Universidad de Oviedo

Universidad de Oviedo EPI GIJÓN

ESCUELA POLITÉCNICA DE INGENIERÍA DE GIJÓN

GRADO EN INGENIERÍA INFORMÁTICA EN TECNOLOGÍAS DE LA INFORMACIÓN

ÁREA DE CIENCIAS DE LA COMPUTACIÓN E INTELIGENCIA ARTIFICIAL

AUTOCOMICS: Comics illustration from textual descriptions using Deep Learning

D. Sánchez Bermúdez, Martín
TUTOR: Dª. Remeseiro López, Beatriz
COTUTORES: D. Vinci, Walter
D. Ménguez Álvarez, Guillermo

FECHA: Julio 2022

Index

Figures Index

Equations Index

1.- Starting hypothesis and scope

Generative models learn to describe a dataset as a collection of random samples drawn from a high-dimensional probability distribution. A generative task is achieved by creating new samples, such as images or text, that do not exist in the training set but maintain the general characteristics, such as coherence and realism, of the original data.

In recent years we have witnessed notable advances in this field, especially related to realistic images (GANs, VAEs, diffusion models) and natural language modeling (GPT, BERT). Large language models such as GPT-3 have also been used for the creation of CLIP [1], a model that is capable of automatically annotate images with auto-generated text (see Figure 1).

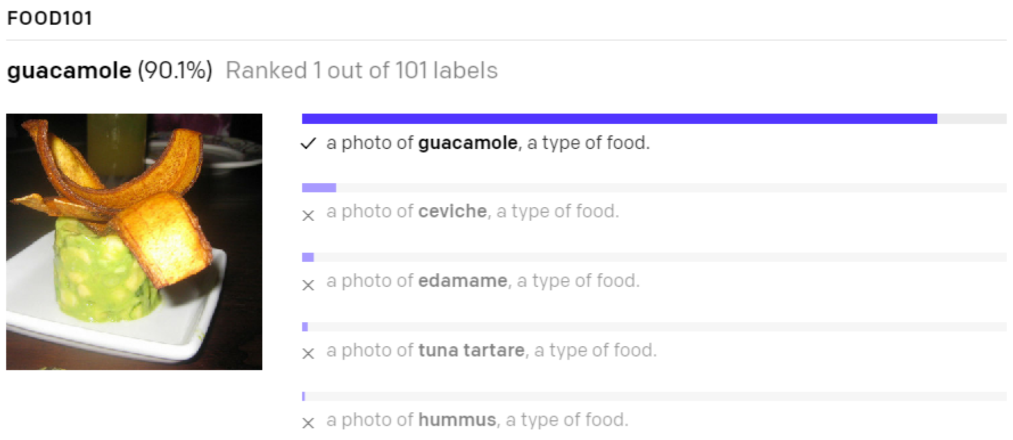

FOOD101
guacamole (90.1%) Ranked 1 out of 101 labels
✓ a photo of guacamole, a type of food.
× a photo of ceviche, a type of food.
× a photo of edamame, a type of food.
× a photo of tuna tartare, a type of food.
× a photo of hummus, a type of food.

Figure 1: Caption prediction of zero-shot CLIP classifier on a food dataset [1].

In this project, we propose to use CLIP to guide a generative model of natural images that will illustrate the pages of a comic from texts that narrate the story. The idea of using CLIP to guide image generation has already been explored by the ML/AI community and is rising a new wave of AI-generated art, in which the artist “creates” clever textual descriptions that are used to drive the generative model and provide an artistically appealing result.

Given the same textual description, the resulting image depends on what kind of visual knowledge and what associations between image and text have been learned by the generative model and CLIP.

The goal of this project is to train a generative model using a comics image dataset to enable CLIP-guided illustration of comics. When run, the project should provide a visual demonstration that this custom training enables the creation of more suitable illustrations than those that can be obtained from community-available models trained with standard datasets such as ImageNet [2].

2.- Objectives and state-of-the-art

2.1.- RELATED WORK

2.1.1.- Convolutional networks

Although creating a neural network of this type is not part of the main tasks carried out during the project, one of these will be used through several steps on Diffusion Models as it will be explained in the next section. Therefore, an understanding and practice of this type of neural network has been required at the beginning of the project.

As a general approach, a Convolutional Neural Network (CNN) [3] is a type of Artificial Neural Network with supervised learning that processes its layers imitating the visual cortex of the human eye to identify different characteristics in the inputs that make it able to identify objects. To do this, a CNN contains several specialized hidden layers with a hierarchy: this means that the first layers can detect lines and curves and are specialized until they reach deeper layers that recognize complex shapes such as a face or the silhouette of an animal.

This type of network takes as entry image pixels. Pixel colors have values between 0 and 255, although they are typically normalized for the network into values of 0 to 1. So, in the case of a color image, the value of the color contained in each channel will be normalized on the scale 0 to 1 for the entry layer of the network (see Figure 2: Pre-processed image channels for CNN entry layer [3].).

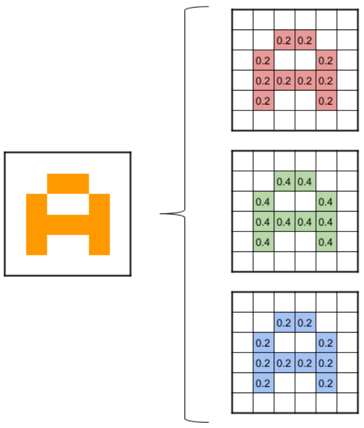


Figure 2: Pre-processed image channels for CNN entry layer [3].

Depending on the type of images, the size of the first layer will change. Having images of 28x28 pixels, if the image is unicolor (greyscale) it will only contain a unique channel and an input size of 784 will be enough for the entry layer. However, if the image is in color, it will consist of 3 channels (for the colors red, green, and blue), which will make a total input size of 2352 for the entry layer.

Once the image fits the network entry layer, the convolutional process starts. This process consists of operating mathematically (scalar product) groups of nearby pixels from the input image against a small matrix called kernel. Having a kernel of size 3×3 pixels, it scrolls through all the input image pixels and generates a new output matrix, which will be a new layer of hidden units (see Figure 4). If the image is in color, the kernel would be 3x3x3 in size, where the last 3 represents the number of channels. Then, results from scalar product of those 3 channels are added plus a bias unit and will make up 1 output as if it was 1 single channel.

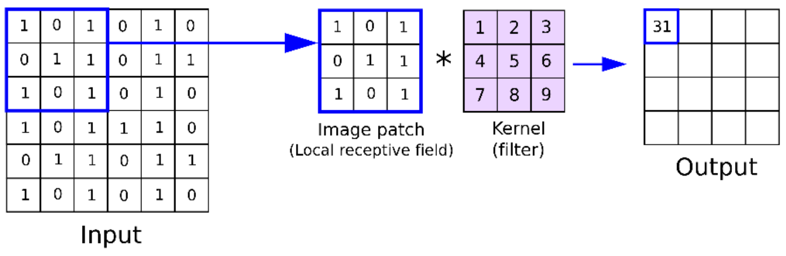


Figure 3: Matrix product on Convolution process [5].

During this process, more than one different kernel will be applied, and we will get as many final matrices as kernels applied (called feature mapping). Each of those new matrices of pixels would represent characteristics of the original image.

On backpropagation, kernel values are adjusted after each iteration. That is an advantage against other traditional neural layers. Taking as example a convolutional layer applied to the input of Figure 4, using a kernel size of 3x3 with 32 different kernels would only require adjusting 288 parameters on that layer for obtaining 32 outputs of size 4x4 (equivalent to 512 output units). On a traditional layer, taking as input the same image of size 6x6 and generating 512 units as output all interconnected, would require adjusting more than 18000 parameters.

There are some ways to change the result we get from the convolutional process that produce a variation on the output size. They are focused on how the kernel is scrolled through the image [4]:

- **Padding**: It aggregates pixels around the image. Their values are usually 0 and, in this case, it is called zero-padding. Thank to this operation we can obtain an output image of the same size as the input one.
- **Stride**: On a convolutional process, the kernel typically scrolls one pixel to right or down each iteration. Each of those scrolls is called *stride*. The size of that stride can be modified to scroll more than one pixel per iteration, obtaining a smaller image that the one we would get with a common convolutional process. It is helpful to reduce the amount of data to process on each layer.

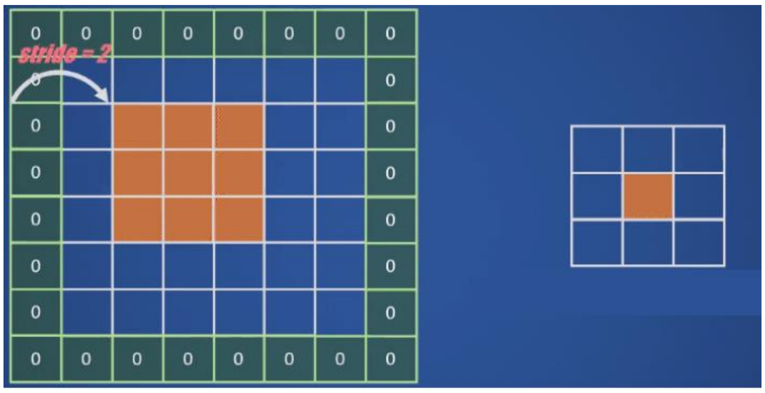


Figure 4: Padding and stride combined [4].

Taking those operations into account, it would be helpful to define a formula that provides us the output size of our convolutional layer. The parameters that modify the output size are input size (width and height), kernel size, padding, and stride. The width and the height of the output image are calculated as:

$$W_{out} = \frac{W_{in} - K + 2P}{S} + 1 \tag{1}$$

$$H_{out} = \frac{H_{in} - K + 2P}{S} + 1 \tag{2}$$

Following these formulas, the output shape of the convolutional layer, having N as number of different kernels applied to the input image, will be: (W_{out}, H_{out}, N).

In addition to the described layers, Convolutional Neural Networks usually implement other components: Activation functions, in CNNs Rectified Linear Units (ReLU) [6] are the choice as activation functions; and Regularization layers [6].

Beginning with ReLU, it is an activation function with the purpose of introducing non-linearity into the model. It takes as input the output of a previous layer and produces as output the input value if it is positive and zero otherwise (see Figure 5 where z is the output).

$$R(z) = \begin{Bmatrix} z & z > 0 \\ 0 & z <= 0 \end{Bmatrix}$$

Figure 5: ReLu activation funcion formula.

Comparing it to another commonly used activation function as Sigmoid, we find some advantages using ReLU:

- ReLU helps preventing the exponential growth in the computation required to operate a neural network as it is easier to calculate output on it.
- It also has a derivative of either 0 for values lower than 0 and 1 for values equal or higher than 0.
- Thanks to its derivative, it prevents the “vanishing gradient” problem, which refers to the tendency for the gradient of a unit to approach zero for high values of the input and is very common when using Sigmoid.

These reasons make ReLU a better option for Convolutional Neural Networks.

In the case of Regularization layers, the reason they are used is because they are a popular way to prevent overfitting. Having a model that tries to obtain lots of features from images, many weights will conform the model. During training, the model can over-adjust to the inputs received in the process. By applying regularization, the model can generalize and improve the performance of the model against data not seen so far.

The aim of using regularization is to make weights to have less impact on loss function, and several methods can be applied to achieve this:

- L2 and L1: they are common types of regularization. Having a loss function, the sum of all weights (squared in the case of L2) of the model multiplied by a constant is added to the calculated error. As the value of the constant is increased, the weight values will be decreased to minimize the loss function.

Being w the value of each weight of the model, N the number of weights of the model, and λ the constant, the new loss function in case of applying L1 regularization would be:

$$Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} w_i \tag{3}$$

In the case of using L2 regularization, the value of weights will be squared to perform the addition:

$$Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} w_i^2 \tag{4}$$

- *Dropout* [6]: it is a layer that prevents the overfitting of the network. It is achievable because the dropout layer acts like a mask ignoring some units on the forward process giving them a value of 0. Because of that, those units do not contribute to the activation of following units in the forward process and their weights are not adjusted on the backward process.
- *Batch Normalization* [8]: in this case, a normalization technique is applied along mini-batches. The formula of Batch Normalization is applied to each unit of a layer. Having z as the output of the unit, m as the mean of all outputs from the unit of the layer to which batch normalization is being applied, and s as the standard deviation, the formula is:

$$z_{out} = \frac{z - m_z}{s_z} \tag{5}$$

On several occasions where convolutional models are built, it is common to perform a classification task with the main features obtained from each image as output after the convolutional process, but in the case of the U-Net model the objective is not the same. After the encoding process where several convolutional blocks are applied to encode the image into feature representations, the decoder up samples and concatenates the features as well as applies convolutional operations. This can be achieved thank to transposed Convolutional Layers.

This type of layer is usually known wrongfully as Deconvolution layer [7]. This is not appropriate because a deconvolution layer reverses the operation of a convolutional layer, so if it is applied to the output of a convolutional layer the original input would be gotten back. In the case of a Transposed convolution, the dimension obtained would be the same as if applying a Deconvolution, but the resulting values would not be the same (see Figure 6).

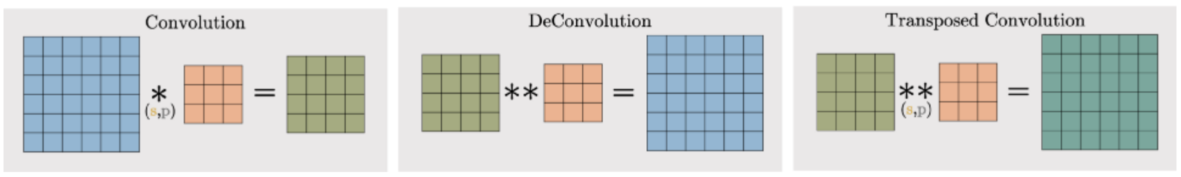


Figure 6: Differences between convolution, Deconvolution, and Transposed Convolution [7].

Transposed Convolutional layers are used for up-sampling. Padding and stride also can modify the output size of the image. There are some steps that can be taken for obtaining an image after Transposed Convolution of the same size than the one used as input on Convolution:

1. Being s the convolution process stride, calculate z as $z = s - 1$.
2. Insert z zeros between rows and columns of the input image.
3. Calculate the p' padding to add on the generated image during step 2 having k and p as kernel and padding used in the convolution process: $p' = k - p - 1$.

4. Perform standard convolution on the resulting image from step 3 with a stride value of 1.

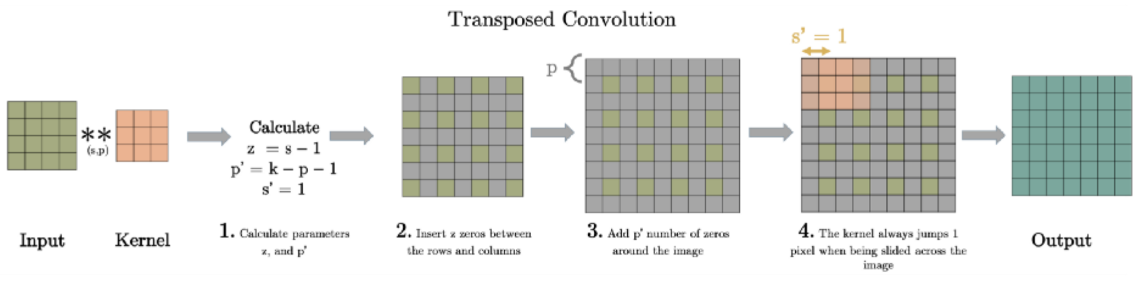

Transposed Convolution
**
(s,p)
Input
Kernel
Calculate
z = s − 1
p' = k − p − 1
s' = 1
1. Calculate parameters z, and p'
2. Insert z zeros between the rows and columns
p
3. Add p' number of zeros around the image
s' = 1
4. The kernel always jumps 1 pixel when being slided across the image
Output

Figure 7: Transposed Convolution process [7].

If the output size objective of the Transposed Convolution process does not need to be the same as the input image on the convolutional process, depending on the hyperparameters used in the transposed process input and output sizes can be calculated. The hyperparameters that modify the output size are input size (width and height), kernel size, padding, and stride. The width and the height of the output image are calculated following the formulas:

$$W_{out} = (W_{in} - 1) \times S + K - 2P \ + 1 \tag{6}$$

$$H_{out} = (H_{in} - 1) \times S + K - 2P \ + 1 \tag{7}$$

2.1.2.- Diffusion Probabilistic Models

Inside all generative models, there is one type that is inspired by non-equilibrium thermodynamics and is called Diffusion probabilistic model [9]. They are based on a Markov chain (a discrete stochastic process where the probability of an event occurring depends only on the immediately preceding event). Along this chain, small amounts of Gaussian noise are added to an image during T steps. These T transitions of the chain are learned for reversing the diffusion process.

