UNIVERSITETI POLIS

FACULTY FOR RESEARCH AND DEVELOPMENT

BACHELOR

IN COMPUTER SCIENCE

October 2021 - September 2024

THESIS : Generative AI

Student: Adea Nesturi

Udhëheqësi: MSc.Andia Vllamasi, PhD. Valerio Perna

Tirana, July 2024

Autostrada Tiranë-Durrës, Km 5 Kashar, Tiranë, Shqipëri, Tel: +355 (4) 2407420 / 2407421, Fax: +355(4) 2407422,
Cel: +355 (0) 694088111www.universitetipolis.edu.al

info@universitetipolis.edu.al

Acknowledgements

I want to express my deepest gratitude to everyone who has supported and guided me in completing this thesis.

First and foremost, I would like to thank my advisors, MsC. Andia Vllamasi and Phd. Valerio Perna, for their invaluable guidance, support, and encouragement throughout this research. Their time, effort, and valuable suggestions greatly contributed to the improvement of this thesis. Their insightful feedback and expertise have been instrumental in shaping this work.

I want to give special thanks to my friends at Polis University for their support. The stimulating discussions and shared experiences have made this journey more enjoyable. On a personal note, I would like to extend my heartfelt thanks to my family for their unwavering support and encouragement and for seeding in me the values of determination and dedication. To my 3 closest friends, for their encouragement and patience and for always helping me to aim higher to reach for my dreams with confidence and determination.

I would also like to express my sincere gratitude to Polis University for being by my side throughout my entire journey. This support has enabled me to successfully complete my bachelor's studies in Computer Science. I am also grateful to the university staff for their continuous support and assistance.

Abstract

Generative AI, characterised by its ability to create content autonomously, has revolutionised various aspects of digital media production, from image and video synthesis to artistic expression. Through advanced algorithms like Generative Adversarial Networks (GANs)[1] and Variational Autoencoders (VAEs)[2], AI systems can generate realistic visuals, manipulate images, and even create entirely new artistic styles. This thesis will discuss the evolution of generative AI techniques, their applications in digital visual media, including filmmaking, advertising, and design, as well as the ethical and societal considerations surrounding their adoption. Additionally, it explores future directions and challenges in the intersection of generative AI and digital visual media, highlighting the potential for innovation and creativity in this rapidly evolving field. Generative AI is the present and future of AI and deep learning, and it will touch every part of our lives. It is the part of AI that is closer to our unique human capability of creating, imagining and innovating. This diploma is aimed towards creating a range of generative architectures, from basic to advanced, until we reach multimodal AI, where text and images are connected in incredible ways to produce amazing results.

Keywords: DCGANs: Deep Convolutional Generative Adversarial Networks; GANs: Generative Adversarial Networks; HFG: Human Face Generation; IQE: Image Quality Evaluation; UL: Unsupervised Learning.

Generative AI e karakterizuar nga aftësia e saj për të krijuar përmbajtje në mënyrë autonome, ka revolucionarizuar aspekte të ndryshme të prodhimit të mediave dixhitale, nga sinteza e imazheve dhe videove deri te shprehja artistike. Nëpërmjet algoritmeve të avancuara si Rrjetet Generative Adversarial (GANs) dhe Variational Autoencoders (VAEs), sistemet e AI janë të afta të gjenerojnë pamje realiste, të manipulojnë imazhe dhe madje të krijojnë stile krejtësisht të reja artistike. Kjo tezë do të diskutojë evolucionin e teknikave gjeneruese të AI, aplikimet e tyre në mediat vizuale dixhitale, duke përfshirë filmimin, reklamimin dhe dizajnin, si dhe konsideratat etike dhe shoqërore që lidhen me adoptimin e tyre. Për më tepër, ai eksploron drejtimet dhe sfidat e ardhshme në kryqëzimin e AI gjeneruese dhe mediave vizuale dixhitale, duke theksuar potencialin për inovacion dhe kreativitet në këtë fushë me zhvillim të shpejtë.Generative AI është e tashmja dhe e ardhmja e A.I. dhe të mësuarit e thellë, dhe do të prekë çdo pjesë të jetës sonë. Është pjesa e AI që është më afër aftësisë sonë unike njerëzore për të krijuar, imagjinuar dhe inovuar. Kjo diplomë synon të krijojë një sërë arkitekturash gjeneruese, nga ato themelore tek ato të avancuara, derisa të arrijmë A.I multimodal, ku teksti dhe imazhet janë të lidhura në mënyra të jashtëzakonshme për të prodhuar rezultate të mahnitshme.

