Deep Learning - Generative Adversarial Text to Image Synthesis

Alex Valle (4854159) Gabriele Berruti (4791919)

Summary

The first observation to make is that a longer version of this short report has been drafted, which contains more details on the implementation and theoretical aspects of the methods in the paper and also on our implementation.
The paper that has been chosen to be analysed is a 2016 Paper that showcases the effort of six researchers from the University of Michigan and Saarbrücken (Germany). The document shows how the use of GANs (Generative Adversarial Networks) allows for advancements in the generation of synthetic images starting from a textual description. It compares the method proposed by the research with previous architectures that, although far from the described goal, are capable of obtaining valid textual feature representations. In particular, the paper aims to demonstrate the effectiveness of the model in generating images of birds and flowers based on a precise textual description of them. In 2016, the ability of an AI system to generate realistic and coherent images from textual descriptions (such as "small red bird with a blue beak") was a current issue and far from being achieved. It should be noted that it's necessary to use natural language and domain-specific attributes to describe the image to be generated.
The difficulty of translating words into images may be divided into two subproblems. First, learn a feature vector from a specific text based on the visualization we want to obtain . Given these features through the use of a certain architecture, create a realistic and coherent image. This paper describes a GAN that creates 64x64 images from text descriptions, utilizing a character-level encoder and a unique manifold interpolation regularizer for fine-grained datasets such as CUB and Oxford Flowers.
During our work, we attempted to reimplement the version proposed by the Paper in order to determine its correctness and to be able to reapply it to different domains and tasks, but unfortunately, it was not possible for the reasons described in the "Experiments" section, while we successfully managed to implement a "toy" version with a different architecture.
All the code related to the Project is public and available on Github at this address :
GitHub repository

Methodologies

This section outlines how the methodology is structured . It describes the datasets, GAN architectures, and optimization methodologies, as well as the encoders used for text and images . It also highlights changes and improvements implemented in our work compared to previous models.

All the methods are based on a GAN architecture with a Generator G and a Discriminator D .
And use (φ) as Text Encoder that is a character-level convolutional- recurrent neural network .

The generator is defined as :

$$G : \mathbb{R}^Z \times \mathbb{R}^T \to \mathbb{R}^D,$$

And the discriminator is defined as :

$$D : \mathbb{R}^D \times \mathbb{R}^T \to \{0, 1\},$$

where

D is the dimension of the generated image

Z is the dimension of the input noise

T is the dimension of the text embedded with φ

Optimization Formula

The optimization objective is defined as:

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{\text{data}}(x)}[\log D(x)] \quad (1)$$

$$+\mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))]$$

$D(x)$ The Discriminator uses a Sigmoid activation function, mapping outputs to $[0, 1]$.

$G(x)$ The Generator outputs data in $\mathbb{R}^D$.

The Discriminator is maximized to distinguish real from fake data, while the Generator is minimized to make $G(z)$ indistinguishable from real data. Maximizing D involves maximizing $\log D(x)$ for real data and $\log(1 - D(G(z)))$ for generated data. Alternatively, G can also be optimized using $\log D(G(z))$ instead of $\log(1 - D(G(z)))$ to achieve the same result.

Paper Datasets

All datasets share the following features: each image is paired with 5 text captions, and during training, a random caption and an augmented version of the image (e.g., crop, flip) are used for each mini-batch.

Caltech-UCSD Birds Dataset (CUB): Contains 11,788 images of birds across 200 categories. The dataset is split into 150 categories for training and validation, and 50 categories for testing.

Oxford-102 Flowers Dataset: Includes 8,189 images of flowers grouped into 102 categories. The dataset is split into 82 categories for training and validation, and 20 categories for testing.

MS COCO Dataset: Used for testing with a broader range of general images and text descriptions.

Text and Image Encoder

To encode textual descriptions and images, the original paper utilized a symmetric architecture combining text and image representations, following the approach of Reed et al. (2006). The dataset consists of N samples, where each sample includes an image, its text description, and its class label.