Diffusion models are latent variable models where the dimensionality of latent variables x_1, ..., x_T is the same as the original data x_0. The forward process defined as $q(x_{1:T}|x_0)$ is based on the afore mentioned Markov chain. It adds Gaussian noise to the data according to a variance β_1, ..., β_T and follows next formulas where $1 < t \leq T$:

$$q(x_{1:T} \mid x_0) := \prod_{t=1}^{T} q(x_t \mid x_{t-1}), q(x_t \mid x_{t-1}) := \mathcal{N}\left(x_t; \sqrt{1-\beta_t} x_{t-1}, \beta_t \mathbf{I}\right) \qquad (8)$$

Reversing that process makes us able to recreate x_0 from the Gaussian noised data. The reverse process $p_\theta(x_{0:T})$ will be defined by the Markov chain after learning Gaussian transitions. It starts at $p(x_T) = N(x_T; \mathbf{0}, \mathbf{I})$ and follows as:

$$p_\theta(x_{0:T}) := p_\theta(x_T) \prod_{t=1}^{T} p_\theta(x_{t-1} \mid x_t), p_\theta(x_{t-1} \mid x_t) := \mathcal{N}\left(x_{t-1}; \mu_\theta(x_t, t), \sum_\theta (x_t, t)\right) (9)$$

Both forward and reverse processes can be graphically represented as in Figure 8.

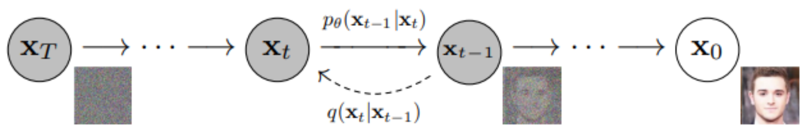

$\mathbf{x}_T$
$\mathbf{x}_t$
$p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$
$\mathbf{x}_{t-1}$
$q(\mathbf{x}_t|\mathbf{x}_{t-1})$
$\mathbf{x}_0$

Figure 8: Process of an image in a Diffusion model [9].

Back into forward process, variances ($\beta_1, \ldots, \beta_T$) can be learned or considered as hyperparameters if they are small enough. Also, the forward process has an important property that allows x_t to be sampled at any step t. Having $\alpha_t := 1 - \beta_t$ and $\overline{\alpha}_t := \prod_{s=1}^{t} \alpha_s$ the formula for obtaining x_t is:

$$q(x_t \mid x_0) := \mathcal{N}\left(x_t; \sqrt{\overline{\alpha}_t} x_0, (1 - \overline{\alpha}_t)\mathbf{I}\right) \tag{10}$$

Diffusion models allow some freedom during implementation. While creating them, variances for the forward process can be chosen. Furthermore, the architecture of the model, as well as the Gaussian distribution parametrization for the reverse process, are eligible during the development process. In this case, as in many examples of implementing a diffusion model, the U-Net model [10]is chosen as model to use in the reverse process, but I will go into depth of this choice later.

Introducing how the loss is calculated in this model, the training is performed by optimizing the variational bound on negative log-likelihood:

$$E[-\log p_\theta(\mathbf{x_0})] \leq E_q\left[-\log \frac{p_\theta(x_{\mathbf{0:T}})}{q(x_{\mathbf{1:T}} \mid x_{\mathbf{0}})}\right] = E_q\left[-\log p(\mathbf{x_T}) - \sum_{t\geq 1} \log \frac{p_\theta(x_{\mathbf{t-1}} \mid x_{\mathbf{t}})}{q(x_{\mathbf{t}} \mid x_{\mathbf{t-1}})}\right] \tag{11}$$

During forward process L_T will be ignored as variances are set as constant values instead of learning them. In the reverse process is where the loss $L_{1:T-1}$ is calculated. From Eq. (2) $\sum_\theta(x_t, t) = \sigma_t^2\mathbf{I}$, we can set $\sigma_t^2 = \beta_t$ so that it does not depend on training as it has a constant value.

Additionally, from the same equation and taking into account the analysis of L_t [1], as we have the value of x_t during the reverse process, we can reparametrize in the mean function, $\mu_\theta(x_t, t)$, having ϵ_θ as the learned function that predicts the noise ϵ given (x_t, t). The simplified mean equation is as follows:

$$\mu_\theta(\mathrm{x_t}, t) = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\overline{\alpha_t}}}\epsilon_\theta(\mathrm{x_t}, t)\right) \tag{12}$$

At this point, the loss function L_{t-1} can be simplified and, using it, the goal of training will be to predict the noise:

$$L_{Simplified} = \mathbb{E}_{t,\mathrm{x}_0,\epsilon}\left[||\epsilon - \epsilon_\theta(\sqrt{\overline{\alpha}_t}\, x_0 + \sqrt{1-\overline{\alpha}_t}\,\epsilon, t)||^2\right] \tag{13}$$

As explained before, the U-Net model [10] is used in the reverse process and here is where it is applied. It will act as the learned function $\epsilon_\theta(x_t, t)$ that predicts the noise in the model.

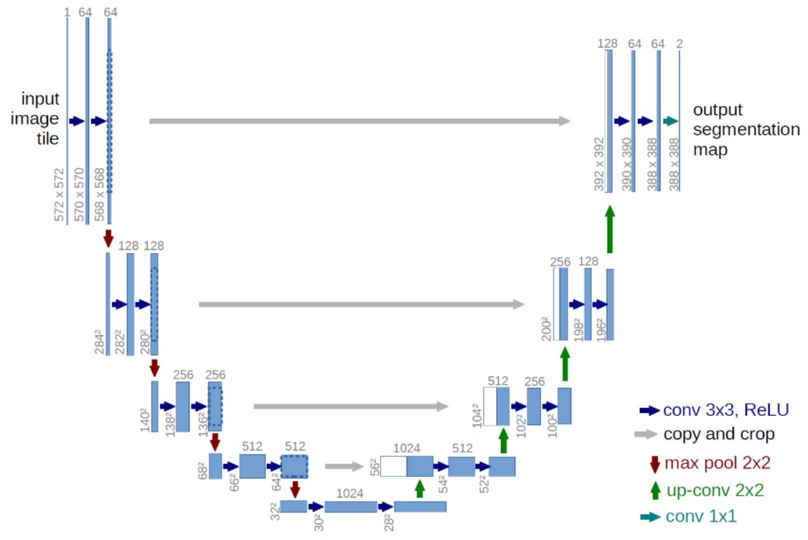


Figure 9: U-Net Convolutional Network for Biomedical Image Segmentation [10].

The name is gotten by its U shape (see Figure 9). Its architecture can be thought as an encoder followed by a decoder. In the first part convolutional blocks are applied followed by max-pooling to encode the image into feature representations at different levels. After that, the decoder up samples and concatenates the features as well as applies convolutional operations. The implementation of U-Net used in the Diffusion Model [9] has some modifications over the original code and adds time-step embeddings *t* as a parameter that will be considered in the process of predicting the noise ϵ that was added to the original image.

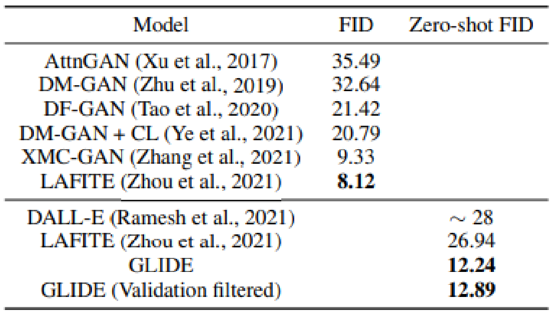
While predicting the noise using U-Net model as the learned function $\epsilon_\theta(x_\mathrm{t}, t)$, it has also the option of passing labels as parameters added to the image and time steps. Enabling this option when creating the U-Net model and using datasets containing pairs of images and labels, we can condition the noise prediction with those labels allowing the model to generate images from noise conditioned by labels on reverse process of the Diffusion model.

2.1.3.- GLIDE

Following what was previously explained about the ability that diffusion models have shown on generating high-quality images, OpenAI released GLIDE. It is presented as a Guided Language to Image Diffusion for Generation and Editing system [11]. Focusing on the text-conditional image generation, GLIDE compares two different guidance strategies: CLIP guidance and classifier-free guidance. The second strategy was preferred by human evaluators when measuring photorealism and caption similarity.

The model was trained with a filtered dataset of approximately 67M text-image pairs [15]. Images of people, violent objects, and some hate symbols were filtered out from the original dataset that contained several hundred million text-image pairs collected from the internet.

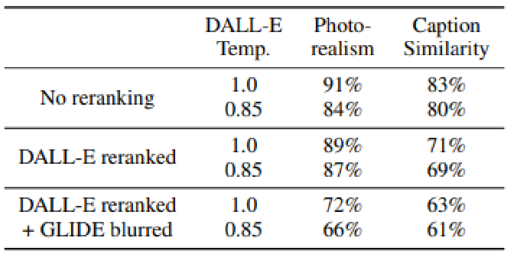
GLIDE was also compared to other text-conditioned image generators using FID (Fréchet Inception Distance), a metric that is commonly used to assess the quality of images created by a generative model obtaining a better quality when FID is lower. MS-COCO [16] dataset (https://cocodataset.org) was used to perform this comparison. Even though GLIDE was not trained explicitly on that dataset, it obtains the best score on the zero-shot FID (see Figure 10).



Model	FID	Zero-shot FID
AttnGAN (Xu et al., 2017)	35.49	
DM-GAN (Zhu et al., 2019)	32.64	
DF-GAN (Tao et al., 2020)	21.42	
DM-GAN + CL (Ye et al., 2021)	20.79	
XMC-GAN (Zhang et al., 2021)	9.33	
LAFITE (Zhou et al., 2021)	**8.12**	
DALL-E (Ramesh et al., 2021)		~ 28
LAFITE (Zhou et al., 2021)		26.94
GLIDE		**12.24**
GLIDE (Validation filtered)		**12.89**

Figure 10: Comparison of different methods using the FID metric on MS-COCO dataset [11].

DALL-E is another system designed for image generation conditioned by text. It was also considered to be taken as an example and as base system for image generation task in this project. Comparisons made on them by human evaluators indicate that GLIDE using classifier-free guidance provides better results than DALL-E, even using CLIP reranking (see Figure 11).



	DALL-E Temp.	Photo-realism	Caption Similarity
No reranking	1.0	91%	83%
	0.85	84%	80%
DALL-E reranked	1.0	89%	71%
	0.85	87%	69%
DALL-E reranked + GLIDE blurred	1.0	72%	63%
	0.85	66%	61%

*Figure 11: Human evaluation results of GLIDE compared to DALL-E [11]**Error! Reference source not found.** using two temperatures for DALL-E (the lower temperature the harder discretization). Percentages are obtained from the difference on the scores given to the models by human evaluators.*

In this way, GLIDE is selected as the most adequate option also considering that it is a smaller model (3.5 billion parameters vs. 12 billion parameters of DALL-E). It will be deepened in this section, as well as used in this project as above mentioned.

Before going into the two guidance strategies used in GLIDE, it is important to consider as background of this model how samples from class-conditional diffusion models could be improved on some occasions with classifier guidance [12]. Having *y* as a target class predicted by a classifier and a class-conditional diffusion model with mean $\mu_\theta(x_t|y)$ and variance $\sum_\theta(x_t|y)$, the model can be perturbed by the gradient of the log probability $\log p_\varphi(y|x_t)$ of the target class, *y*. Being *s* the guidance scale (increasing it was found that improves sample quality but decreases diversity of images), the new mean is given by:

$$\widehat{\mu_\theta}(x_t \mid y) = \mu_\theta(x_t \mid y) + s \cdot \sum_\theta (x_t \mid y) \nabla_{x_t} \log p_\varphi (y \mid x_t) \qquad (14)$$

The first guidance strategy is **Classifier-free guidance** [13]. This technique does not make use of a separate classifier for training the model. The label *y,* which corresponds to the class in the class-conditional diffusion model, is replaced with a null label ∅ with a fixed probability during training. On sampling, the output is extrapolated further in the direction of $\epsilon_\theta(x_\mathrm{t}|y)$ and away from $\epsilon_\theta(x_\mathrm{t}|\emptyset)$. When this strategy is implemented with text prompts, sometimes they are replaced with an empty sequence (also referred to it as ∅) during training. With these premises and having c as caption, the modified prediction of ϵ follows:

$$\widehat{\epsilon_\theta}(\mathrm{x_t} \mid c) = \epsilon_\theta(\mathrm{x_t} \mid y) + s \cdot \left(\epsilon_\theta(\mathrm{x_t} \mid c) - \epsilon_\theta(\mathrm{x_t} \mid \emptyset)\right) \tag{15}$$

This guidance strategy allows the model to create its own knowledge instead of relying on the classifier and simplifies guidance because predicting text is a difficult task for classifiers.

The second guidance strategy is **CLIP Guidance** [11]. CLIP model is based on an image encoder $f(x)$ and a caption encoder$g(c)$. During training, (x, c) pairs are sampled from a dataset and the model optimizes the loss function so that $f(x) \cdot g(c)$ provides high values when image *x* and caption *c* are paired and low values when they are not.

As CLIP provides a score showing how close is an image to a caption, it can be applied to diffusion models replacing the classifier in classifier guidance by CLIP. In this case, during the reverse-process, the mean is perturbed with the gradient of the dot product of image and caption encodings:

$$\widehat{\mu_\theta}(\mathrm{x_t} \mid \mathrm{c}) = \mu_\theta(\mathrm{x_t} \mid \mathrm{c}) + s \cdot \sum_\theta (x_t \mid c) \nabla_{x_t}\left(f(x_t) \cdot g(c)\right) \tag{16}$$

During training, GLIDE model predicts $p(x_{t-1}|x_t, c)$ for each image x_t and the corresponding text caption *c*. Focusing on how to condition image generation by text, it is encoded into a sequence of K tokens, which are sent to the Vaswani transformer [14]. The final token embedding obtained from the transformer is used as class embedding in the ADM model [12], which is adopted by GLIDE. The last layer of token embeddings, which consists of K feature vectors also obtained from the transformer, are projected to the dimensionality of attention layers on the ADM model and concatenated to the attention context of each layer.

As mentioned, GLIDE uses ImageNet ADM model architecture, and it was trained using the same dataset as DALL-E. It also contains an up-sampling diffusion model that was trained to transform 64 x 64 images into 256 x 256.

After the model was initially trained, it was fine-tuned with the objective of generating images unconditionally. The procedure was exactly as pre-training but replacing 20% of text sequences with an empty sequence, providing the model the ability to generate images even when it is not text conditioned.

GLIDE has also the ability to perform Image Inpainting. To achieve that, during fine-tuning, some regions of training examples are randomly erased. Model architecture was modified having four additional input channels: a second set of RGB channels, where the remaining portion of the image is fed to the model; and a mask channel, as additional information from the remaining image. Before fine-tuning, the weights of those channels are set to zero.

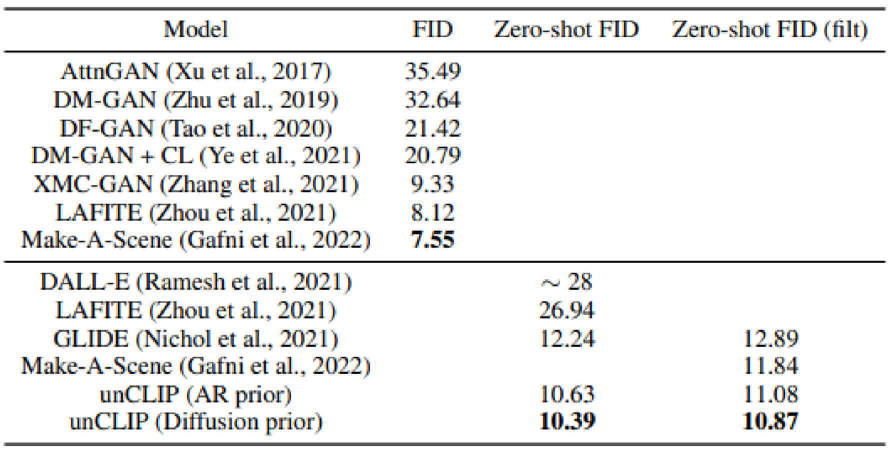
The model that has been used for this project is a small version of GLIDE trained on a filtered dataset that has been released by OpenAI on their GitHub [15]

2.1.4.- DALL-E & DALL-E 2

As seen in the previous section, GLIDE performs better than the initial version of DALL-E developed by OpenAI, and that is why GLIDE was chosen as the base model for this project. However, as of April 13, 2022, the research paper corresponding to the DALL-E 2 version has been published [17].

At that time the project was already started with GLIDE. Even if this was not the case, the API to access DALL-E 2 is not yet available for use since the capabilities and limitations of this model are still being studied, thus preventing it from generating unwanted images such as violence, hate, or even faces of real people or public figures. Nevertheless, some important aspects about this new DALL-E 2 model will be discussed.

As in the GLIDE section, a comparison of the FID metric in the MS-COCO [16] dataset is also presented. The new version of DALL-E, named *unCLIP* because it generates images by inverting the CLIP image encoder, improves the results obtained with DALL-E without having been trained with this dataset either (see Figure 12).



Model	FID	Zero-shot FID	Zero-shot FID (filt)
AttnGAN (Xu et al., 2017)	35.49		
DM-GAN (Zhu et al., 2019)	32.64		
DF-GAN (Tao et al., 2020)	21.42		
DM-GAN + CL (Ye et al., 2021)	20.79		
XMC-GAN (Zhang et al., 2021)	9.33		
LAFITE (Zhou et al., 2021)	8.12		
Make-A-Scene (Gafni et al., 2022)	**7.55**		
DALL-E (Ramesh et al., 2021)		~ 28	
LAFITE (Zhou et al., 2021)		26.94	
GLIDE (Nichol et al., 2021)		12.24	12.89
Make-A-Scene (Gafni et al., 2022)			11.84
unCLIP (AR prior)		10.63	11.08
unCLIP (Diffusion prior)		**10.39**	**10.87**

Figure 12: Results obtained on FID metric using MS-COCO dataset [17].