Fjalet Kyçe: DCGANs: Deep Convolutional Generative Adversarial Networks; GANs: Generative Adversarial Networks; HFG: Human Face Generation; IQE: Image Quality Evaluation; UL: Unsupervised Learning

[1] A **generative adversarial network** (**GAN**) is a class of machine learning frameworks and a prominent framework for approaching generative AI.

[2] **Variational Autoencoders (VAEs)** are generative models explicitly designed to capture the underlying probability distribution of a given dataset and generate novel samples.

Table of Contents

List of abbreviations

DCGANs - Deep Convolutional Generative Adversarial Networks
CLIP - Contrastive Language-Image Pre-training
GANs - Generative Adversarial Networks
IQE - Image Quality Evaluation
HFG - Human Face Generation
UL - Unsupervised Learning
VAE - Variational Encoders
AI - Artificial Intelligence

Summary of pictures

Chapter 1

1.1 Aim of this study

This diploma aims to build a framework for estimating generative models via an adversarial process, where we train two models: a generative model G that captures the data distribution and a discriminative model D that estimates the probability that a sample came from the training data and not G. This framework corresponds to a minimax two-player game. In the studied framework, the generative model is pitted against a discriminative model that learns to determine whether a sample is from the model distribution or data distribution.

1.2 Methodology

We have seen many kinds of deep learning architectures and methods in the last few years. We have witnessed supervised learning in which we show and train the neural networks with many input-output pairs. So we supervise the learning. Providing the labels for each example. Then, we have self-supervised learning in which we only have the data input and train the network to detect patterns within that data input. For example, we hide parts of the data and train the network to fill the missing parts. We have an agent that acts in an environment and trains the network to generate a policy to take the correct actions to develop the maximum amounts of a specific type of reward.

In the last years, specifically in 2014, a new era and new types of learning and AI architecture, which are generative models in AI, began. At the time, it starts the process of us coming home in a way with AI. But why come home? By adding to that supervised learning, self-supervised learning, and reinforcement learning, adding the generative and creative capabilities of creative AI, we are coming home to our origins and our future.

Our nature is creative, generative, evolutive, fractal. And so if we think very quickly about the beginnings of humankind, right at the start in the first times of humanity. We were discovering, exploring. After a while, we began to imagine and paint what we were interacting with. We began to create variations of those things. As we fast forward through our story, we arrive at the last centuries, when we start to connect it all. We connect different fields, such as science, technology, generation, and creation. As we arrive at the 20th century, the era of computational creativity begins. We begin to wonder, can we use machines? Can we use computers to create and generate? In the seventies and eighties, we started to use algorithms to create new worlds, such as in video games with procedural algorithms. We are beginning to play with fractals[3], patterns that the laws of nature repeat at different scales. For example, trees are natural fractals, patterns that repeat smaller and smaller copies of themselves to create all that biodiversity. But

[3] A fractal is a complex geometric pattern created through mathematical equations and algorithms, exhibiting self-similarity with smaller copies at different scales.

we also see this within ourselves because many of our organs and structures are fractal; it happens in the lungs we breathe.

There is a code of genius called Inigo Quilez from Spain. He created a platform called Shadertoy[4], and he uses pure mathematics, sines and cosines, trigonometry and exponential functions and other types of mathematical functions to produce worlds and landscapes that are all created with mathematical calculations. This creates a transition between the ages of the human generator and when the entity that was building and generating was always the human. Now we go to the era in which humans and machines create together; they imagine together. And what could be more exciting than this?

And so, the generative AI revolution is being expressed in many different fields. We have seen an evolution in the last few years, from 2014 until now. We are starting a new decade, a decade of the 2020s, where different models and architectures can invent, create, and imagine visuals, text, etc. A lot of scientists are proposing that generative AI is going to be the future of AI. Max Welling, one of the top experts in the field, says this. He says that, as humans, we are running continuous simulations of reality. So, we are imagining generating our reality, integrating over the expected value that we are predicting. So many of these experts think that Artificial General Intelligence (AGI)# will manifest, among others, through the combination of generation and causality. Our perception as humans is a generative model trained to produce simulations of our environment. So, as human beings, our brain and mind are constantly generating, predicting, creating predictions, and simulations of what's going to happen and checking if these generations and predictions fit with what's happening and what's going to happen.