Encoders Overview The image encoder is based on GoogLeNet, producing 1,024-dimensional embeddings, while the text encoder employs a deep convolutional-recurrent architecture (char-CNN-RNN), also generating 1,024-dimensional text embeddings. These embeddings are mapped to a shared space, enabling compatibility between textual and visual modalities.

Training Objectives The main goal is to ensure that text descriptions are most compatible with images from their respective classes. To achieve this, we compute the similarity between the encoded version of the input image v and the current text description t related to the current class y. The encoded versions, $\phi(v)$ and $\varphi(t)$, return vectors, and the similarity is computed via their dot product, which returns a scalar. This scalar represents how well the image and text description match. The goal is to maximize the correlation between the image and the text of the same class, ensuring that the image-text pairs that belong to the same class have the highest similarity.

Loss Function To optimize the encoders, a hinge loss or 0-1 loss function is used. This loss function penalizes the similarity score between mismatched image-text pairs, while encouraging high similarity scores for matching pairs. Specifically, the loss function encourages the model to predict the correct class y by minimizing the hinge loss, defined as:

$$\Delta(y, f(x)) = \max(0, 1 - y * f(x))$$

where $f(x)$ is the classifier output for the given image or text input. This results in a classification loss that forces the encoder to learn meaningful representations.

Our Implementation Details In our implementation, instead of using the original encoder described by Reed et al. (2006), we used CLIP (Contrastive Language-Image Pre-Training), a model developed by OpenAI. The CLIP model is pre-trained on large datasets and fine-tuned on the CUB and Oxford-102 datasets to generate relevant vector representations for both images and text descriptions. Pretraining on large datasets accelerates the training process and improves the model's ability to generalize across different domains. Fine-tuning on domain-specific datasets such as CUB and Oxford-102 enhances the quality of the embeddings for the task at hand.

Paper's Architectures and our

The GAN architecture used in the paper follows the implementation where all training images are resized to $64 \times 64 \times 3$. Text embeddings are projected into a 128-dimensional space and concatenated with convolutional feature maps in both the generator and discriminator networks. During training, the generator and discriminator are updated in alternating steps to optimize the model. The hyperparameters used include a base learning rate of 0.0002, ADAM optimizer with a momentum parameter of 0.5, generator noise sampled from a 100-dimensional unit normal distribution, a mini-batch size of 64, and 600 epochs. The Paper implementation is based on the dcgan.torch2 framework.

Vanilla GAN: Generative Adversarial Network without Conditional Latent Space

The Vanilla GAN consists of a Generator that only relies on the noise vector z and a Discriminator that evaluates whether the input image is real or generated, without any dependence on a conditional latent space.

GAN-CLS: Generative Adversarial Network with Conditional Latent Space

GAN-CLS utilizes a text encoder, φ (such as CLIP or GoogLeNet), to create embeddings from text descriptions t that are related to images x, resulting in embeddings h and $\hat{h}$ for mismatched descriptions. A random noise vector z is sampled from a Gaussian distribution and passed through the Generator to generate an image $\hat{x}$. The Discriminator computes a score s_r for the real image and its corresponding text embedding. For a fake image, the Discriminator calculates s_f.
The Discriminator loss L_D is computed as:

$$L_D = d\_loss\_f + d\_loss\_w + d\_loss\_r$$

The Generator's loss, L_G, consists of three parts: BCELoss, MSELoss, and L1Loss, and is calculated as:

$$L_G = BCELoss(s_f, \text{real\_label})$$
$$+MSELoss(q_r, q_f) + L1Loss(\hat{x}, x)$$

The losses are backpropagated and the models are updated with the learning rate α.