Regarding the operation of this model, it uses CLIP to obtain a spatial representation of the descriptive text of an encoded image and of the image itself, also encoded, thus performing the CLIP training processes (see Figure 13). At the bottom of the figure, it is showed the text-to-image generation where pairs of images x and captions y form the training dataset. The descriptive text embedding obtained using CLIP z_t is fed to an autoregressive or a diffusion prior $P(z_i \mid y)$ conditioned by the caption, thus producing the embedded image z_i. Finally, this embedded image is used to condition a diffusion decoder $P(x \mid z_i, y)$ conditioned on image embedding and optionally on text captions to produce the final image x. In this second part, the CLIP model will remain frozen to train the prior and the decoder.

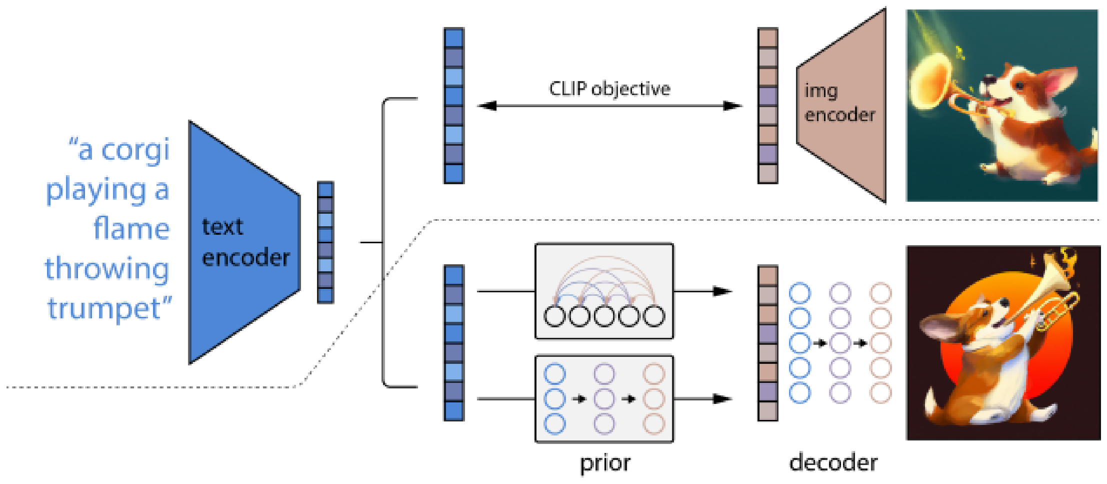


Figure 13: High-level schema of unCLIP, DALL-E 2 [17].

2.1.5.- IMAGEN

Imagen is a text-to-image diffusion model developed by the Google Research Brain Team [18]. The research paper, as well as the website where it is presented, came out on May 23, 2022, so at the time of this project, tests cannot be carried out with it. It is formed by a large transformer language model for text understanding and a diffusion model for good quality image generation. They focus their research on generic large language models that have been pre-trained only with structured text and that show high effectiveness on encoding text for image generation. Creating a larger language model has given Imagen better results than increasing the size of the diffusion model.

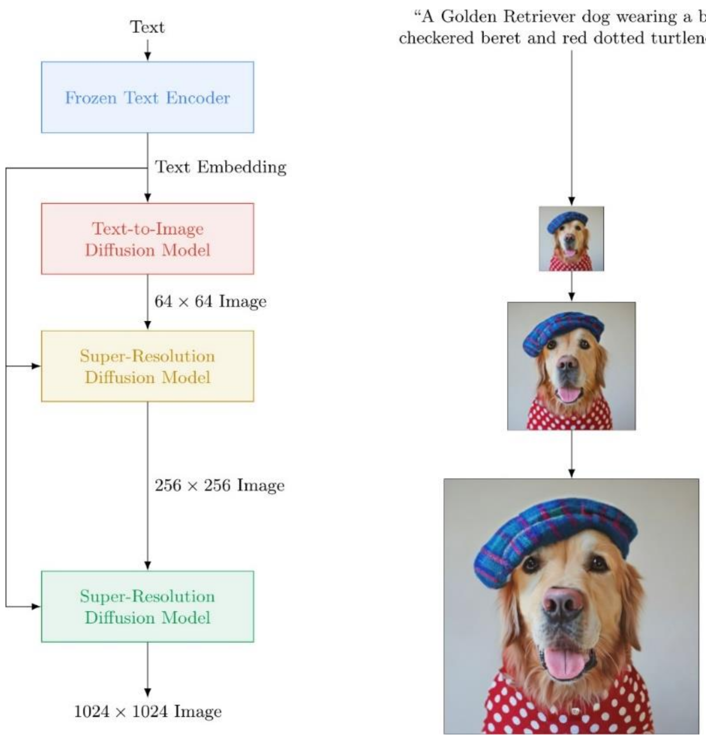
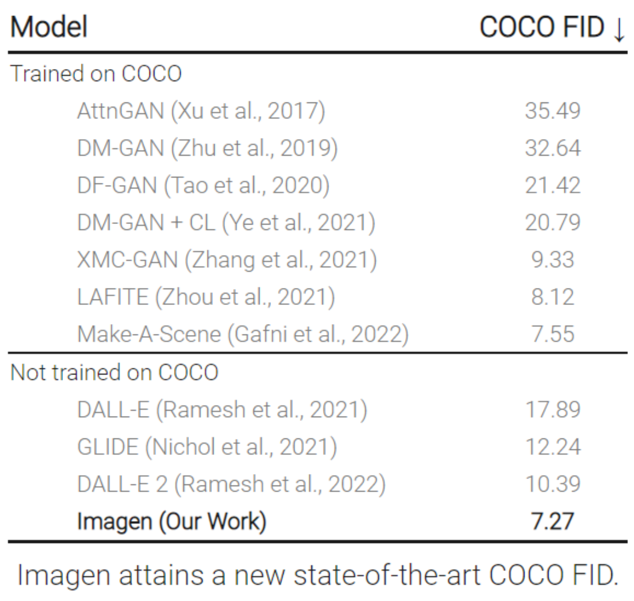

Text
Frozen Text Encoder
Text Embedding
Text-to-Image Diffusion Model
64 × 64 Image
Super-Resolution Diffusion Model
256 × 256 Image
Super-Resolution Diffusion Model
1024 × 1024 Image
"A Golden Retriever dog wearing a blue checkered beret and red dotted turtleneck."

Figure 14: Imagen model schema [18].

Imagen operation begins with the frozen T5-XXL encoder to transform text into embeddings. These text embeddings are then applied in a conditional diffusion model mapping the text embedding onto 64x64 pixel images. Next, two text-conditional-super-resolution diffusion models are used, thus allowing the image to be transformed to a resolution of 1024x1024 (see Figure 14: Imagen model schema [18].).

Imagen also achieves a score of 7.27 on the FID scale in the COCO [16] dataset without having been trained on it, thus surpassing the results obtained by DALL-E 2 and GLIDE (see Figure 15: FID scores on COCO [16] dataset of different text-to-image models [18].).



Model	COCO FID ↓
Trained on COCO	
AttnGAN (Xu et al., 2017)	35.49
DM-GAN (Zhu et al., 2019)	32.64
DF-GAN (Tao et al., 2020)	21.42
DM-GAN + CL (Ye et al., 2021)	20.79
XMC-GAN (Zhang et al., 2021)	9.33
LAFITE (Zhou et al., 2021)	8.12
Make-A-Scene (Gafni et al., 2022)	7.55
Not trained on COCO	
DALL-E (Ramesh et al., 2021)	17.89
GLIDE (Nichol et al., 2021)	12.24
DALL-E 2 (Ramesh et al., 2022)	10.39
Imagen (Our Work)	7.27

Imagen attains a new state-of-the-art COCO FID.

Figure 15: FID scores on COCO [16] dataset of different text-to-image models [18].

In addition, Imagen proposes a benchmark for text-to-image models called DrawBench where it will be compared with different models such as GLIDE, VQ-GAN+CLIP, Latent Diffusion Models, and DALL-E 2 to verify what human raters prefer contrasting the quality of the images as the alignment of the text and the images.

3.- Work Methodology

3.1.- SOFTWARE COMPONENTS

This section includes a description of all the software components used during this project.

3.1.1.- Git

Git is a distributed version control software designed for developing and maintain small and large projects in an individual or collective way. It allows to save a record of all changes in a project, having access to their history, and provides the possibility of recovering a previous state at any time [19].

Some of the advantages of using Git compared to other source control systems are:

- It is fast. Nearly all operations are performed locally. This provides an advantage over other centralized systems that must deal with a greater number of connections to the central server. It is written in C and was built to work on Linux kernel providing speed and performance as a primary goal of the system.
- It is distributed so that although development is carried out centrally, each user has a backup of the code, thus being able to recover it on the central server at any time.
- It is impossible to change the history of the project since Git provides cryptographic integrity to every file and every commit in the project.
- Git provides a "staging area" where commits can be reviewed before committing before completion.

3.1.2.- GitLab

GitLab was born as a Git repository hosting system, i.e., a hosting for projects managed by the Git version system in a single web application. However, inside this software, many other very interesting tools have emerged for programmers and development teams, which involve the entire flow of development and use of applications, tests, etc. [20].

The tools that were used in GitLab for this project are:

- Code version control and organization using the Repository.
- Planning of the project in the Issues tool.
- Upload of documentation of specific parts of the project in the Wiki section.
- Monitoring of work by tutors.

3.1.3.- Google Colaboratory

The choice of this tool has been mainly because Deep Learning models require quite large computational loads for a conventional computer. Thanks to the fact that this tool has GPUs where you can execute your code with greater capacity than those you can find on a conventional computer, Deep Learning models used for this project were trained and executed using Google Colaboratory [21].

Although the free version normally provides an NVIDIA Tesla K80 GPU with 12GB of memory, for a part of this project it was insufficient, specifically in the fine-tuning of the GLIDE model. Although a reduced version of the original model was used, the large number of parameters and the computational requirements of training such a model meant that the Pro version of Google Collaboratory had to be used. Some of the advantages of this version are:

- Priority access to faster GPUs and TPUs.

- More RAM memory and disk memory.
- Continuous code execution for up to 24 hours and possibility to enable background execution after closing the browser.

3.1.4.- PyTorch

PyTorch is a Python package that provides tensor computation like NumPy with strong GPU acceleration and Deep Neural Networks built on tape-based autograd system [22]. Tape-based autograd refers to the use of reverse-mode automatic differentiation, which is a technique used to compute gradients efficiently used by backpropagation.

As mentioned, PyTorch provides tensors, which are also contained in NumPy library. They are generic *n*-dimensional arrays used for numeric computations (see Figure 16).

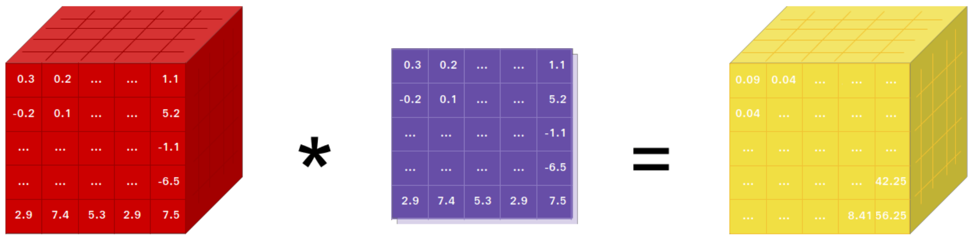


Figure 16: Tensor multiplication [22].

Tensors can be stored on CPU or GPU memory for accelerating computation. PyTorch provides multiple tensor routines as slicing, indexing, math operations, linear algebra, and reductions for improving computations.

This library implements scalable and distributed training that optimizes performance. It exists a rich ecosystem of several libraries and tools that permits extend PyTorch, easing the development in several fields. It is also well supported on major cloud platforms [23], such as Google Colaboratory.

3.1.5.- TensorBoard

TensorBoard is known as a visualization toolkit of TensorFlow [24], which is another platform for machine learning also used as a Python package, but which has not been chosen as main tool for this project.

Even though implementations of the project have been done with PyTorch, this visualization toolkit was used for:

- Tracking metrics such as loss and accuracy.
- Visualizing graphs representing models built.
- Displaying images.
- Storing results from train/test processes.

3.1.6.- Python packages for dataset generation

One of the tasks of this project was generating or creating datasets that were used to feed the Deep Learning models. Some of the packages that were mainly used are:

- NumPy. It is a package for creating and operating multidimensional vectors and matrixes. In this project, NumPy was used for storing images that were received after a HTTP request.
- Pandas. It is the most used Data Science and Computation package. It permits to operate data coming from different types of files such as csv, parquet, etc. as a single and simple dataframe made up of columns and rows. It was used for creating files which paired images and their captions and to read that information from different file sources, clean it, and prepare it as it could be feed to the model.

- OpenCV. As a Computer Vision library, the main purpose of using this package was to resize images that were previously obtained from several sources and to store them with a proper format and naming.
- Requests and BeautifulSoup. The dataset generation task started as a Web Scraping Task, so the use of Requests package for HTTP requests and BeautifulSoup for analyzing web HTML content and get images from it was essential.

3.1.7.- Gradio

Gradio consists of a Python module used for the implementation of Machine Learning Apps. It allows to create a demo for a machine learning model available through a web interface.

It is made up of different components that can be combined in a personalized way depending on the requirements of the application. These applications can be included in Python Notebooks or presented as a web application.

It also permits to host the applications on Huggin Face, which provides servers with their corresponding links to access the applications.

3.2.- PROPOSED METHODOLOGY

3.2.1.- Initial research on Generative Models

The initial main purpose of the project was to train a generative model using a comic dataset for providing the model the ability to generate comics guided by CLIP. Even though some of the steps carried out changed from the original methodology of the project, the main approach was the following.

First, in order to understand the models that I would deal with during the project, before starting with any research or implementation of the same, I dedicated part of the time to inform myself and study about Deep Learning and, more specifically, generative models using *The Deep Learning Book* [26].

In addition to this, I also dedicated this beginning of the project to learn about the CLIP neural network that would be used in the first instance to guide the generative model since CLIP can provide a caption that represents the content of a specific image.

3.2.2.- Familiarization with PyTorch implementing a VAE

To implement, understand, and work on the models of this project, it was essential to familiarize myself with the Python framework that would be used during the whole project.

Since I had not previously used PyTorch to implement neural networks, I had to introduce myself to this framework by implementing by scratch some simple models such as a Variational Autoencoder (see Figure 17). Though it is a very simple generative model compared to those implemented and used later, it was an adequate way to consolidate the knowledge of generative models and PyTorch. Additionally, it was useful to test different image datasets and care about the possible difficulties that will be faced while using them in neural networks.

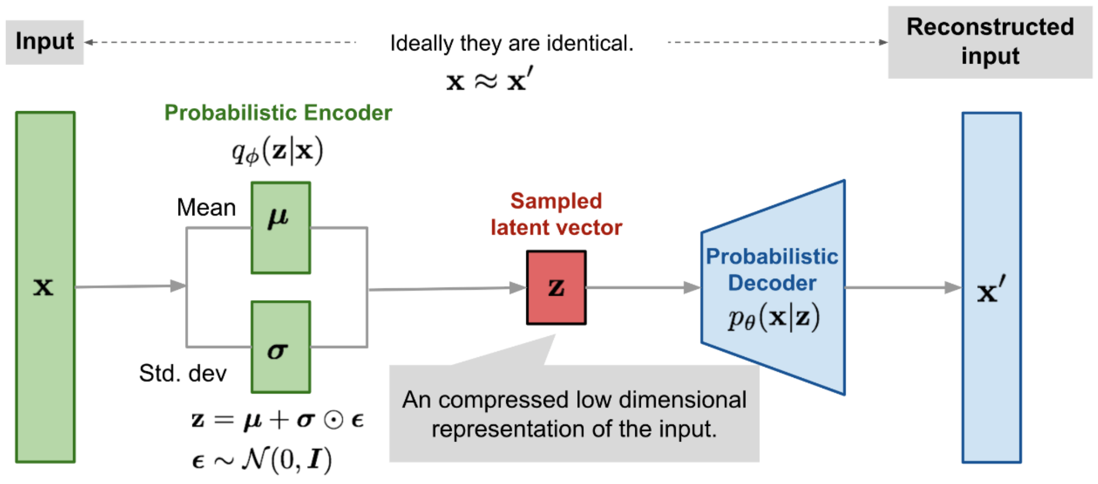

Input
Ideally they are identical.
$\mathbf{x} \approx \mathbf{x}'$
Reconstructed input
Probabilistic Encoder
$q_\phi(\mathbf{z}|\mathbf{x})$
Mean
$\boldsymbol{\mu}$
$\boldsymbol{\sigma}$
Std. dev
$\mathbf{x}$
$\mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}$
$\boldsymbol{\epsilon} \sim \mathcal{N}(0, \boldsymbol{I})$
Sampled latent vector
$\mathbf{z}$
An compressed low dimensional representation of the input.
Probabilistic Decoder
$p_\theta(\mathbf{x}|\mathbf{z})$
$\mathbf{x}'$

Figure 17: Variational Autoencoder schema [29].