Artificial Intelligence is born from the combination of two approaches. We require precision from the ability to predict, classify, and recommend. We combine that with the capacity to deal with entropy and the capacity to imagine, augment and create. We are combining all of the prediction classifications we have done in the last few years with the capacity to invent, make, and generate. We have been doing much of that exploitation and can now explore. We will be combining accuracy and entropy[5] and divergence and convergence. All of these combinations are taking us to a new type of balance. There are scientists like Lee Smolin, a theoretical physicist, and his reference in the book Road to Reality of Roger Penrose, who has a theory that all of this generation combined with evolution could be one the keys to our universe. He has this theory in which the singularities within the black holes are the source of new faces of the universe. So, each singularity of a black hole may produce a different phase of the universe in which the fundamental constants of physics are readjusted. So, it seems that the basic constants of our

[4] Shadertoy is an online community and tool for creating and sharing shaders through WebGL, used for both learning and teaching 3D computer graphics in a web browser.
[5] In computing, entropy is the randomness collected by an operating system or application for use in cryptography or other uses that require random data. This randomness is often collected from hardware sources, either pre-existing ones such as mouse movements or specially provided randomness generators.

universe favour the proliferation of black holes. So again, it's just another demonstration that the generative principles may be everywhere in our universe.

And so, by doing so, generative principles may be everywhere in our universe. And by doing generative AI, we are coming home. AI is coming home. The further we go, the closer we get. The further we go, the closer we get. After automating many processes in the last decades in previous years and being able to manufacture, transform, predict, and converge to solutions, we are now coming home in conjunction with exploration and exploitation, precision and entropy, divergence and convergence. We're just coming home to a new age of democratising growth and expansion that will increase human potential.

1.3 Applications of Generative AI

Generative creativity will allow us to explore data in new and more profound ways. We work with visual elements of 1000 x 1000 pixels, which is around 1 million dimensional space, which allows us to explore these architectures in incredible ways. We can generate completely new samples and data from these latent spaces. For example, where the natural images live, we're going to transform them into mathematical spaces with a much lower amount, much lower dimensionality and a much lower amount of dimensions. From those, we will be able to generate entirely new data. When we do that transformation, we have an increased opportunity for serendipity. We can explore many unexpected possibilities and infer new patterns from those generations performed.

So, what is going to be our palette? In generative and creative AI, we can work with any kind of continuous space, space of images, audio, as well as discrete spaces like text, 3D models, etc. Anything that is data or information, we can generate it into something new. This can be expressed in the design field of producing new products like jewellery, shoes, and text. We can navigate probabilistic spaces through generative AI to explore millions of objects of the same family, where the proximity connects the similarity in various directions.

We know that the GPT model generates new text and games. The synthesis of new walls and textures in architecture. We have seen models of 3D design with chairs and multimodal AI[6]. In multimodal AI, we include data of different types in the models to convert between them, for example, audio to text, language to another language, images from text, etc. Recently, we have worked with Clip Model Writing, where we connect visual elements with the natural language to provide the prompt and generate visuals from it.

What is CLIP? CLIP is a method of learning from natural language supervision which stands for Contrastive Language-Image Pre-training.

[6] Multimodal generative AI models add a new layer of complexity to state-of-the-art LLMs.

In summary, CLIP is a model that embeds images and text in the same vector space. It is trained using 400 million image and text pairs, which are self-supervised. Similar embeddings are created for an image and a corresponding text. For instance, an image of a dog and the sentence "an image of a dog" would have similar embeddings and be close to each other in the vector space. This is important as it can be used to build many interesting applications, such as searching for an image in a database with a description or vice versa.

The authors found that CLIP can be used for various tasks on which it was not trained. It even achieved remarkable zero-shot performance on multiple benchmarks such as ImageNet, an image classification dataset. Zero-shot learning refers to the fact that the model was not explicitly trained on any of the 1.28M training examples in the ImageNet dataset.

To classify images using CLIP, you can embed each of the 1000 possible classes/objects in ImageNet with CLIP using the prompt "a photo of an {object}" (for example "a photo of a dog" or "a photo of a cat"). This gives you 1000 different embeddings corresponding to all the possible classes. Next, you can embed the image you want to classify with CLIP. Finally, you can take the dot product between the image embedding and all the text embeddings. Since CLIP is trained so that both images and text are in the same embedding space, and the dot product calculates the similarity between the embeddings, it is highly likely that the dot product with "a photo of a dog" will be the highest. Thus, you can predict that the image is a dog. If you want to turn CLIP into an accurate classifier, you can also pass the dot products through the softmax function to get a predicted probability for each class.

1.4 Latent spaces and representation learning.

Preparation for the coding phase: What does a generative model describe? It represents a dataset - a set of data and information generated using a probabilistic model. A probability distribution[7] is a mathematical function that gives the probabilities of different possible outcomes for an experiment. By sampling that model, we can generate new data.

Machine Learning (ML) and Deep Learning (DL) enable machines to learn from past data and make predictions or decisions based on future, unseen data. This technology is inspired by the human mind and has numerous applications with endless possibilities. With quintillions of data generated worldwide, obtaining fresh data is easily accessible. However, working with this vast amount of data requires using algorithms, which can be categorised into two groups. We have to distinguish between discriminative models and generative models.