GAN-INT: Generative Adversarial Network Interpolated

This architecture improves upon GAN-CLS by interpolating between two text embeddings. The Generator is conditioned on the interpolated embeddings to generate more diverse images. Although this modification has not been implemented yet, it can be added with minor adjustments.

$$\mathbb{E}_{h1,h2\sim p_{\text{data}}}[\log(1 - D(G(z, \beta h1 + (1-\beta)h2)))]$$

WGAN: Wasserstein GAN - Our Architecture

WGAN introduces the Wasserstein distance to improve training stability. The Discriminator computes the difference between real and fake scores, and the Discriminator's weights are clipped to satisfy the Lipschitz constraint. The Generator is trained to maximize the Discriminator's score. The WGAN training steps are as follows:

Algorithm 1 WGAN training algorithm with step size α

Require: minibatch images x, matching text t, mismatching text $\tilde{t}$, training batches S, Discriminator iterations NI

for $n = 1$ to S **do**
  $h \leftarrow \varphi(t)$ {Encode matching description}
  for $i = 1$ to NI **do**
    $z \sim \mathcal{N}(0,1)^Z$
    $\hat{x} \leftarrow G(z, h)$
    $s_r \leftarrow D(x, h)$
    $s_f \leftarrow D(\hat{x}, h)$
    $L_D \leftarrow s_r - s_f$
    Clip weights of D within $[-c, c]$
  end for
  $L_G \leftarrow -s_f$
end for

The Wasserstein loss encourages the Generator to produce images that the Discriminator cannot easily distinguish from real ones. The iterative process ensures stable GAN training and minimizes the Wasserstein distance between real and generated data distributions.

Experiments and Paper Qualitative Results

The paper compares several GAN architectures: the GAN Baseline, GAN-CLS, GAN-INT, and GAN-INT-CLS. The GAN Baseline and GAN-CLS models correctly reproduce some color information, but the generated images do not appear realistic. The GAN-INT and GAN-INT-CLS models produce more plausible images that match all or part of the captions. The results on the CUB dataset showed that GAN-INT and GAN-INT-CLS performed better in generating bird images compared to GAN Baseline and GAN-CLS. On the Oxford-102 Flowers dataset, all models generated plausible flower images that aligned with their captions. The GAN Baseline model exhibited the highest variety in flower morphology, particularly when the caption did not specify petal types. However, models like GAN-CLS, GAN-INT, and GAN-INT-CLS generated more class-consistent flower images. It was speculated that generating flowers is easier than birds due to structural regularities in bird species, which make it easier for the discriminator to identify fake birds. The supplementary materials also provide additional results, including examples for the GAN-INT and GAN-INT-CLS models on both the CUB and Oxford-102 datasets, and the Vanilla GAN, an end-to-end variant of GAN-INT-CLS that does not rely on pre-training the text encoder.

The GAN-CLS model was trained on the MS-COCO dataset, which features more complex images with multiple objects and variable backgrounds. Despite the complexity, the model showed sharpness and diversity, but struggled with scene coherence in complex scenarios like human figures in baseball scenes. A comparison with AlignDRAW revealed that GAN-CLS produced sharper, higher-resolution outputs, while AlignDRAW captured finer variations in text.

In conclusion, the model successfully generates images from detailed descriptions, with manifold interpolation enhancing results. The approach also demonstrated the ability to disentangle style and content and showed generalizability to more complex datasets like MS-COCO. Future work will focus on improving resolution and handling more diverse text inputs.