After this first approach of studying generative models and familiarizing with the Python framework that I would be using during the project, the next step was to start the research about the state-of-the-art on generative models and datasets that could be used on them.

3.2.3.- Base model election

The focus of this generative models' research was on GLIDE and DALL-E. These two models developed by OpenAI were the ones with better performance at the beginning of this project.

In this field, tools and changes occur in very short periods of time and that is why every so often both technologies and models become outdated. That is why a new version of DALL-E and the Imagen model developed by the Google research team have appeared close to the end of this project. These two models beat on performance the previous ones, as explained in 2.1.-, but they have not been used on the implementations of this project due to its launch date.

After going through GLIDE and DALL-E models, as it is explained before, because of several comparisons between the models the final decision was to use GLIDE as base model for the project. Specifically, the version selected is the one with Classifier-free guidance as it was preferred over CLIP guidance by human evaluators, which compared both models on photorealism and caption similarity.

At this time, the initial idea of using a model that generates images guided by CLIP was replaced by the Classifier-free guidance version of GLIDE due to its better performance.

3.2.4.- Dataset research and generation

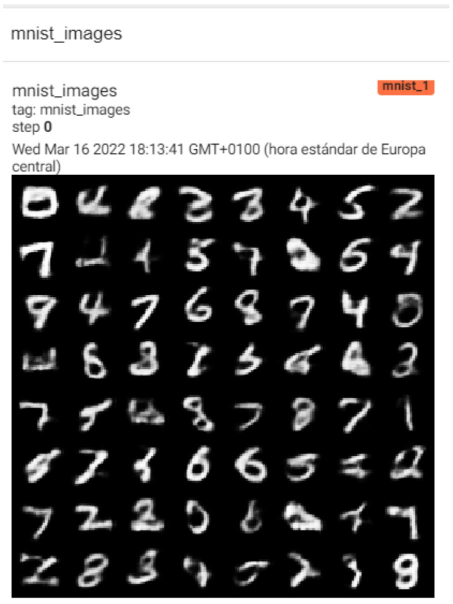
The first intention was to identify some image dataset focused on the style of comics. The best option for the project would be to include a caption of the images but, otherwise, it was planned to obtain this through CLIP.

The main problem found was the lack of image datasets exclusively of comic strips, since only one dataset with images of full pages of action comics could be collected. For this reason, it was proposed to generate my own the dataset that would be used for the project on a specific topic chosen using web-scrapping techniques and filtering large public datasets.

Once the base model was selected, the following step was to understand it and review it in depth for being capable to figure out what to do with the model and how it works.

3.2.5.- Convolutional Layers research and TensorBoard

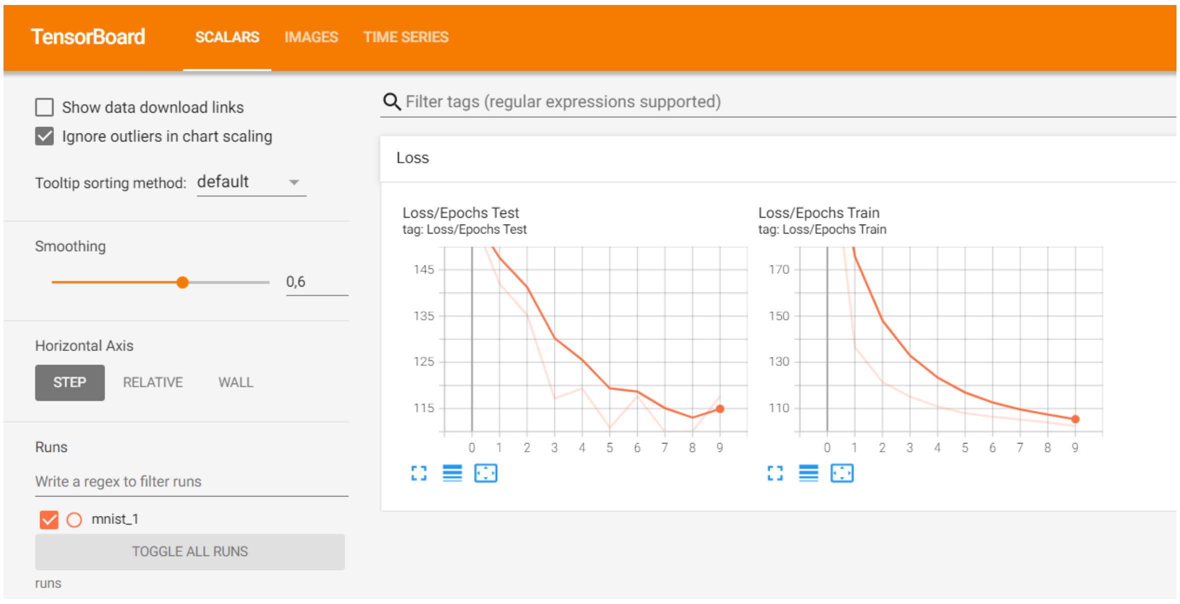
Parallelly, work continued with PyTorch and simple generative models beginning to generate images from these models while applying different configurations of Convolutional Layers, checking their possibilities since they are used in all deep models related to images.


mnist_images

mnist_images
tag: mnist_images
step **0**
Wed Mar 16 2022 18:13:41 GMT+0100 (hora estándar de Europa central)

Figure 18: Images generated using an implemented Variational AutoEncoder

In addition, the collection of metrics during their training was included in these models, as well as the images generated using the TensorBoard tool to be able to visualize them clearly. This tool begins to be implemented at this point in the project since it will be very useful during future training processes for more complex models and will be the metric collection and visualization tool used throughout the project.



Show data download links
✓ Ignore outliers in chart scaling
Tooltip sorting method: default
Smoothing 0,6
Horizontal Axis: STEP RELATIVE WALL
Runs
Write a regex to filter runs
✓ mnist_1
runs

Figure 19: TensorBoard used to visualize metrics during training process of a Variational AutoEncoder.

3.2.6.- GLIDE Fine-Tuning exploration

Once the GLIDE model was fully understood, an implementation of a Fine-Tuning on the model was proposed to try to improve the generation of images on a specific topic.

For achieving this, a Google Collaboratory Notebook was used as a basis. In this Notebook, the Fine-Tuning process was carried out, but the model learned and improved the generation of images from a specific dataset that did not contain captions. The approach then would be to understand how this Fine-Tuning has been implemented and try to modify or create a new Notebook where this Fine-Tuning would be carried out using caption conditioning. The dataset to use for this process would be generated as explained previously.

A detailed schema of the different steps of the Fine-Tuning process from the Notebook was done to have a clear vision of the process. After that, it could be figured out how to apply caption conditioning to the image generation and to the Fine-Tuning. Also, some first versions of datasets were obtained applying Web Scraping from Wikipedia articles of specific topics as different types of animals and fruits.

3.2.7.- Conditional Diffusion Model implementation

As GLIDE is a Diffusion Model, another objective for this research and to get the basis of how the model works was to implement a Diffusion Model by scratch. For this task, we used as reference an annotated PyTorch Paper from Labml.ai [27] with the correspondent code.

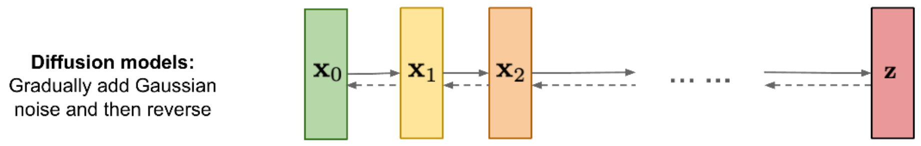


Figure 20: Diffusion model schema [30].

The implementation of the Diffusion Model focused on generating images of hand drawn numbers using the dataset MNIST for training the model. At a first version, the model could generate random numbers. In lately versions of the model, it was conditioned by captions that were numbers, and the model could perform the image generation of those specific numbers.

3.2.8.- GLIDE Fine-Tuning caption conditioned implementation

At the time the Diffusion Model was implemented in a similar but simpler way than the GLIDE model, and the schema of the Fine-Tuning process was clear and detailed, the next step was to implement the GLIDE Fine-Tuning with the image dataset obtained by Web Scraping and caption conditioned.

This task of implementing the Fine-Tuning on GLIDE had lot of importance on the project and it also required different changes in the implementation approach of both the Fine-Tuning itself and the image dataset created.

During the implementation, the theoretical content of the GLIDE model and the Fine-Tuning was documented to check why the performance was not as expected. Larger datasets were used and filtered until the results of the train and test losses of the execution of the Fine-Tuning showed that the generation of images for the dataset had improved.

Also, some hyperparameters of the model like the optimizer and the learning rate were modified in different executions to try how the modifications in the model were being executed as well as the way that images during Train and Test were provided to the model.

3.2.9.- Gradio Application development

At this moment the Fine-Tuning started to perform correctly and improved the GLIDE model results.

To present the results of the project a Colaboratory Notebook with a Gradio application was developed.

Different models that were studied and compared during the research, as well as the implementation of the Diffusion Model and the Fine-Tuning version of GLIDE, were added to this application. The user can select the model that is going to use for image generation and can provide several captions to generate images with the different models.

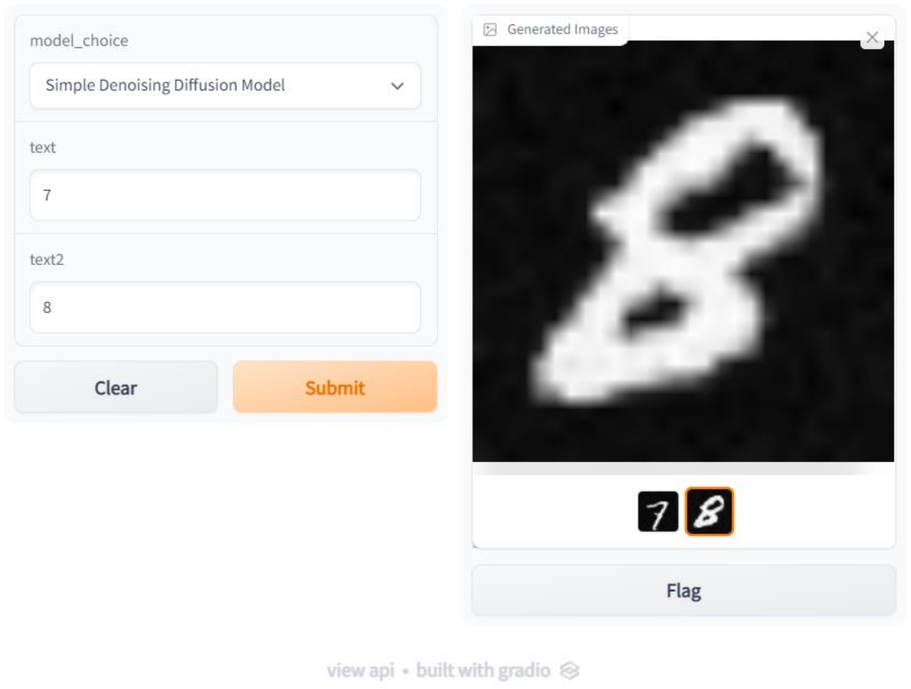

model_choice: Simple Denoising Diffusion Model
text: 7
text2: 8

Figure 21: Initial Sketch of Gradio Application using the Diffusion Model to generate handwritten numbers.

After the model generates the images, they are presented in a Gallery format provided by Gradio that can represent a small comic stripe (see Figure 21).

3.2.10.- Initial objective achieved and potential improvements

In this application several image generation models including the Fine-Tuning GLIDE version can be used to generate images. These images are presented in a certain way as a small comic strip, and their content will depend on what caption is used by the user.

With this resulting application, the initial objective of the project has been achieved.

The project is open to possible improvements on the performance of the fine-tuning or in the development of the Gradio application implementing it as a web service on some cloud provider. This will be deepened in the last section of this report.

4.- Work accomplished and results obtained

All the code, scripts and Python Notebooks mentioned in this section can be found in this GitHub Repository: https://github.com/martinmsb/HP-SCDS_EPI_-Autocomics. Be aware that for any execution of this code the file paths used may need to be changed to the correspondent ones depending on the execution environment.

4.1.- DATASET RESEARCH, GENERATION, AND TESTING

One of the most time-consuming tasks due to its significant importance when performing Fine-Tuning on a deep learning model is the generation of a suitable dataset.

In this case, the image dataset must have a text related to the content of the images. Thus, having descriptive texts about images, both images and captions can be used to perform a Fine-Tuning process to the GLIDE model selected in initial the research process.

Different datasets have been explored for achieving a good result for this task and several scripts for automating a dataset generation process have been developed.

4.1.1.- Initial comic dataset

As an initial proposal for a dataset for the project, a set of comic book images obtained from Kaggle was used [28].

The dataset contained 52,156 images divided into two datasets, one for the training process and the other one for testing. Both datasets contained the images divided into 86 classes and were resized to 288x432 using OpenCV.

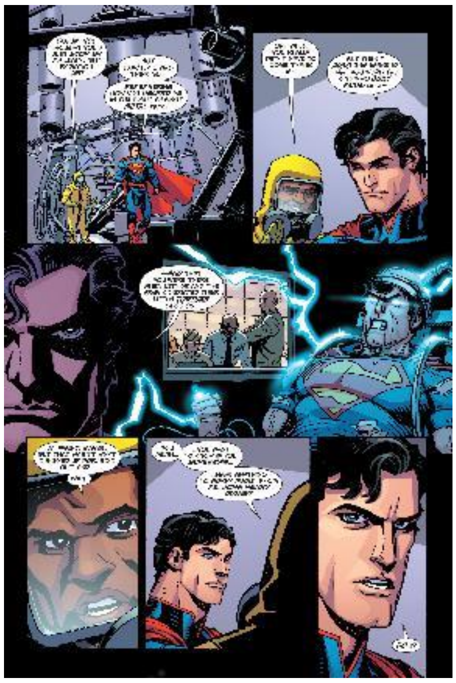


Figure 22: Example of an image from the Kaggle Dataset [28].

Although it might initially seem like an ideal dataset for the project, when exploring how the GLIDE model was implemented and comparing it with the dataset, different complexities were identified:

- First, although the images are divided into classes depending on the type of comic they belong to, they do not contain a description of their content, which is essential for the model.
- The goal of the Fine-Tuning is to make the model able to generate comic images. If the model is Fine-Tuned using those images, the model would learn how to generate images containing a whole page of comic stripes with text that is not readable and mixed with comics covers. Therefore, the model could not be used to generate individual vignettes to be able to illustrate a comic as originally intended.

Having these main drawbacks made it impossible to achieve the project objective using this dataset.

A search was performed looking for datasets that included individual comic strips without text on them and identified with a descriptive caption. That would allow Fine-Tuning to be carried out in a way that would bring the initial objective closer.

This search was not satisfactory either, since no datasets that met these characteristics could be found. Because of that, this first approach of using an already prepared dataset was discarded and the creation of a custom dataset began to be considered.

4.1.2.- Web Scraping for generating dataset

Once the decision was made to generate the dataset autonomously, it was considered as an option to carry out a Web Scrapping job and thus obtain multiple images with their respective descriptions.

As a data source, due to its large number of images and the fact that they always appear with a description, Wikipedia was considered a good option. The implementation of the script was done in a Colaboratoy Notebook (see Figure 23).

The idea of this was to be able to obtain the images of certain specific topics. To achieve this, it was proposed to start from a list with customizable Wikipedia article titles, which could contain any variety of articles. This list would be iterated and each item would be processed as follows:

1) First, the Python Requests module is used to send a GET request and thus obtain the content of the article. Instead of sending the request directly to the page for this article, Wikipedia provides an API named MediaWiki that can be requested by specifying the name of the article and the type of data structure where the HTML content of the article will be placed.

   In this case, the content of each article was obtained within a JSON sending the request to the API.