Discriminative models are models that are used in supervised learning[8]. It estimates the probability that an observation x belongs to class y. One of the examples of discriminative

[7] Generative AI models aim to learn the underlying probability distribution of the data they are trained on, allowing them to generate new samples that resemble the training data.

[8] Supervised learning is a category of machine learning that uses labeled datasets to train algorithms to predict outcomes and recognize patterns.

modelling is supervised learning, which will be used in the coding project along the way. In contrast, the generative model is just about understanding the probability of generating a specific type of information, a particular type of input and a specific type of data. And so we will teach the model that, too, from a particular input. Typically, we don't use labels, although we can use them because we have conditional generative AI in which we can also condition the process with labels. In traditional basic generative AI, we don't use labels. An example of generative modelling could be self-supervised learning or pure generative AI. The generative model is about understanding the probability of generating a specific type of information, a particular type of input, or a specific type of data. It teaches the model that would be a noise of a certain image and generate a specific output, which can be the image of a particular photo.

An essential part of deep learning and AI is that we want to learn representations of the data. Representation learning learns the most important features to describe the data and how to generate those characteristics (mapping function[9]). It identifies the non-linear mathematical manifold[10] (because the connection between the information is computed non-linear ways) in which the data resides and sets the required dimensions to describe that space. If we have a set of images of cats or dogs, we want to know the essence of that data. We want to learn lower-dimensional mathematical spaces with good representations of the original spaces' essence. We learn to form and group features characteristics of a highly dimensional space in a space of fewer dimensions. For example, those natural images of landscapes in a space of fewer dimensions.

Instead of directly working with a high dimensional space, we describe the data using a latent space of lower dimensionality. So, the advantage of working with a latent space of less dimensionality is that it is easier to manipulate the space and do operations. An example is the StyleGAN generative architecture, in which one of the dimensions of the space could define or express the degree of the smile on a face, such as if the face has glasses or the size of the nose, etc. If we have 1000 times 1000 pixels limits, that is one million pixels. One million dimensions are very expensive and complex to work with that data. And not only because of the computation. If we think of an image of a person that may contain one million pixels. But what is more straightforward? It is essential to work with each of the individual pixels by pixel or to summarise that image in specific features like that person has glasses, that person is smiling, that person has long hair, and only works with those features. What is simpler? Of course, the second is to reduce that very high dimensionality to just a few dimensions of features we will work with. That is why we will convert and transform those very high-dimensional spaces into smaller spaces that we call latent spaces of lower dimensionality. We are going to be learning a mapping function that takes points. First, we will map the very high-dimensional latent space and reduce all the very high-dimensional spaces to smaller dimensional spaces. We can take points in that

[9] A mapping shows how the elements are paired. Its like a flow chart for a function, showing the input and output values
[10] It is based on the manifold hypothesis which says that in a high dimensional structure, most relevant information is concentrated in small number of low dimensional manifolds.

little latent space to reconstruct and recreate new points in the original domain. This could be new images of cats, dogs or people. And so, each of the points is in the original high-dimensional space. This is representation learning.

Something fascinating and fun to think about is that we are always doing representation learning as humans ourselves. For example, we always map high-dimensional visual spaces to latent spaces when we describe people by their hair colour or height. For example, we talk about a specific individual and say, “Have you seen x lately? She has dyed her hair red and looks beautiful; she has new glasses, etc.” And the person listening to us describes the latent space as a high dimensionality. We have to visualise x in our minds from the simplified low-dimensional description being explained to us. As we imagine, we are mapping from the point in the latent space back to the high-dimensional visual space of images. We are doing this all the time, and so we learn that with coding this type of architecture, the same process happens in our minds, in the human brain of mapping.

Mapping from the high-visual dimensional space to a latent spacing, which we describe that complexity with a small number of dimensions and then back from the latent space back to the high-dimensional space again - which is the learning essence of the probabilistic space of the input data. By doing this navigation, we can do many incredible things, such as adding more smiles to a face to have a picture of a person and add more smiles to that picture. How can we do that? Once we have learned the latent space of the images of faces of people, we have learned a mathematical space with a small number of dimensions that expresses the essence of those faces. What we can do is find the direction, the vector in that latent space that points in the direction of more smiles. Then, if we add that vector to the position in the latent space of the original image, we will obtain a new point in the latent space, which we map back to the high-dimensional space, giving us a picture with more smiles.