Our Code implementation

The code has been written in Python , while back in 2016 the researcher decided to write the code for training the Net in LUA . So first of all we understand the code written in LUA by the authors and after we decide to translate it in Python . The main reason why we do it is that the code was deprecated after about 10 years (especially the library) and for us was not possible to try to train the Net ourself . Talking about the Datasets they were also not reacheable because the hyperlink redirect to a website not existing anymore . So we try to develop a solution in which we train the Net with the same concept described before but using different Datasets . Also the trained weights cannot be adapted to the new solution that we develop , so we decide so re-train the Network from zero for about 200 epochs . After the 200 epochs we decide to stop the training because it was taking too long .
The code in subdivided in different part and diffent .ipynb file . The first one is "1) dataset usage.ipynb" in which is possible the are two example on how to load Bird and Flowers Dataset using the module "gan_t2i" and store it in HDF5 format other than using a Dataloader on this datasets . The second file is "2) CLIP - Fine Tuning.ipynb" in which is shown how to load the CLIP model (ViT-B/32) and to train it (we use an already trained CLIP network to extract text features) The Third file is "3.2) COLAB GAN example.ipynb" . In this file we combine all the module that we develop until now . First of all we decide which network we would like to train between : "Vanilla GAN", "GAN_INIT_CLS" and "WGAN" . After this part we download the weights related to the CLIP Network used to extract the text features and also the model itself . What about the Dataset initially it is stored in HDF5 format but this time is also transformed and normalized other than tokenized . After this section we create the training, validation and test dataloaders and check the outcome size of the CLIP model feature related to text and images . And then depending on the Newtwork that we would like to train we define it and define the embedding projection dimension that in our case is 128 . Finally after this long pre-processing and initilization part we can decide if start to train the Net using the correspondent Algoritm from zero or from a specific checkpoint (which corrispond to the weights) loading it in the Model . The result on the GAN model after 186 epochs of training from scratch are not enough to prodoce a result which correspond to the description given by the text , that's due to the fact that the Net is too complex and in fact to achieve a satisfactory result, at least 600 epochs are necessary, as the original paper also specifies. Besides the visualization factor it's possible to observe that the loss related to the Discrimination and Generator part tend to move in the right direction , in fact the Generator loss especially in the WGAN model tend do descrease (minimize) and the Discriminator loss it's maximizing it's values as well as described in the Min-Max optimization formula .

WGAN on FASHION-MNIST Dataset

We try also to conduce some experiments on training the Net addressing the problem to other Datasets such as Fashion MNIST dataset (keras.datasets.fashion_mnist) .
So in general the WGAN employs the Wasserstein distance to define a value function with properties theoretically superior compared to the one introduced in the original GAN paper . As seen before to verify that the Discriminator (critic) respects the 1-Lipschitz requirement, the authors implemented weight clipping. However, this technique might lead to problems like as low convergence in deep critics and other unwanted phenomena. WGAN-GP substitutes weight clipping with a "gradient penalty." This variant includes a loss term to keep the L2 norm of the discriminator gradients around a set value, allowing for smoother training . But it's based on the Algoritm **??** flow-chart described in the section related to the Architectures.

Data Preparation

The dataset used is the FASHION-MNIST dataset , in which each sample is a 28x28 grayscale pixel image with values normalized in the range [-1,1] . The label related to the image will not be used in this case to train the WGAN .

Discriminator Net

In this case the Discriminator use a Zero Padding layer to transform the input image (28,28,1) into a shape of (32,32,1) and other 4 Convolutional Block to transform it in (16,16,64) $\rightarrow$ (8, 8, 128) $\rightarrow$ (4, 4, 256) $\rightarrow$ (2, 2, 512) Leaky Relu is used as activation function .
The Convolutional Block is composed of 2D Convolutional Kernel and depending on the parameters a Normalization and dropout layer . The output shape of the discriminator is one value that rappresent the "real" or "fake" classification of the input image, indicating whether the image is from the real dataset or generated by the Generator.

Generator Net

The Genetor Net use the upsample_block(..) function to handle the upsampling process by increasing the spatial dimensions of the input with UpSampling2D, followed by a Conv2D layer . Depending on the parameter it may also include BatchNormalization and a specific activation function (like LeakyReLU or Tanh) to introduce non-linearity and improve training stability other than a Dropout layer to prevent overfitting. The Generator starts by taking a noise vector as input, transforming it through a Dense layer into a tensor of shape (4, 4, 256). The data is then passed through three Upsampling blocks . In the same way in the second block, the shape changes from (8, 8, 128) to (16, 16, 64), again using UpSampling2D and Conv2D, with BatchNormalization and LeakyReLU applied. The third one increases the shape from (16, 16, 64) to (32, 32, 1) by applying UpSampling2D and Conv2D to reduce the channels to 1, followed by a Tanh activation to ensure the output is in the correct range. Lastly , a Cropping2D layer is applied to reduce the spatial dimensions from (32, 32) to (28, 28) to generate the image .