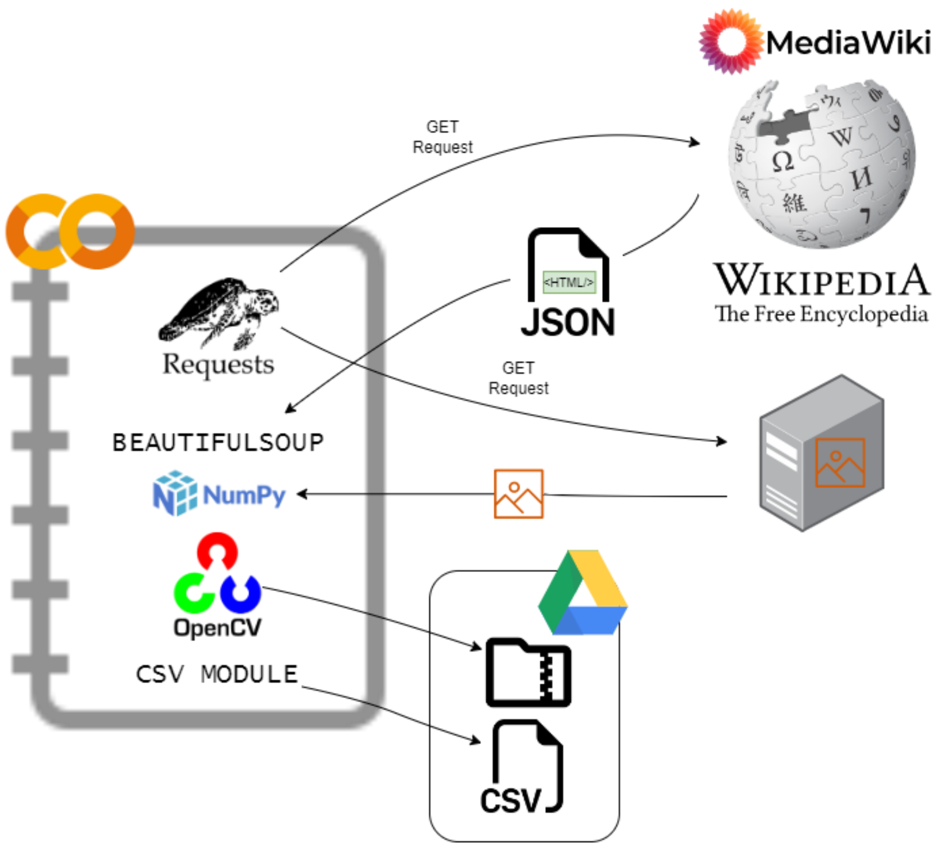


Figure 23: Web Scraping schema for dataset generation.

2) As soon as the JSON containing the HTML is received, the content of the article is selected and the BeautifulSoup module is used to parse it.

   All the images and their respective descriptions on Wikipedia are contained in a "div" element that belongs to the "thumbinner" class. Using the *findAll* function of BeautifulSoup we can get all these elements that contain an image and its respective text.

   At the same time, inside this “div” element, the URL of the image is contained in an "img" element, checking its "src" field the URL is obtained. The same happens with the text, it is inside another "div" and extracting the text included in it we have access to the description of the image.

3) Now that the image URL and its respective text are obtained in variables, we need to download each image to store them.

Again, using the Requests module, a GET request is sent to the URL containing the image. Once we receive the response containing the image, the image is obtained in bytes. To start processing the image and store it, it is checked if the variable where the text of the image is stored is not empty.

4) Using Numpy, the bytes containing the image are converted into an array. In this format, thanks to the OpenCV module, the array of bytes where the image is stored can be converted into an image format. If during this process no error has occurred, the images will be resized so that they all have the same size of 64x64 to use in the GLIDE model.
5) Once the image is in the correct format to be stored as a PNG, it is stored in a folder and given an identifier. Next, the text is added to a CSV document along with the image identifier.
6) After all images and captions of all articles have been processed and stored in a proper way, the script proceeds to store in Google Drive both the CSV with the descriptions of the images and the folder where they were stored compressed in a ZIP file. In this way we have the images and texts saved in such a way that they are easily accessible by the models that are implemented in the Colaboratory Notebooks.

At the same time, the implementation of the Fine-Tuning on the GLIDE model was being carried out, so once it was possible to have a dataset generated with this Web-Scraping process, different Fine-Tuning tests were carried out on the model.

During the different executions, the hyperparameters of the model were varied to see how the dataset responded with different values of the same. Although differences were perceived, the results of the different loss curves were not as expected (see Figure 24).

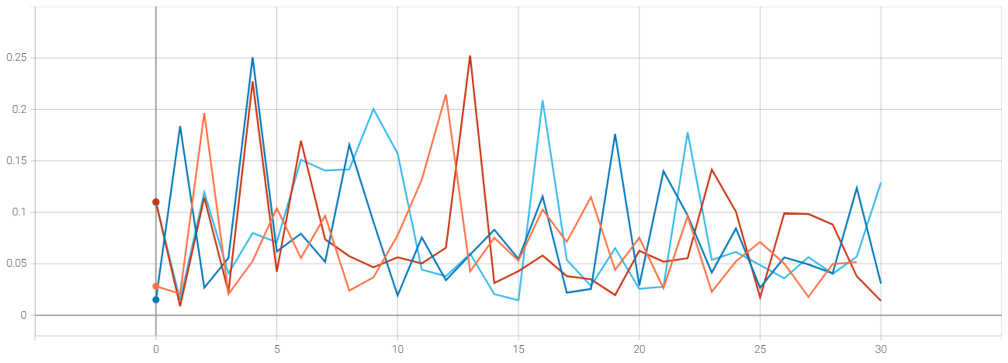


Figure 24: Loss results of different executions of the Fine-Tuning over GLIDE using a Food dataset generated by Web-Scrapping from Wikipedia (Loss value / Epoch).

The loss curve did not decrease as the training epochs progressed but remained constant with jumps to higher values in some of the epochs. In addition, after each epoch, a sampling was carried out with the same descriptive text (see Figure 25).

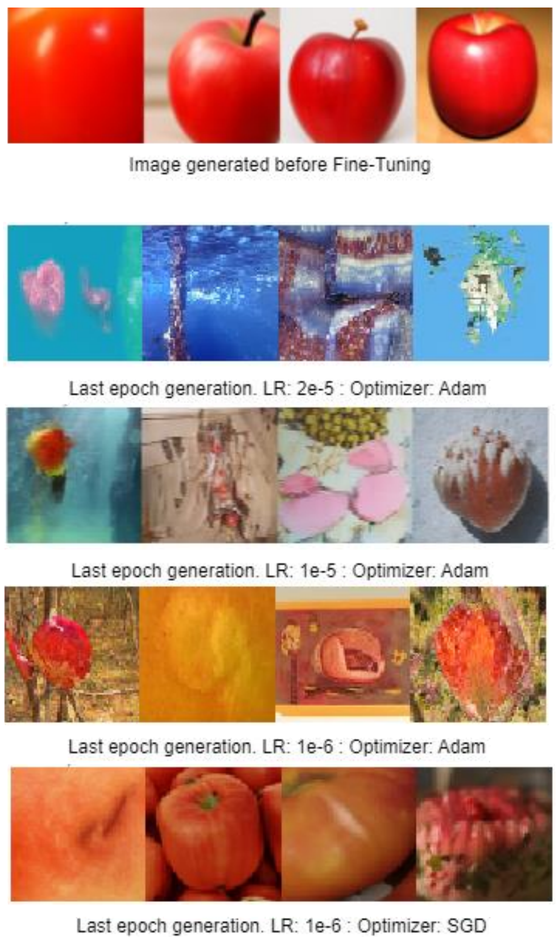


Image generated before Fine-Tuning
Last epoch generation. LR: 2e-5 : Optimizer: Adam
Last epoch generation. LR: 1e-5 : Optimizer: Adam
Last epoch generation. LR: 1e-6 : Optimizer: Adam
Last epoch generation. LR: 1e-6 : Optimizer: SGD

Figure 25: Four images samplig using as caption: "A red apple".

This sampling was also done before carrying out the Fine-Tuning using the same caption. The dataset had been generated using Wikipedia articles with the name of some food to create a specific set of images and thus improve the generation of images by the model in this topic. Since the image dataset had been generated in this way, the caption to use for image sampling was "A red apple". As can be appreciated, the images generated prior to Fine-Tuning are much better than those generated at the end of the process.

The cause can be that the articles from which the dataset has been generated contain images of the corresponding foods, but it may contain images whose caption has that food like a map image, some historical event, etc. Also, it is difficult to generate a big dataset with this practice and a notable number of images are needed to obtain a good result in the Fine-Tuning since the model is trained for a long time with millions of images.

Due to the set of results obtained from this process and the conclusions about them, the generation of images with the Web-Scraping process applied to Wikipedia articles is ruled out.

4.1.3.- Flick dataset

Following the above results, we considered using a larger dataset. The main objective at this moment was to get the Fine-Tuning to be carried out correctly regardless of whether the type of images to be used was not the right one for the final objective of the project.

The dataset proposed was the Flickr dataset. This is a standard benchmark for sentence-based image description, and several versions are available. For this project, a Kaggle version [31] of the dataset was used. Another Colaboratory Notebook was created for implementing a script that prepared that dataset for using it in the Fine-Tuning process.

The dataset processing performed by in the script is as follows:

1) First, the Kaggle API must be installed in the environment, which will allow downloading datasets contained in the platform. From your user in Kaggle you can obtain a JSON file that contains your user and a key to be able to use this API. After executing the necessary commands in the script, you can upload this file identifying yourself and thus having permission to download any dataset from the platform.

2) Once all the dataset images and the corresponding text file with the corresponding descriptions have been downloaded, both images and descriptions must be prepared to use them in the model. First, the file with the captions is loaded into a dataframe using the Pandas module. In this way, the columns will be given the same format as those expected by the Notebook where the Fine-Tuning will be performed. Since the number of images is not a multiple of the batch size of the model, the extra images will be eliminated both from the folder that contains the images and from the dataframe with the captions.

3) Before storing the results of this process, the images must all be the same size, and this does not happen in this dataset. Therefore, using OpenCV again, the images are resized, transforming them into 64x64 pixel images.

4) Finally, a zip file is generated with the folder where the images are stored, and a CSV file is generated with the captions and the identifier of the image that we had also obtained from Kaggle. A copy of them is made to Google Drive and everything would be ready to perform the Fine-Tuning with this new dataset.

The first runs of Fine-Tuning performed with this dataset were reduced from 30 to 5 epochs because the number of images was several times larger than the previous dataset.

Seeing no improvements either in the loss curve or in the generation of images, in addition to slightly varying the hyperparameters since they already had the appropriate values to perform a Fine-Tuning, it was proposed to apply 10 epochs of Fine-Tuning to discard that the performance was low due to the number of times the model had seen these images.

For the images generation corresponding to the end of each training epoch, since the Flickr dataset contains a large number of images where different people appear, this time the caption used was "A man in a red shirt" (see Figure 26).

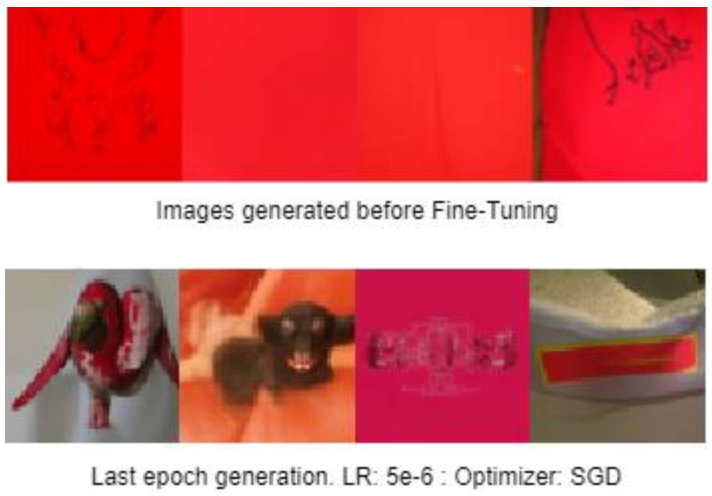

Images generated before Fine-Tuning
Last epoch generation. LR: 5e-6 : Optimizer: SGD

Figure 26: Four image sampling using as caption "A man in a red shirt".

This time, the initial image generated by GLIDE before performing the Fine-Tuning cannot be considered an image according to the provided description either. This is because GLIDE is not trained with any type of images where people appear and that is why the image generated in this example does not resemble what is expected. Regarding the image generated after performing the Fine-Tuning, it does not meet expectations either.

Checking the curves of the Loss function, this time the Fine-Tuning Notebook has been updated in parallel to provide a graph for the Train stage and another for the Test stage (see Figure 27). Again, the loss curve during training does not behave as if the Fine-Tuning was being performed properly and the model was adapting to the new dataset. The loss curve of the Test remains constant and even worsens in the last epochs.

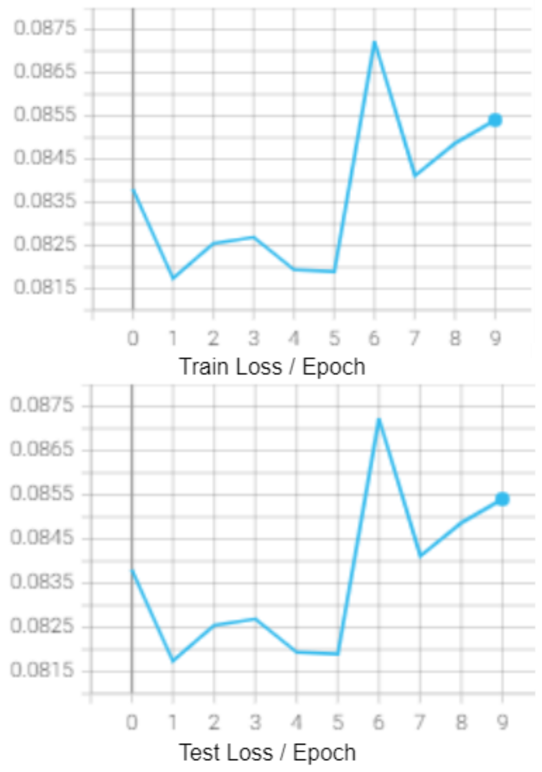


Figure 27: Train and Test Loss curves during an execution of Fine-Tuning over GLIDE using a Flikr dataset.

The executions of Fine-Tuning carried out with this dataset are not being satisfactory either, so this adaptation of the model to new datasets has not yet been achieved. As GLIDE is not trained on images of people and the Flickr dataset is based on images of this style, this may be the reason that the model is not able to adapt to it.

This dataset is then discarded because the results again do not show that the Fine-Tuning has been applied to GLIDE correctly.

4.1.4.- Laion Dataset

Laion is an open dataset of 400 million images [32]. It was built for research and all images and text in the dataset has been filtered using CLIP dropping images and captions whose similarity is below 0.3. That threshold was determined by human evaluators as a good value of image-text-content matching.

The dataset is divided into 32 big parquet files where information about each image is stored. Two different processes have been taken for generating the dataset from the Laion one.

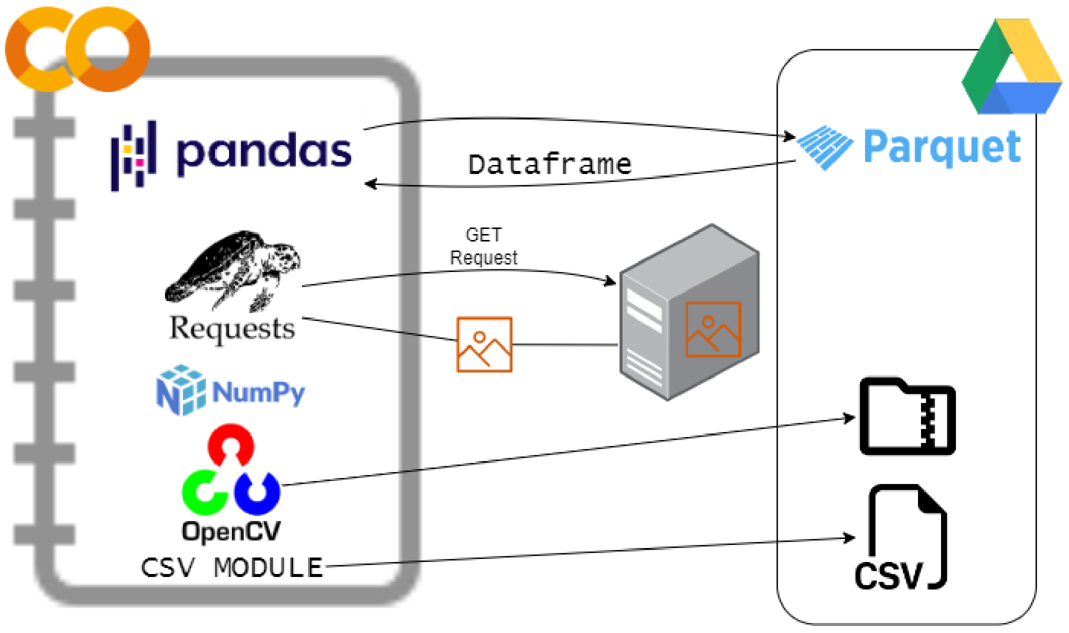


Figure 28: Schema of dataset generation from Laion dataset.

The first preparation of a dataset using Laion image-text pairs was performed using only the first parquet file. A Colaboratory Notebook was developed as a script for preparing the dataset (see Figure 28). The procedure carried out to appropriate the dataset to what was needed for the model was as follows:

1) As starting point, the first of the parquet files was saved in Google Drive to be able to process it using Colaboratory. Thanks to the Pandas module, the content of the file can be saved in a dataframe that allows its processing in a simpler way.

2) The file contains more information than necessary, so several columns were discarded, leaving only the URL through which to obtain the image and the text that describes it.

The objective regarding the size of the dataset was 50,000 pairs of images and text, so the dataframe was reduced to only 65,000 images of the almost 13 million that it contained.

3) Once the dataframe was reduced, the problem that several of these URLs containing the images were not available was identified. To deal with this problem, the entire dataframe was iterated, checking the availability of the images and if they could not be obtained, the corresponding row was dropped.