1.4 Navigating Latent Spaces

Therefore, these latent spaces learn the essence of the probabilistic space of the input data. Latent space is a crucial concept as it captures the fundamental features of the input data in a lower-dimensional space. This mathematical representation groups similar items together, making it a valuable tool for various applications, including language processing and image generation. Navigating from latent spaces, sampling from those latent spaces and interpolating between positions in those latent spaces, we can do many incredible things, like adding more smiles to a face. We could have a picture of a person and add more smiles to a face. How do we do that? Once we have learned the latent space, for example, of the images of faces of people, we have learned a mathematical space that, with a small number of dimensions, expresses the essence of those faces, then what we can do is we can find the direction, the vector in that latent space that points in the direction of more smiles. Once we have that vector, we increase the smile of any image by doing a simple operation, mathematical operation in which the new position in

the latent space is going to be equal to the old position in the latent space plus that vector, that direction that points towards the smile, multiplied by a number that is gonna be the intensity of that transformation.

$$Z - new = z - old + i * vector - smiles.$$

So we take an image without a smile, pass it, map it to the latent space, and then move it toward the smile vector with the intensity that we want to give the image more or less a smile.

Another example is morphing between images. We can take two pictures and morph from one to another. We can take a photo of a man and morph it into a photo of a woman or a man without glasses to an image of a man with glasses. So again, this is very simple. We take two pictures, and we map them to the latent space. Then, mathematically, we calculate them using the latent space. So, we decode and map back to the high-dimensional space to obtain the corresponding images of the morphing process if P is the position in the latent space if P1 and P2 are the positions of the beginning and the end of the morphing process, Alpha will be a number from one to 0 indicating the amount of interpolation between P1 and P2. Then P new, the new position in the latent space will be the initial position multiplied by one minus alpha plus the ending position multiplied by alpha. And we're gonna be changing multiplied by alpha from zero to one.

$$P - new = p1 * (1 - alpha) + p2 * alpha$$

To morph, we will move alpha from zero to one, 0.1, 0.2, 0.3. Whenever we obtain a P point in the latent space, we map it back to the high-dimensional space to get the new image. When we do this multiple times from zero to one with the alpha from zero to one, we will obtain a set of images that will morph from the initial to the final image.

1.5 GANS: Generative Adversarial Networks.

So, we continue approaching the moment we begin working with these revolutionary architectures that have evolved and added new variations over the years. There are a lot of different generative architectures, from the variational autoencoders (VAE) to the transformers and the CLIP model. We will be working here with the most popular and successful ones so far, the generative adversarial networks with many variations. What GANs do is that through the interaction between two parts, a generative and discriminative part, they learn the latent space of, again, a probabilistic distribution.[11] So there is this battle, this fight between a part that generates and a part that discriminates. And through training in this fight, we learn latent space. This learning allows them to generate and invent new elements corresponding to that probabilistic space but differs from the training set.

[11] Probability Distribution is a key concept in machine learning, data engineering, and artificial intelligence

So, what is the generator doing? It will continuously attempt to convert random noise into images that appear from the original dataset. Conversely, the discriminator is typing to predict if the images it sees are from the actual dataset or if they are fakes created by the generator. On the one hand, the generator takes random noises in the input and tries to invent a fake image that looks convincing and as similar as possible to the original training data as the actual images. We give the discriminator two types of input. We are providing actual images from the exact original dataset, such as people's faces. Then, we also gave it the fake images generated by the generator. The discriminator must predict whether the images it sees belong to real or fake images.

Therefore, we are going to be producing two types of loss. What are the loss values? The loss values are the results of the mathematical function that evaluates how far we are from a perfect result of our network. So, it evaluates the difference between our ideal and current results. That is the loss value. We have two loss values, one for the discriminator and one for the generator, because they have different objectives. The generator will fool the discriminator into thinking the fake image is actual. Conversely, the discriminator has a different purpose. The discriminator will always try to predict correctly that the real is accurate and the fake photos are fake. They both have various objectives so they will have different loss values. So again, the generator is trying to learn to create fakes that look real, and the discriminator is trying to separate what is fake from what is real. The discriminator is trying to learn to separate, to discriminate between real and fake. There are so many applications of generative adversarial networks.