WGAN-GP MODEL

The model overrides the `keras.Model` module and overrides the `train_step` function derived from the model. The model consists of a discriminator and a generator. The discriminator processes real and fake images, while the generator creates fake images from random noise. The model is initialized with specific hyperparameters like `latent_dim` (dimension of the random input to the generator), `discriminator_extra_steps` (extra steps to train the discriminator), and `gp_weight` (weight for the gradient penalty).

Gradient Penalty Function

The model uses a custom gradient penalty function, which helps to enforce the Lipschitz constraint by calculating the norm of the gradient of the discriminator's output with respect to interpolated images. This is added to the discriminator's loss.

Training Step

During training, we work using batches of 512. The discriminator is updated multiple times per generator step (3 extra steps are typically used). For each discriminator step, fake images are generated based on the noise vector. The discriminator is used two times on the fake and real images, and the discriminator's loss is computed based on the function passed (we will define it after), obtaining `d_cost`. Based on the real and fake images, we compute the gradient penalty. Finally, the total discriminator loss (`d_loss`) is computed as :
`d_loss = d_cost + gp * self.gp_weight`.
The generator is trained after the discriminator. The generator's loss is computed based on the discriminator's output, and it depends on the generated images coming from the Generator Network. The generator loss function will be defined after. Each model (discriminator and generator) is updated using

their respective optimizers. The training process alternates between updating the discriminator and the generator, with the gradient penalty ensuring stability. The optimizers used for this task are the same and correspond to Adam with a learning rate of 0.0002.

Discriminator Loss

The discriminator loss is simple and computes the mean of the batch values obtained on real and fake images, as well as the difference between them.

Generator Loss

The generator loss is straightforward because it is the negative mean of the logits (unnormalized output values produced by the model) for fake images, pushing the generator to produce images that the discriminator classifies as real. A positive score suggests the image is real, and a negative score suggests it is fake. By minimizing this negative loss, the generator is maximizing the discriminator's score for the generated images, effectively making the fake images more "realistic" in the eyes of the discriminator.

Training time

The network is trained for around 30 epochs, with a noise dimension of 128. After each epoch, a callback generates and saves a number of images based on the latent noise vector as PNG files. The values from the generator are in the range of [-1, 1] due to the `Tanh` activation function, so these values are rescaled to the range [0, 255] to be saved correctly. We perform the inverse process to normalize the pixels in the range [-1, 1] when preparing each sample of the dataset.

In general , especially because the problem has been simplified with respect to the previous one , after only 50 epochs we can appreciate good result in term of reconstruction of image related to the MNIST datasets . And thanks to the gradient penalty term we can generalize the enough to obtain a good reconstruction also on "zero-shot" data coming from the same dataset .

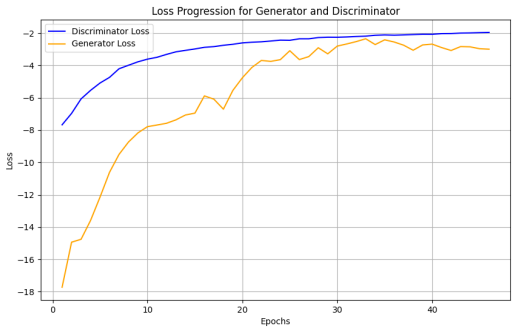

Figure 1: *Discriminator Loss during Training (WGAN)*

The visual results of the model's performance after various epochs are shown below. As it is possible to observe, the reconstructions generated by the "Generator" become increasingly coherent with the noise given as input as the epochs increase.

Figure 2: *Grid of generated images at different epochs*