1 - D(G(z_i)))$$

- J_D measures how well the discriminator classifies real and fake samples.
- x_i is a real data sample.
- $G(z_i)$ is a fake sample from the generator.
- $D(x_i)$ is the discriminator's probability that x_i is real.
- $D(G(z_i))$ is the discriminator's probability that the fake sample is real.

The discriminator wants to correctly classify real data as real (maximize $\log D(x_i)$) and fake data as fake (maximize $\log(1 - D(G(z_i)))$)



How does a GAN work?

GANs train by having two networks the Generator (G) and the Discriminator (D) compete and improve together. Here's the step-by-step process

1. Generator's First Move

The generator starts with a random noise vector like random numbers. It uses this noise as a starting point to create a fake data sample such as a generated image. The generator's internal layers transform this noise into something that looks like real data.

2. Discriminator's Turn

The discriminator receives two types of data:

- Real samples from the actual training dataset.
- Fake samples created by the generator.

D's job is to analyze each input and find whether it's real data or something G cooked up. It outputs a probability score between 0 and 1. A score of 1 shows the data is likely real and 0 suggests it's fake.

3. Adversarial Learning

- If the discriminator correctly classifies real and fake data it gets better at its job.
- If the generator fools the discriminator by creating realistic fake data, it receives a positive update and the discriminator is penalized for making a wrong decision.

4. Generator's Improvement

- Each time the discriminator mistakes fake data for real, the generator learns from this success.
- Through many iterations, the generator improves and creates more convincing fake samples.

5. Discriminator's Adaptation

- The discriminator also learns continuously by updating itself to better spot fake data.
- This constant back-and-forth makes both networks stronger over time.

6. Training Progression

- As training continues, the generator becomes highly proficient at producing realistic data.
- Eventually the discriminator struggles to distinguish real from fake shows that the GAN has reached a well-trained state.
- At this point, the generator can produce high-quality synthetic data that can be used for different applications.

Types of GANs

There are several types of GANs each designed for different purposes. Here are some important types:

1. Vanilla GAN

Vanilla GAN is the simplest type of GAN. It consists of:

- A generator and a discriminator both are built using multi-layer perceptrons (MLPs).
- The model optimizes its mathematical formulation using stochastic gradient descent (SGD).

While foundational, Vanilla GANs can face problems like:

- **Mode collapse:** The generator produces limited types of outputs repeatedly.
- **Unstable training:** The generator and discriminator may not improve smoothly.

2. Conditional GAN (CGAN)

Conditional GANs (CGANs) adds an additional conditional parameter to guide the generation process. Instead of generating data randomly they allow the model to produce specific types of outputs.

Working of CGANs:

- A conditional variable (y) is fed into both the generator and the discriminator.
- This ensures that the generator creates data corresponding to the given condition (e.g generating images of specific objects).
- The discriminator also receives the labels to help distinguish between real and fake data.

Example: Instead of generating any random image, CGAN can generate a specific object like a dog or a cat based on the label.

3. Deep Convolutional GAN (DCGAN)

Deep Convolutional GANs (DCGANs) are among the most popular types of GANs used for image generation.

They are important because they:

- Uses Convolutional Neural Networks (CNNs) instead of simple multi-layer perceptrons (MLPs).
- Max pooling layers are replaced with convolutional stride helps in making the model more efficient.
- Fully connected layers are removed, which allows for better spatial understanding of images.

DCGANs are successful because they generate high-quality, realistic images.

4. Laplacian Pyramid GAN (LAPGAN)

Laplacian Pyramid GAN (LAPGAN) is designed to generate ultra-high-quality images by using a multi-resolution approach.

Working of LAPGAN:

- Uses multiple generator-discriminator pairs at different levels of the Laplacian pyramid.
- Images are first down sampled at each layer of the pyramid and upsampled again using Conditional GANs (CGANs).
- This process allows the image to gradually refine details and helps in reducing noise and improving clarity.

Due to its ability to generate highly detailed images, LAPGAN is considered a superior approach for photorealistic image generation.

5. Super Resolution GAN (SRGAN)

Super-Resolution GAN (SRGAN) is designed to increase the resolution of low-quality images while preserving details.

Working of SRGAN:

- Uses a deep neural network combined with an adversarial loss function.
- Enhances low-resolution images by adding finer details helps in making them appear sharper and more realistic.
- Helps to reduce common image upscaling errors such as blurriness and pixelation.