We can translate between different styles of images; we can make a super-resolution of images; we can edit pictures with GauGAN[12]; we can convert text phrases to visual elements; we can convert photographs to sketches, etc. There is a website called www.thispersondoesnotexit.com where we can see pictures and portraits that don’t exist. They have been generated by a GAN network, where these networks will not replace the human creatives; they will complement the creatives. There is an example of another type of generative work called CYCLEGAN[13]. CYCLEGAN is a type of style transfer. So you give it a set of images that could be paintings, and they do not have to be paired. What appears in the photographs and the paintings doesn't have to be the same thing, but this architecture will learn to transform one style into another. So, the photographs in the paintings or the paintings into the pictures, both ways. There are a lot of different variations. There are also generative adversarial networks that can learn to understand the structure, basically the 3D positioning of the features, so that we can use it to reconstruct from a picture movement in 3D. They have used it to do enjoyable things like taking the Mona Lisa of Davinci and making it speak and move in 3D. Now, there are challenges with these

[12] GauGAN, an AI demo for photorealistic image generation, allows anyone to create stunning landscapes using generative adversarial networks.
[13] The Cycle Generative Adversarial Network, or CycleGAN, is an approach to training a deep convolutional neural network for image-to-image translation tasks.

models. Some deepfakes [14]create confusion, and we could have problems understanding what is accurate and invalid, spam issues, ethics issues, etc.

Still, there are so many different benefits because the combination of the human and artificial perspectives will improve many of our creative possibilities. We will be able to generate more innovative and personalised predictions and recommendations. We are already seeing new markets, open-source creativity arise, and the rise of NFTs[15] in the blockchain. We're going to see a lot of combinations of these generative AI with evolutionary algorithms for the design of all sorts of things: new molecules in medicine, 3D printing, etc. So, we are redefining the connection between creation, production and consumption. Inspiration - generation - exploration - manufacturing cycles are faster, more exploratory, creative and iterative. This also creates benefits for accessibility, where more people can access the capacity to invent and generate new things. The generation cost will decrease dramatically with all the open-source frameworks appearing in this area. Many new datasets are also being provided for free for generative AI, and the time required to create elements will be significantly reduced. Generative AI allows us to experiment with thousands or hundreds of thousands or millions of possibilities quickly. Again, we combine what was learned on the analytical side of things with the more entropic part that is the basis of our nature.

Data was the problem a few decades ago, and one reason for the explosion of deep learning is the availability of more and more data. Generative AI allows us to invent and generate synthetically new data, accelerating even more possibilities. We can also apply generative AI to the typical visual field and acoustic fields, music, and music generation. And again, as we said before, combining evolving algorithms, evolving generative AI, and design will increase the possibilities. We will be able to mimic the process of natural evolution, giving birth to new ideas and shapes with this generative AI that will not be static. It will evolve through multiple generations of these processes, and this will be further improved and emphasised with the convergence of generative AI and reinforcement learning[16], which has been applied to games like Game of Chess or other types of games. As we combine reinforcement learning with generative AI, the possibilities go way beyond because we can combine more powerful ways of exploration, exploitation, divergence and convergence to apply new generative evolution products, shapes, bodies, minds and much more. We can see examples of tools that use these evolutionary processes on the internet, like Ganbreeder[17], which combines evolutionary algorithms. This generative AI is a collaborative tool that uses these gang networks to experiment, discover, and produce elements that can be mixed between them to evolve new possibilities. As mentioned, we have spoken about the new markets that appear to be the crypto art markets with NFTs. This

[14] A video of a person in which their face or body has been digitally altered so that they appear to be someone else, typically used maliciously or to spread false information.
[15] NFTs can really be anything digital (such as drawings, music, your brain downloaded and turned into an AI).
[16]Agents who learn to optimize a target in an environment through trial and failure.
[17] Ganbreeder is a collaborative art tool for discovering images using BigGAN. Using the web app, you can mix or "breed" neural networks with each other into evocative, otherworldly imagery. The results can then become physical art.

is a way of introducing scarcity and value to art in digital form. With these unique signatures, crypto signatures are added to the digital files and written on the blockchain.

1.6 Examples of reinforcement learning and generative AI

So, in reinforcement learning, we have an agent that optimises a target in an environment through trial and error, chasing rewards. So what happens if we are trying to drive a car through a road within an environment? We can do that trial and error in an environment that we can produce differently. But if we involve generative AI, we can do the trial and error in an environment that is a dream, a hallucination created by a generative AI model. An example of generative medicine is Insilico Medicine. This company combines generative AI with reinforcement learning to design new molecules that are applied to treat specific diseases. They use a type of generative AI called a variational encoder that can map the space of chemical molecules to a latent space, learning the probability distribution of the essence of that chemical molecule to a latent space and the probability distribution of the essence of those chemical molecules. Then, it can reconstruct, invent and generate new types of chemical molecules. But now, what is the challenge? If we were just randomly inventing new molecules, it could be challenging to invent new molecules that could be especially useful for a specific disease. But what do we do? We involve them and their reinforcement learning to constrain the process and ensure that the molecules generated by the creative AI only fulfil specific requirements and constraints. So, the combination of reinforcement learning and generative AI produces molecules that achieve the objective in only 40 or 50 days compared to the traditional process, which could take two years or more. This is a beautiful example of combining generative AI with other AI and deep learning parts.