   For the images that were available, the same procedure was followed as in the section where the dataset was generated using Web Scraping. A GET request is made using the Requests module, the received bytes are configured as an array with Numpy and transformed into a 64x64 image using the OpenCV module, storing the image later in the corresponding folder.

4) As the images had been stored identified by the order of download (the first image had id and name 0, the second 1, etc.) the next step was to reset the index of the dataframe that contained the descriptions since having eliminated rows due to lack of images there were jumps in the value of the index of some rows.

   After adding the index column as the image identifier on the dataframe since it corresponded to the id assigned to the images, the last step was to copy both the dataframe as a CSV file and the folder where the images had been downloaded into a ZIP file to Google Drive. Now the new dataset is ready for use in the Notebook where the GLIDE Fine-Tuning is performed.

Fine-Tuning executions performed with this dataset had a duration of 10 epochs where in each epoch 40,000 images were used for the training phase and 10,000 for the test phase.

In this case, for the generation of images, since the dataset does not contain a specific theme, the caption "A big city at night" was used. This time the image generation prior to Fine-Tuning compared to image generation after that process continues representing the caption (see Figure 29) but visually validating them, the image generated before the Fine-Tuning could be more adjusted to the description. At least the model maintains some of its image generation accuracy and still performs decently.

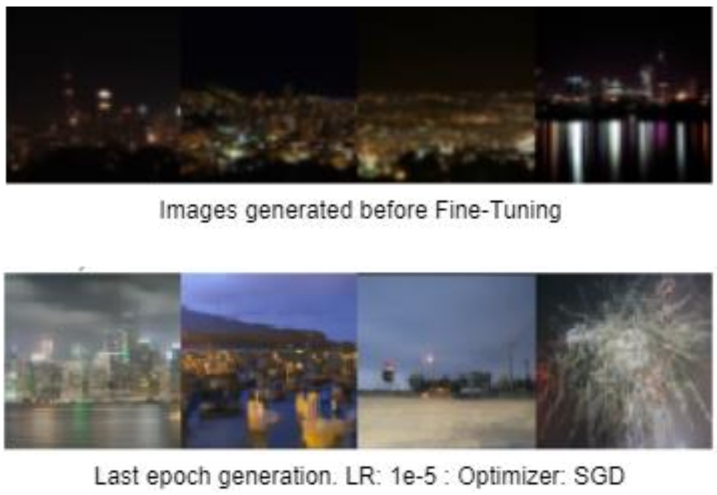

Images generated before Fine-Tuning
Last epoch generation. LR: 1e-5 : Optimizer: SGD

Figure 29: Four image sampling using as caption "A big city at night".

About the loss, not much improvement on it. Loss values during training do not draw the expected descending curve throughout the process. Regarding the test, it seems that it improves initially but after the last epochs the loss value is like the initial one.

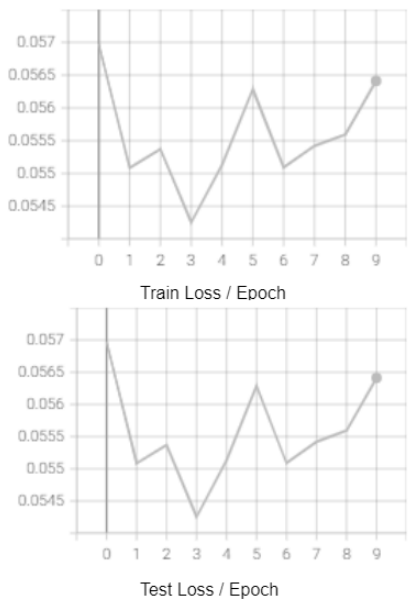


Figure 30: Train and Test Loss curves during an execution of Fine-Tuning over GLIDE using partial Laion dataset. LR: 1e-5; Optimizer: SGD.

As the results with Laion dataset could be improved, another process for generating a dataset from the Laion one was performed but, in this occasion, filtering the images. This filter was implemented in the way that the images selected where only those whose corresponding caption contained certain words. Thereby, the dataset resulting meets the condition of having a large number of images, 50,000 as in the previous example but with images that are more similar to each other and all from a specific theme. In this way, after Fine-Tuning, the model should be able to generate better images from descriptions that belong to the theme of the generated dataset.

To generate this filtered dataset, the process implemented in the Colaboratory Notebook changed slightly (see Figure 31):

1) First, Requests and BeautifulSoup modules were used to obtain the URLs of all available Parquet files, since this time as images would be filtered, it would be necessary to use several files.

2) This function performs the function of downloading each Parquet file as needed. Its content will be saved in a dataframe. It will also generate another dataframe where the identifiers assigned to the selected images will be added along with their descriptions.

3) After this, a download function will be called passing as parameters the words that will be used to filter the images.

   Once the Parquet file has been processed as a dataframe, all its rows will be iterated, checking if the description of the images contains any of the words used for the filter.

4) For each of the images whose description matches the filter made, the following process will be carried out.

   A GET request is made using the Requests module, the received bytes are configured as an array with Numpy and transformed into a 64x64 image using the OpenCV module, storing the image later in the corresponding folder.

If no error has occurred during this process and the image has been stored correctly, a row will be added to the created dataframe to store the pairs of images and descriptions.

This process will be repeated making use of all the necessary Parquet files until reaching 50,000 stored images.

5) One the dataframe of pairs contains the necessary number of rows and all the corresponding images are stored, the dataframe is stored as a CSV file and the images folder will be compressed into a ZIP file. Both files are copied to Google Drive to be able to test the Fine-Tuning process with them.

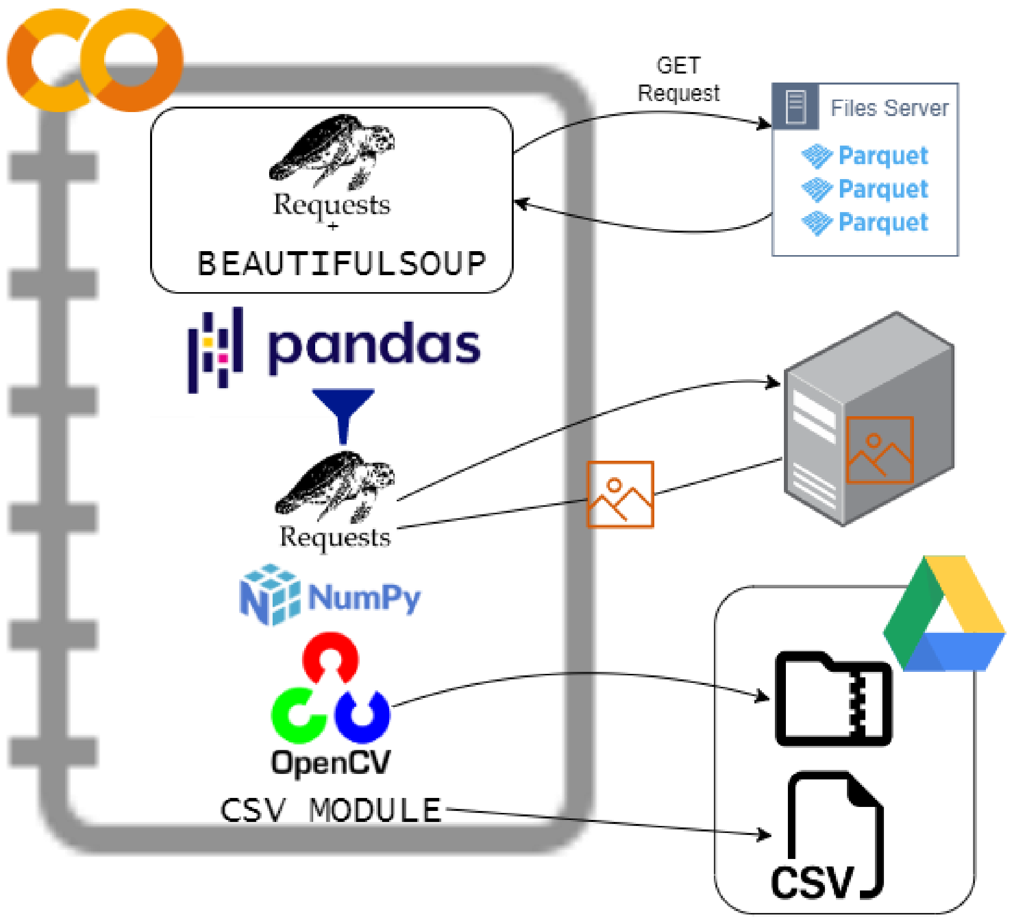


Figure 31: Schema of dataset generation from Laion dataset filtering images.

Fine-Tuning executions performed with this dataset had a duration of 8 epochs because maximum execution time allowed by Colaboratory Pro version is 24 hours and this number of epochs perfectly fixed the time. In each epoch 40,000 images were used for the training phase and 10,000 for the test phase.

The Fine-Tuning could have been maintained for 10 epochs, but this required stopping the execution, saving the partial results, and continuing with it another day so as not to exceed the maximum time allowed. Although this was done in a first run, it was considered not suitable for this stage of the project since the objective was to test different combinations of parameters and different filtered datasets until a good result was obtained from the Fine-Tuning process.

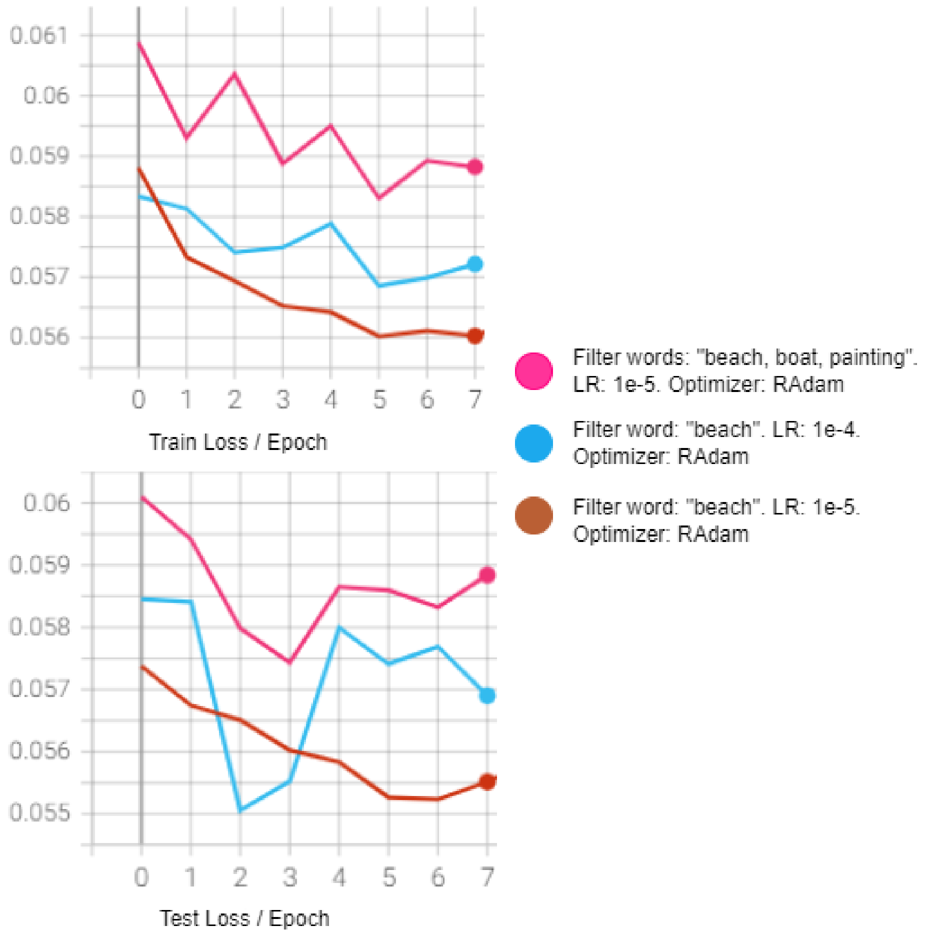


Figure 32: Train and Test Loss curves during an execution of Fine-Tuning over GLIDE using filtered Laion dataset

Figure 32 contains different results of the Fine-Tuning process carried out. This time it can be verified that in all the executions, the loss curve during the training process is clearly descending and improves its value as the executions proceed. In addition to having improved the generated datasets, at the same time small modifications have been made in the way the model was fed with the pairs of images and captions and this has allowed reaching the point where the Fine-Tuning is carried out correctly.

Once the training loss metric obtained during the Fine-Tuning shows the descending curve behavior as expected, it can be presumed that good images had been generated by the model, at least as good as those produced before Fine-Tuning.

The executions that have a loss curve with the lowest values could initially be the ones selected as the final model post Fine-Tuning.

This has not been the case since the execution that uses three words to filter the dataset, one of which is painting, generates better results when it comes to representing an image like what a comic strip would be. If the words boat or beach are also added to the description of the image to be generated, the results are visually at least as good as before this Fine-Tuning process since affirming that they are better would be something subjective (see Figure 33). What can be verified is that the style of the new images is different compared to those generated prior to the Fine-Tuning process and similar between them.

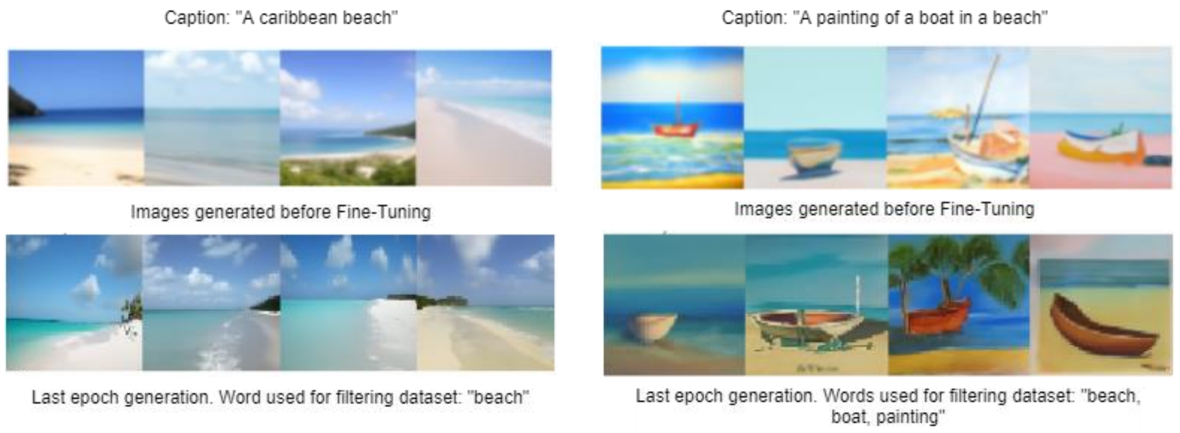

Caption: "A caribbean beach"
Images generated before Fine-Tuning
Last epoch generation. Word used for filtering dataset: "beach"
Caption: "A painting of a boat in a beach"
Images generated before Fine-Tuning
Last epoch generation. Words used for filtering dataset: "beach, boat, painting"

Figure 33: Images generated before and after Fine-Tuning using a filtered dataset from Laion.

After all the datasets tested, a GLIDE Fine-Tuned model has been achieved that improves the initial generation of images for certain text descriptions. Initially, this model after the Fine-Tuning must improve its generation of images for paintings, thus simulating the generation of comic strips and more specifically for those that contain beaches or boats.

Therefore, the final dataset used for the project will be the one generated from the Laion dataset, filtering the images including only those whose text descriptions contain the words "painting, beach, and boat". This dataset could be modified, and words added or deleted to change the topic of the painting, but it is expected that as long as the "painting" filter word is maintained, the generation of images of this style can improve the one provided by GLIDE before the Fine-Tuning process.

4.2.- GLIDE FINE-TUNING IMPLEMENTATION

This part of the project consists of adjusting the previously trained model with millions of images and a long training time. One of the most important things for this process is the choice of the dataset to be used. In the previous section, it has been explained how this selection and generation of the datasets used to perform the Fine-Tuning had been carried out.

This section will explain the different parts of a script that conforms the implementation of Fine-Tuning to the GLIDE model (see Figure 34). This implementation has also been done using Google Colaboratory to easily access the different datasets that have been tested and to have access to virtual machines with GPUs with enough power.

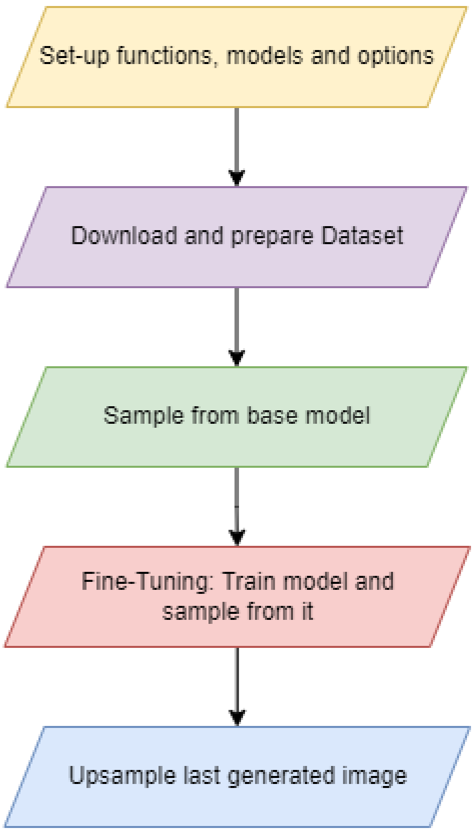

Set-up functions, models and options
Download and prepare Dataset
Sample from base model
Fine-Tuning: Train model and sample from it
Upsample last generated image

Figure 34: Fine-Tuning process schema.