As humans who involve a generative model, we can pay attention to different parts of the information and the world around us and form representations of concepts and shapes in a latent space expressed through our neurons. A model that seeks to minimise the divergence, the entropy between the information we receive and what we expect.

Chapter 2

2.1 Coding a Basic Generative Architecture from Scratch in Python + Pytorch

We will code a GAN architecture that will learn the probabilistic space of images of numbers so that we can use it to generate brand-new shapes and pictures of those numbers. Understanding the algorithmic and mathematical operations involved in the architecture training- we will code the architecture of the discriminator and then train them. And the training of GAN consists in training the generator and the discriminator.

How do we train the generator? We begin with z. Z represents a noise vector, a mathematical vector of random numbers and noise. We are going to give it an input into the generator architecture. The generator architecture will produce an output that we will call fake X because the production of the image is fake. It doesn't belong to the original, accurate training data, and we want to use it to fool the discriminator. So, we give that fake X as an input to the discriminator. The discriminator produces an output of Y, and we use a mathematical function to calculate our loss value. And that is going to be a value, the loss, that expresses how good the generator's performance is and how close the results are to the ideal results we want to obtain. What are the ideal results we want to obtain with the generator? The generator seeks to fool the discriminator; It means the generator intends to convince the discriminator that the image, the fake, X is real. We are using the value one to represent the ideal value of an actual image and the value zero to represent the ideal value of a fake image.
So when we pass it through the discriminator we will obtain a 0.1, 0.7, a 0.9, a 0.3 depends. That is going to be the value Y, and the output of the discriminator may be a 0.1, a 0.6, a 0.9 or a 0.4. When we enter that into the loss mathematical function that is going to compare two things - it's going to compare that Y, the output of the discriminator versus the number one, because the objective of the generator is to fool the discriminator and that the output of the generator is real. We want the discriminator to output one that is real. Next, we will compare the output of the discriminator to the value one. We will see the details of that comparison, which will tell us our loss value. We will use this value in a chain rule of calculus and the back propagation algorithm[18], which is done automatically by PyTorch. We will use that to update the internal parameters of the generator network, which we will keep repeating the same thing repeatedly. And that loss value is going to keep decreasing. So, the difference between the output of the discriminator and the value one will be smaller and smaller. The loss is decreasing, and the generator's performance will keep improving. In each of those cycles, we will keep updating the generator's parameters, thanks to the loss value that is back propagated to those parameters. That is the training of the generator.

[18] In machine learning, backpropagation is an effective algorithm used to train artificial neural networks, especially in feed-forward neural networks.

Now, for the discriminator training, we do the following: We take a Z noise vector and pass it through the generator. The generator produces a fake output - fake X. Then we also take a set of real images from the original dataset and input both to the discriminator, which will receive both fake and real images. That will produce outputs, the output of Y, which could be 0.1, 0.9, 0.5, etc (one represents the ideal value for a real image and zero for a fake image). We will make a couple of comparisons; we will compare the output of the processing of the fake inputs to the number zero and the output of the processing of the real inputs to the output one. Why? The discriminator wants to be able to predict that the fake inputs are fake, therefore zero, and the real inputs are real, therefore one. That's why we compare the outputs of processing the fake inputs to the value zero because we want them to be predicted as fake, as zero and the outputs of processing the real inputs to the value one, which represents real because we want the discriminator to predict that the real inputs are real. Once the training is finished, we can sample from those latent spaces to pick new points of new Z vectors in the latent space and pass them through the generator to generate brand-new outputs.

2.2 Understanding Cross Entropy in Depth.

We will discuss in detail how the loss and the performance value of the discriminator and generator are calculated in each process. To calculate the performance of the discriminator and generator, we are going to use something called cross-entropy, which is the measure of the difference between two probability distributions, between the probability distribution that the generator and discriminator are currently producing through the training process and the ideal ones that we want to produce. We are going to go from information to entropy to cross-entropy.
Starting with information - information is the number of bits required to encode and transmit an event. So an event, an occurrence, we want to encode with bits. How many bits do we need? That's what it is. Low-probability events have more information because they are more surprising. If they have a lower probability, they are more surprising, therefore you need more bits to encode them. High-probability events have less information, they are less surprising because you need fewer bits to encode them. We can express this with this equation, H of X, which is what we use to represent information, is equal minus the logarithm of the probability of X.

$$h(x) = -log(P(x))$$

Let's see an example. Imagine we have an event with a low probability. We have a low probability event, from zero to one. Let's say, for example, 0.1 - the logarithm of 0.1 is -1. When we use the above equation, a logarithm of 0,1 is -1, with a minus in front of, equal 1. So the information of a low probability event is 1, which is a high number. So therefore a low probability event has more information and is more surprising. Now, the opposite, a high probability event and we take an example the logarithm of 0.99 or 0.9 which is minus 0.045. With the minus in front when we put it in the equation it's 0.045.