There already exists a Fine-Tuning implementation for the GLIDE model. It only uses some faces pictures and does not implement image generation guided by a caption, but it was used as base implementation for the Fine-Tuning of this project

4.2.1.- Setup functions, models, and options

In the first part of the implementation, after getting a copy of the OpenAI repository with the GLIDE implementation, all the necessary models and modules are imported.

Since the processing will be performed using the GPU provided by the Google Colaboratory virtual machine, the rendering of PyTorch tensors will be performed using CUDA if the virtual machine provides a GPU. This will allow to perform the same tensor operations as on the CPU but taking advantage of the GPU-optimized parallel processing that CUDA toolkit provides. It is transparently implemented by PyTorch as the module provides CUDA Support in the *torch.cuda* package. To achieve this, at the beginning of the implementation, the device to which the models and tensors used throughout the program will be loaded is defined, giving priority to the use of GPU processing through CUDA.

Next, instances of the Text2ImUNet model and the Gaussian Diffusion model are created and initialized with the default options provided by the GLIDE repository.

Similar instances are created for the up-sampler model, which provides an increase in resolution to the generated images. During the Fine-Tuning process this up-sampler model has not been used since when creating the instances that compose the model exceeded the use of GPU memory. For the application implemented with the different image generation models, this up-sampler model has been applied to increase the resolution of the images generated by GLIDE.

Once all these instances have been created and loaded in the GPU, a function is defined that will allow visualizing the images that are generated by scaling the values of the tensors to the range [0, 255] of the RGB scale. Then the images are sent to the CPU and a necessary reshape is performed to the tensors to be able to load them in an array format and use the Image package of the PIL module to display the images.

Finally, the resolution of the images is defined as 64x64 pixels to achieve faster processing and avoid saturation of the GPU memory. The increase value that will be applied when performing the up sampling can also be provided at this point.

4.2.2.- Download and prepare dataset

At this point, the images are downloaded. First, two functions are inherited from the initial implementation of the Fine-Tuning for transforming the images into tensors and another one for downloading images them if the dataset contains URLs instead of a ZIP file with images. In this case, the images are stored in Google Drive, so only the first function will be used to transform them into a tensor that can be sent to the GPU and processed by the model.

Next, the ZIP stored in Google Drive is extracted where the folder with the images is contained. Using the Glob module to extract files or pathnames that meet a certain pattern, all the images contained in the folder that have the .jpg extension are obtained.

These are sorted using the *sorted()* function. In the case of the dataset generated from Laion, as images have been identified with a number and the default order of this function would order them using the image name as string, when loading the images their order would be: 1.jpg, 10.jpg, 100.jpg etc. instead of following the order: 1.jpg, 2.jpg, 3.jpg, etc. To avoid this, an order function is passed as argument to *sorted()* so that it obtains the identification number of the image and compares them after casting them to an integer number to achieve the desired order.

Once the images have been sorted, they are transformed to tensors using the function mentioned above. After this, the first 100 images are shown using the defined function that allows them to be displayed.

Since in the initial implementation image descriptions are not used for Fine-Tuning, the processing of descriptive texts must be implemented from scratch.

First, the CSV file with the captions is loaded into a Pandas dataframe. These descriptions are ordered following the value of the image identifier so that it coincides with the order applied to them. Then, the captions are chosen as a subset of the dataframe and transformed into a Numpy array to later perform a division of Train and Test datasets.

To finish this process, the text and images are divided into a dataset of 40,000 images and descriptions for training and 10,000 for testing, dividing them randomly.

Finally, some of the images and descriptions are shown from random positions of the datasets to verify that everything has been performed correctly and each image has its corresponding text in the same position of its respective data structure.

4.2.3.- Sample from the base model

In this step, the functions used to obtain the images generated by the model will be defined. First, different variables used in the generation of images are defined:

- The number of steps that will be performed in the diffusion model to remove noise from the image.
- The value of guidance scale where higher values move further away from unconditional outputs and lower values move closer to unconditional outputs.
- The up-sampler temperature, which is used for the up-sampling, lowering the value from 0.997 will generate up-sampler images blurrier but with fewer artifacts [34].

Then, an instance of a Gaussian Diffusion model used only for the test stage is defined, and a Classifier-free guidance sampling function was also inherited from the initial implementation of the Fine-Tuning and following the guidance of the GLIDE documentation.

This function will be used during each denoising step for image generation. It will receive the current timestep, the input image corresponding to that step, and the arguments that contain the captions. It will make use of the Text2ImUNet model, which will be Fine-Tuned, to eliminate the noise corresponding to the step that is being performed. This process will consider the guidance scale value and will generate the output image corresponding to the diffusion denoising step.

After this, the function that will be called each new images needs to be generated image is defined. This function will do the following:

- It will first encode the textual description used to guide the generation of the images using a tokenizer provided by CLIP. This Tokenizer treats spaces like part of the tokens and will transform each word into an integer digit. It also considers the position of the words, so a word located at the beginning of the sentence will have a different value than if it was located in another position. Then a mask is added to complete the maximum number of words that can be used.

- Next, a dictionary is generated that will be used as an argument for the model. This dictionary has the following structure:

  {tokens: tensor_of_tokens * half_batchSize + unconditioned_tokens*batch_size,
  mask: tensor_of_mask * half_batchSize + unconditioned_mask*half_batch_size}

  The unconditioned tokens and mask correspond to an empty textual description and are required for image generation using Classifier-Free Guidance.

- Finally, to generate the images, the Gaussian Diffusion evaluation model is used by calling the p_sample_loop() function corresponding to the set of steps for image noise removal in a Diffusion Model.

  The Classifier-Free guidance function is passed as parameter. This function makes use of the model that will later be Fine-Tuned, the size of the images to be generated (4 images with 3 channels for RGB and 64x64 pixels), the selected device, and the generated arguments with the dictionary of tokens.

After the function is defined, a batch of 4 images is generated for testing the model and to have a reference of image generation before applying the Fine-Tuning.

4.2.4.- Fine-Tuning: Train model and sample from it

The first thing that will be done in this section of the code is to initialize TensorBoard where all the loss results and the images generated each epoch will be saved. In addition, an initial image will be stored to be able to compare later with those generated after Fine-Tuning. Then, the code corresponding to the Fine-Tuning will begin:

- First, the model gradients are set to 0, all the hypermeters are defined, and a value will be given to them before being assigned to the model. The path where the images will be saved will also be established. Images can be downloaded and checked in that temporary path of the assigned virtual machine during execution but will also be stored using TensorBoard for later retrieval.
- After that, another function that will generate the tokens corresponding to the captions is defined. The identifier of the first image of the batch and the array with the captions will be passed as parameters. This time, three sets of text tokens will be generated from the captions and a set of tokens will be used as unconditional. This practice is recommended in the GLIDE paper [15] since the model itself has been Fine-Tuned after the initial training. In this way, the model improves generation of text-conditioned outputs but it still is capable of generating images unconditionally.
- Arrays are instantiated to store the loss and show it in a plot after each batch and the number of epochs to Fine-Tune the model is defined. From here the Fine-Tuning process begins. A loop is started to repeat the following process for each epoch:
    1) The value of the sum of the loss obtained after each batch of both Train and Test is reset to 0 and the generated image counter is reset to 0.

The training mode is activated in the Text2ImUNet model so that layers such as Dropout, BatchNorm, etc. act accordingly.

2) Then is the training process. The *chunked* function is used to split the images into batch size chunks, in this case 4 images. For each group of images, a random tensor of the size of the batch size and with values between 0 and the number of betas of the Diffusion model is generated. Another random tensor that will act as noise to be applied to the images will be generated with batch size and values following a normal distribution.

   Next, the diffusion process will be applied to the images of the corresponding batch, adding the noise generated during the number of defined steps. Then, the arguments corresponding to the token dictionary are generated using the function defined above and the captions corresponding to the batch images.

   With the diffused data, this is sent to the model in charge of removing noise from the image along with the number of steps that noise has been applied and the arguments that contain the tokens of the textual descriptions. After receiving the output, the loss is calculated using both output and the noise applied to the image. Then *backward* function is called on the loss to store the gradients for all tensors in the model, and then the optimizer iterates through all the parameters to update their value using the stored gradients.

   Finally, this process is repeated until processing each image and caption of the training set, accumulating the loss of each batch in a variable to obtain the average of the epoch and store it in TensorBoard.

3) Now the testing process begins. The evaluation mode is activated in the Text2ImUNet model. The same process is done as in the train process, except that this time the update of the gradients will not be applied to the model parameters.

   Finally, the calculated loss will be saved in TensorBoard together with an image generated from the model once this first epoch of the Fine-Tuning has been processed.

4.2.5.- Up-sample last generated image

In this last section of the Notebook, we will proceed to use the GLIDE model that increases the resolution of the images applying it to the last image generated. Nevertheless, during the various Fine-Tuning processes carried out along this project, this part of the script has not been executed to avoid loading the corresponding model in the GPU. This allows to save GPU memory since sometimes the Fine-Tuning process stopped due to out of memory problems.

At the end of the script, the results of the Fine-Tuning of TensorBoard are saved to Google Drive to restore them later and the final state of the model after the Fine-Tuning. The last cell of the Notebook will allow to show previous results of the Fine-Tuning saved on Google Drive to TensorBoard.

4.3.- CONDITIONAL DIFFUSION MODEL IMPLEMENTATION BY SCRATCH

To understand the GLIDE model in depth and implement Fine-Tuning correctly, the creation of a Diffusion Model by scratch was proposed, since GLIDE is based on this type of Deep Learning model. This implementation will follow the theoretical content of 2.1.2.- and will be developed over a Google Colaboratory Notebook.

To ease the implementation, the paper *Denoising diffusion probabilistic models [9]* and an annotated implementation guide created by Labml [27] were used as an example of implementation. This guide does not include guidance of the model by label or by text, but it will be added in the implementation to make the resulting model more like GLIDE.

4.3.1.- Dataset selection and device

The dataset selected to carry out the tests of the model to be implemented was MNIST, formed by handwritten digits from 0 to 9. This dataset is divided respectively into Train and Test sets and transformed into tensors.

Again, the device where the model will be executed is chosen. As in the case of GLIDE, priority is given to execution on the GPU using CUDA.

4.3.2.- Model Implementation

First, an instance of the U-Net model will be declared. This model will be in charge of predicting the noise that has been applied to the different images. The instance will have to be declared applying as hyperparameters:

- The size of the images, in this case 28x28 pixels.
- The number of input channels that will be 1 since the images are grayscale.
- The number of output channels that will also be 1.

- The number of classes that will serve as guidance for noise prediction. Since an implementation of the model whose generation of images is guided by labels will be proposed, the value 10 will be defined here, which is the number of different digits that will be found in the model.

Next, the gather function that will be used in the model is defined. This will work as follows:

Having the function *gather*(a, b) and both a and b two one-dimensional tensors, a tensor of size [len(b), 1, 1, 1] is generated and filled with the values a[b[i]]. Usage example:

b = ([2, 5, 5, 4, 3])

a = ([1.0000e-04, 4.0800e-03, 8.0600e-03, 1.2040e-02, 1.6020e-02, 2.0000e-02])

Returned tensor: ([[[[0.0081]]], [[[0.0200]]], [[[0.0200]]], [[[0.0160]]], [[[0.0120]]]])

Having the final tensor with this size the operations with it will be performed elementwise.

At this point, the *DenoiseDiffusion* class is created where the model will be created, and it will contain the necessary functions for its operation:

- In the model initialization function, the model to be used for the reverse process where the noise is removed is assigned. The β_1, ..., β_T and $\alpha_1, \dots, \alpha_t$ tensors are also defined, as well as the number of steps to apply, $\overline{\alpha}_t$ values and σ_t^2, these last values will have the same value as β_t explained in the theoretical section.

- The functions *q_xt_x0* and *q_sample* will be in charge of carrying out the forward process where noise is added to the image. *q_sample* is in charge of receiving the batch images as parameters, the steps to apply to each image and the noise to apply. The function *q_xt_x0* will be used to carry out the operations corresponding to $qxt\square x_0\square \coloneqq \mathcal{N}\left(x_t; \sqrt{\bar{\alpha}_t}x_0, (1-\bar{\alpha}_t)\mathbf{I}\right)$ (10)for sampling x_t at any timestep. With the result of this function, the image resulting from the forwarding process will be generated with the noise added to the corresponding timesteps.

- The function p_sample will be in charge of performing a step of the reverse process, eliminating the noise corresponding to *t* step following the $\mu_\theta(\mathrm{x_t}, t) = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(\mathrm{x_t}, t)\right)$ (12) and using the result for applying $p_\theta(x_{t-1} \mid x_t)$ formula in $p_\theta(x_{0:T}) \coloneqq p_\theta(x_T)\prod_{\mathrm{t=1}}^{T} p_\theta(x_{\mathrm{t\text{-}1}} \mid x_\mathrm{t}), p_\theta(x_{t-1} \mid x_t) \coloneqq \mathcal{N}(x_{\mathrm{t\text{-}1}}; \mu_\theta(x_\mathrm{t}, t), \Sigma_\theta(x_t, t))(9)$. The necessary variables for the formulas are defined, the U-Net model is used for predicting the applied noise initially following $\epsilon_\theta(x_\mathrm{t}, t)$, but in this implementation it also uses as guidance the label of the original image y following $\epsilon_\theta(x_\mathrm{t}|y, t)$. Once the afore mentioned equations are calculated, the image resulting of the *t* step is returned.
- Finally, the *loss* function is defined. It will be calculated as the simplified loss in $L_{Simplified} = \mathbb{E}_{t,\mathrm{x_0},\epsilon}[||\epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_\mathrm{t}}\, x_0 + \sqrt{1 - \bar{\alpha}_\mathrm{t}}\, \epsilon, t)||^2]$ (13). First, a tensor with a number of steps to apply to each image of the batch is defined. Then *q_sample* function is used to apply the noise to the images during the corresponding steps. After that, an attempt is made to predict the applied noise using the U-Net model passing as parameters the batch of noisy images, the steps applied to each one, and the labels corresponding to the digit they represent. As last step, Mean Squared error between the applied and predicted noise is calculated.

4.3.3.- Training and sampling code

When the Diffusion Model is already implemented, a training and sampling class is defined as *Configs*:

- First, the device where the model and images will be loaded is assigned as well as the class previously created for the Diffusion Model and the U-Net that will be used in.

  The parameters for the images are also declared: number of channels and size; and the hyperparameters of the model: number of time steps, batch size, number of examples to generate, learning rate, optimizer, and epochs. In the *init* method, the previously declared U-Net model and the *DenoiseDiffusion* class are initialized with the previously assigned hyperparameters. The Train data corresponding to the division of the dataset is also declared.

- Then, the *sample* function responsible for generating images from the model is defined. Using the evaluation mode of the model, a batch of noisy images is created from which the images will be generated. Also, a batch of random labels between 0 and 9 will also be generated to guidance the noise removement on the images. The *p_sample* function of the model will be used as many times as time steps have been declared. Finally, the images will be saved both in TensorBoard and in PNG format, assigning the corresponding label to each image, thus being able to check if the results are close to what is expected.

- A *train* function is also defined that is equivalent to the process of a training epoch. In it, the set of training images and labels is iterated, dividing the set according to the batch size, and they are loaded into the corresponding device. The gradients are set to zero and the loss function of the Diffusion Model is called. Then backward function is called on the loss to store the gradients for all tensors in the model, and then the optimizer iterates through all the parameters to update their value using the stored gradients. This function returns the loss result of the epoch.

- To finalize the implementation of this class, the run function is defined, which will make the calls to the training and sampling functions in each epoch. After finishing the training of the model, the final state of the model will be saved.

4.3.4.- Training execution and results stored

A function is defined that will initialize the model and start training it. TensorBoard is also initialized where all the results of the training process will be saved.

To finish the script, the function in charge of starting the training is executed and once it is done, the results will be copied to Google Drive. Finally, the results saved from TensorBoard during training are displayed (see Figure 35).

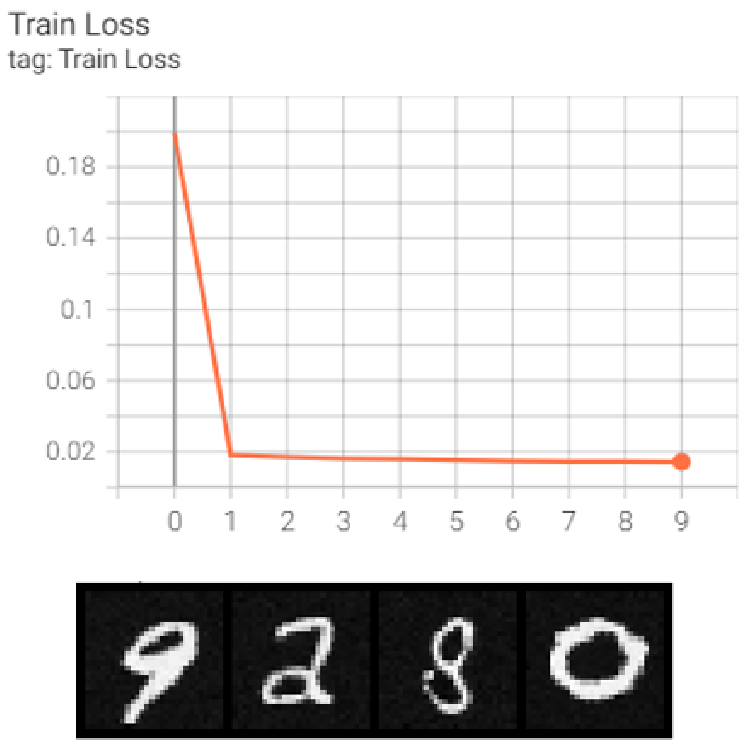


Figure 35: Loss values during the Training process and image generation for labels 9, 2, 8 and 0.

4.4.- Gradio Application

In order to show the results of all the models studied and implemented during the project, it was proposed to make a simple application using the Python Gradio module. This module allows to generate interfaces for machine learning projects in a very simple way and integrate them into a Notebook or a webpage. In this case, as has been said in previous sections, it is presented in a Google Colaboratory Notebook, although when it is executed, it generates a URL through which it is possible to interact with the same interface.

To implement this Notebook, all the models to be used are instantiated: Diffusion Model implemented by scratch for the MNIST dataset, GLIDE with its version of Classifier-free guidance Fine-Tuned in this project, the version of the DALL E Mini model as well of an improvement of this called DALL·E Mega and finally to CLIP Guided Diffusion Model.

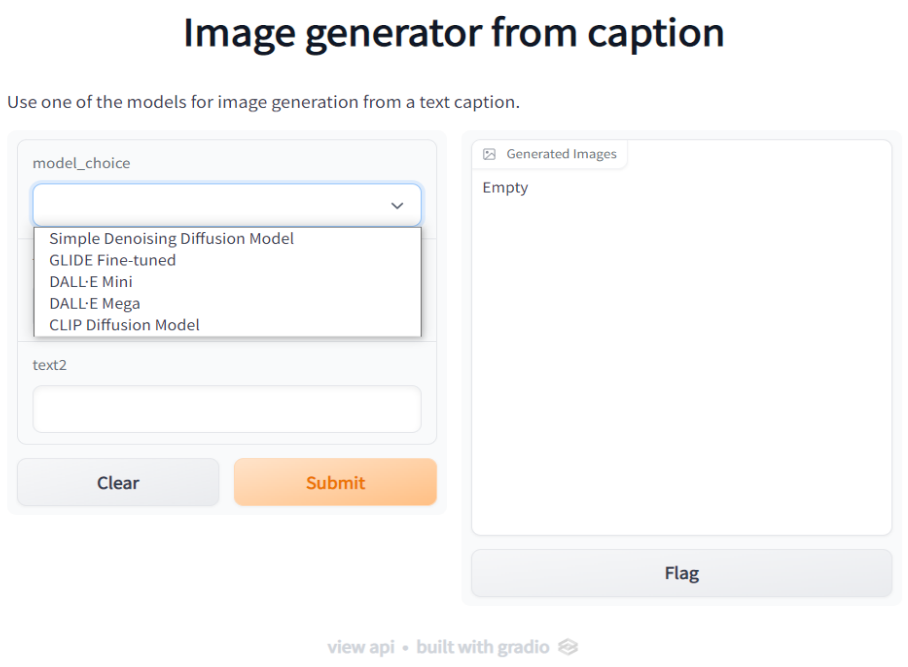


Image generator from caption

Use one of the models for image generation from a text caption.

model_choice

Simple Denoising Diffusion Model
GLIDE Fine-tuned
DALL·E Mini
DALL·E Mega
CLIP Diffusion Model

text2

Figure 36: Gradio application with a Dropdown for selecting the model to use.

Once all these models have been implemented and initialized, the Gradio interface is defined. This will have a Dropdown to select the model to use, two input elements to enter the text captions from which the generation of images will be guided, and for the models' output, an element called Gallery that allows images to be displayed as a gallery format (see Figure 36: Gradio application with a Dropdown for selecting the model to use.). It will simulate the generation of a small comic strip.

As this Gallery element only supports images contained in a List with specific formats: Numpy array, PIL.Image or String, the format of the generated images produced by the models must be adjusted so that the behavior of the models and the generated images fit the requirements of the element.

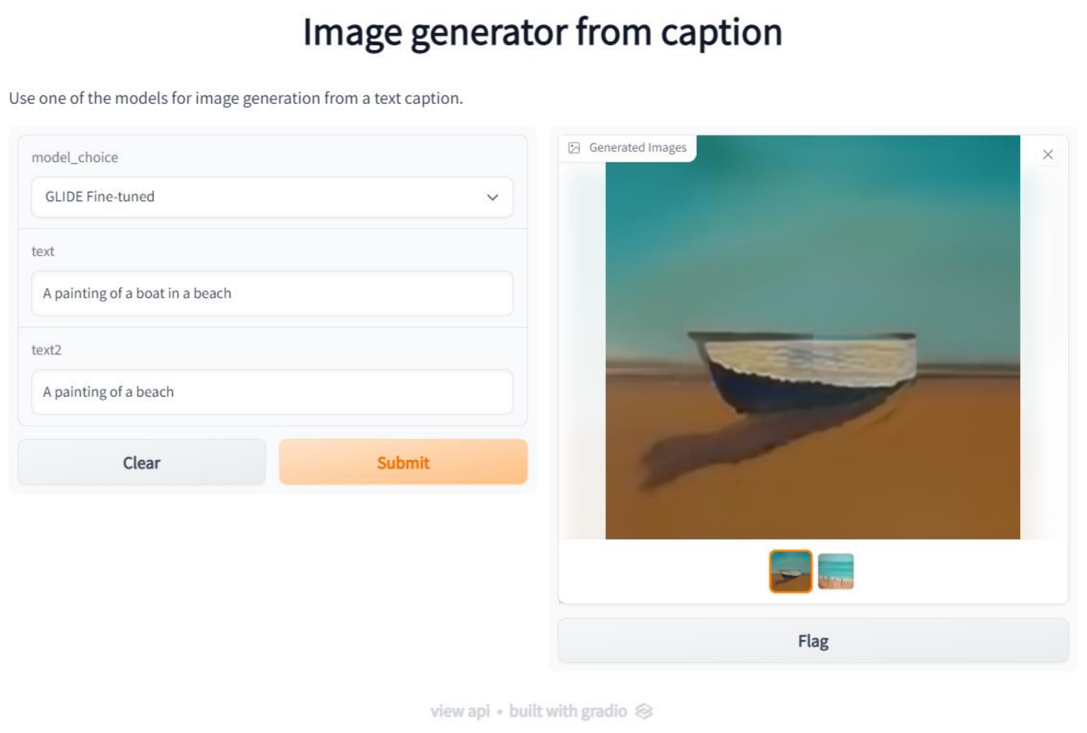


Image generator from caption

Use one of the models for image generation from a text caption.

model_choice: GLIDE Fine-tuned

text: A painting of a boat in a beach

text2: A painting of a beach

Figure 37: Images generated on Gradio Application using GLIDE Fine-Tuned model.

The logic of the application is as follows:

1) Once the user has chosen the model and written the text captions, pressing the send button will execute a function that Gradio assigns to the interface.

2) In this function, depending on the selected model, another function corresponding to it will be executed.

3) Now the text captions are sent to the selected model either together or separately depending on how image generation is implemented in it. Once the model has generated the images, they are returned to the correspondent model function.

4) The pixels values scale will be adjusted if necessary and the shape of the image will be modified if it does not have an adequate one. The images will also be transformed to a format allowed by the Gradio Gallery element in case they are not and at the end images in a valid format are added to a list.

5) Finally, this list with the images will be returned by the function assigned to the Gradio interface and these images will be displayed in the Gallery format (see Figure 37: Images generated on Gradio Application using GLIDE Fine-Tuned model.).

5.- Conclusions and future work

Before beginning with the conclusions of this project, I want to thank the tutors of the HP SCDS company as well as my university tutor for the help provided to acquire the necessary knowledge to carry out the project as well as for guiding me during its development, offering the necessary help in any situation of difficulty and providing me with the tools both in the form of documentation and in the form of software and hardware when needed.

Bearing in mind that this TFG is a research project, and it has not gone at all times as initially planned, the initial objective of the project can be considered that has been fulfilled considering all the adversities encountered during the course of the project.

It has been possible to obtain greater knowledge in Deep Learning Generative Models as well as to implement models of this type and even improve the performance of a large model considered one of the best in this type of task at the time of starting the project. In addition, results have been able to be presented as a simple application where the ability to generate images of different models that in one way or another have been mentioned or deepened throughout the project can be compared.

Since I had never carried out a project of this caliber from scratch in this area during my university career, I have been able to learn how to deal with this type of projects on the area of Artificial Intelligence. I also improved the capacity of handling theoretical concepts with ease, which was the basis for carrying out the rest of the project successfully. I have also been able to verify the importance of the choice of dataset when training Deep Learning models.

To be able to carry out a project in this field, I believe that it is necessary to have understood and internalized many concepts of different branches acquired during the degree, so I can affirm that I have been able to combine a large part of what I have learned these four years in different subjects.

As a last detail of these conclusions, I think it is very relevant to add that I have acquired great fluency to understand complex and large code developed by professionals and I consider that this skill will be very important and useful for the beginning and course of my professional life.

Regarding future work or extensions that could be applied to improve this project, as a research project I believe that it can always be improved in one way or another.

I propose below three points of improvement:

1) As the main problem found in this work, the dataset used for the Fine-Tuning can always be improved to get closer to the objective of generating comic strips. Although this has been a difficult task since no specific datasets have been found and it has been resolved adequately, some images that make up the dataset that are even more specific with the objective of the project could achieve great improvements in it.

2) In order to present the results, an application has been implemented that contains everything necessary to interact with the different models, compare them and simulate small comic strips. However, this has been developed in a Google Colab Notebook and can only be accessed when it is run. As possible improvements, this same implementation could be deployed to a server in a Cloud environment to have greater availability of it.

   To obtain even a better final result, the implementation approach of this application could be modified and proceed to develop a Web Application instead of using the Gradio module so that it is more customizable and with greater possibilities when using it. As mentioned above, this Web Application could be hosted in a Cloud environment using the necessary services available from the different providers, considering that they contain specific services for the deployment of Artificial Intelligence models and web services. However, this implementation would come out the scope of an exclusive research project since it would add a large development component to it.

3) The world of technology and specifically Artificial Intelligence and Deep Learning is changing very quickly. That is why while this project has been developed, new models have come out that improved the performance of GLIDE but that could not be considered when performing the Fine-Tuning or deepen them to the same level as GLIDE since the project was too advanced and focused on this last model or because those models were not available to public use.

6.- References

[1] OpenAI. CLIP: Connecting text and images. *https://openai.com/blog/clip/* (2021). (Last accessed July 2022).

[2] Deng, J., Dong, W., Socher, R., Li, L. J., Li, K., & Fei-Fei, L. *ImageNet: A large-scale hierarchical image database. In 2009 IEEE Conference on Computer Vision and Pattern Recognition (pp. 248-255). IEEE*. (2009).

[3] Bagnato. Aprende Machine Learning: "¿Cómo funcionan las Convolutional Neural Networks? Visión por Ordenador" *https://www.aprendemachinelearning.com/como-funcionan-las-convolutional-neural-networks-vision-por-ordenador/* (2018). (Last accessed July 2022).

[4] Sotaquirá. Codificando Bits: "Padding, strides, max-pooling y stacking en las Redes Convolucionales." *https://www.codificandobits.com/blog/padding-strides-maxpooling-stacking-redes-convolucionales/* (2019). (Last accessed July 2022).

[5] Reynolds. Convolutional Neural networks (CNNs). https://anhreynolds.com/blogs/cnn.html (2019). (Last accessed July 2022).

[6] Baeldung. Baelding: "How ReLU and Dropout Layers Work in CNNs." *https://www.baeldung.com/cs/ml-relu-dropout-layers* (2022).

[7] Aqeel Anwar. Towards Data Science: What is Transposed Colvolutional Layer? *https://towardsdatascience.com/what-is-transposed-convolutional-layer-40e5e6e31c11#:~:text=Transposed%20convolutions%20are%20standard%20convolutions,in%20a%20standard%20convolution%20operation* (2020). (Last accessed July 2022).

[8] Ketan Doshi. Towards Data Science: Batch Norm Explained Visually — How it works, and why neural networks need it. *https://towardsdatascience.com/batch-norm-explained-visually-how-it-works-and-why-neural-networks-need-it-b18919692739* (2021). (Last accessed July 2022).

[9] Ho, Jonathan, Ajay Jain, and Pieter Abbeel. "Denoising diffusion probabilistic models." *Advances in Neural Information Processing Systems* 33 (2020).

[10] Ronneberger, Olaf, Philipp Fischer, and Thomas Brox. "U-net: Convolutional networks for biomedical image segmentation." *International Conference on Medical Image Computing and Computer-Assisted Intervention*. Springer, Cham (2015).

[11] Nichol, Dhariwal, Ramesh, Shyam, Mishkin, McGrew, Sutskever and Chen. "Glide: Towards photorealistic image generation and editing with text-guided diffusion models." *arXiv preprint arXiv:2112.10741* (2021).

[12] Dhariwal, Prafulla, and Alexander Nichol. "Diffusion models beat gans on image synthesis." *Advances in Neural Information Processing Systems* 34 (2021).

[13] Ho, Jonathan, and Tim Salimans. "Classifier-Free Diffusion Guidance." *NeurIPS 2021 Workshop on Deep Generative Models and Downstream Applications*. (2021).

[14] Vaswani, Shazeer, Parmar, Uszkoreit, Jones, N. Gomez, Kaiser and Polosukhin. "Attention is all you need." *Advances in Neural Information Processing Systems* 30 (2017).

[15] OpenAI. GLIDE [Source code]. *https://github.com/openai/glide-text2im* (2021). (Last accessed July 2022).

[16] Lin, T. Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D. & Zitnick, C. L. *Microsoft coco: Common objects in context. In European Conference on Computer Vision (pp. 740-755). (2014).*

[17] Ramesh, A., Dhariwal, P., Nichol, A., Chu, C., & Chen, M. (2022). Hierarchical text-conditional image generation with clip latents. arXiv preprint arXiv:2204.06125.

[18] Saharia, C., Chan, W., Saxena, S., Li, L., Whang, J., Denton, E., Kamyar, Karagol, S., Gontijo, Salimans, Ho, Fleet & Norouzi, M. *"Photorealistic Text-to-Image Diffusion Models with Deep Language Understanding." arXiv preprint arXiv:2205.11487.* (2022).

[19] Git-scm. Git –fast-version-control. *https://git-scm.com/* (2022). (Last accessed July 2022).

[20] Torrado. Introducción a GitLab. *https://desarrolloweb.com/articulos/introduccion-gitlab.html* (2017). (Last accessed July 2022).

[21] Colab.research. Google Colaboratory. *https://colab.research.google.com/* (2022). (Last accessed July 2022).

[22] PyTorch. PyTorch [Source code]. *https://github.com/pytorch/pytorch* (2022). (Last accessed July 2022).

[23] PyTorch. PyTorch. *https://pytorch.org/* (2022). (Last accessed July 2022).

[24] TensorFlow. TensorBoard. *https://www.tensorflow.org/tensorboard?hl=es-419* (2022). (Last accessed July 2022).

[25] Numpy. NumPy. *https://numpy.org/* (2022). (Last accessed July 2022).

[26] Goodfellow, Ian, Yoshua Bengio, and Aaron Courville*. Deep learning. MIT press* (2016).

[27] Labml.ai. Annotated PyTorch Paper Implementatios*. https://nn.labml.ai/* (2022). (Last accessed July 2022).

[28] Bircanoglu. Kaggle: Comic Books Images. Resized Comic Books Images with Train and Test Set. *https://www.kaggle.com/datasets/cenkbircanoglu/comic-books-classification* (2017). (Last accessed July 2022).

[29] Weng. From Autoencoder to Beta-VAE. https://lilianweng.github.io/posts/2018-08-12-vae/ (2018). (Last accessed July 2022).

[30] Weng. What are Diffusion Models? https://lilianweng.github.io/posts/2021-07-11-diffusion-models/ (2021). (Last accessed July 2022).

[31] Jain. Kaggle: Flick 30k Dataset. https://www.kaggle.com/datasets/hsankesara/flickr-image-dataset (2020). (Last accessed July 2022).

[32] Schuhmann. Laion-400-Million Open Dataset. *https://laion.ai/blog/laion-400-open-dataset/* (2022, version accessed of May 2022). (Last accessed July 2022).

[33] Neverix. Kaggle: GLIDE multiple-image tuning. *https://www.kaggle.com/code/neverix/glide-multiple-image-tuning* (2022). (Last accessed July 2022).

[34] Mullis. Replicate: Pyglide. https://replicate.com/afiaka87/pyglide (2022). (Last accessed July 2022).

[35] Gradio. Build & Share Delightful Machine Learning Apps. https://gradio.app/ (2022). (Last accessed July 2022).