So the information of a high probability event gave us 0.045 and the information of a low probability event gave us 1. This demonstrates why we use this equation because the logarithm with the probability and with a minus in front of it allows us to produce a high value for low probabilities and a low value for high probabilities.

Continuing with entropy, it is a beautiful concept. All of these concepts, information and entropy, are used throughout artificial intelligence and machine learning. Entropy is the number of bits required to represent a randomly selected event from a probability distribution. So we have a probability distribution and we randomly select an event. So what is the number of bits required to represent one of those randomly selected events? And that's what happens in the following. In entropy, we have skewed distribution, which means the distribution of the different probabilities is not uniform, it is the opposite of uniform. So for example, we say an event has a probability of 0.9 and another event has a probability of 0.05 and another or 0.05. First of all the equation of entropy is which is represented by a capital H of X minus the sum of the product, the probability of each of the occurrences by the logarithm of these probabilities. So it is the sum from 1 to N to the product of the probabilities, times the logarithm of the probability.

$$H(X) = -\Sigma P(x_i) log2\ (P(x_i))$$

Let us see an example of a skewed distribution again with a skewed distribution we imagine that there are three events and one of them has a probability of 0.9, another one of 0.05 and another one of 0.05. This is very skewed because one of the events has a very high probability and the other two very low probability. The result of the computation would be 0.9 multiplied by the logarithm of 0.9, which is minus 0,045. Plus 0.05, which is the second event times the logarithm of 0.05, which is minus 1.3, plus 0.05 times the logarithm of 0.05, which is again -1.3, Now if we multiply and sum all of this we obtain 0.17, which is a low value of entropy. So skewed distribution has low entropy because it is unsurprising. Now what happens in the opposite case? A uniform distribution[19]. So we imagine a uniform distribution where there are three types of events and all of them have the same probability, 0.33. So the result will be 0.33 times the logarithm of a 0.33, which is minus 0.48, adding also the other two 0.33 times the logarithms, as explained above. The result is 0.47, a higher value. So the uniform distribution has a larger entropy and it's surprising because we are not sure which one of them is going to happen since all of them have the same probability.

Moving on to cross entropy, it's called cross entropy because we are going to compare two distributions and then we are going to use the entropy to compare them. So the cross entropy is the number of bits required to represent an average event from one distribution compared to another distribution. We have P that represents the target, the distribution, so imagine if we are looking at the generator of the discriminator, this is the target and the ideal distribution that we

[19] The Uniform distribution is used to represent a random variable with constant likelihood of being in any small interval between min and max.

want to obtain. That is P, and Q is the approximation of P. Q is the distribution that we are obtaining through the training process and the cross entropy is going to give us the number of bits required to represent an average event from one of those distributions compared to the other. So cross-entropy is going to give us the number of extra bits that we need to represent an event using that approximation[20] that we're learning instead of the target, P. The equation is going to be

$$H(P,\ Q)\ =\ -\Sigma\ x\ in\ X\ P(x_i)\ log(Q(x_i))$$

2.3 Understanding equations to calculate discriminator and generator loss

Now we are going to use the binary cross-entropy loss equation. Why is it binary cross entropy? Because the discriminator wants to predict two things, the real images that are real and the fake images that are fake and that's why the equation has two parts.

$$BCELoss\ =-\ 1/n\Sigma\ (y_i log(y\hat{}_i)\ +\ [(1\ -\ y_i)log(1\ -\ y\hat{}_i))$$

The y represents the target probability, the target label in this case, that is one for real and zero for fake. The y-hat represents the prediction, the discriminator's current output, and the approximating probability. So again, the target probability times the logarithm of the approximating probability. These are the two parts, one more focused on the real images, one more focused on the fake images. When we are evaluating the real images we are using the logarithm of the prediction of the discriminator and when we are evaluating the fake images it's going to be a logarithm of one minus the output of the discriminator applied to the output of the generator. That happens because in this case, we are using fake images, so the output of the discriminator is using the processing that comes from using the input that was the output of the generator applied to the original z noise vector. This is also what we call a minimax game in which the discriminator is trying to minimise the equation and the generator is trying to maximise it.

$$-\ 1/n\Sigma\ (logD(x)\ +\ log(1\ -\ D(G(z)))$$

In this case of the discriminator, the discriminator wants the first part to output a one, because we want the real images to be predicted as real, as one, so the logarithm of one is zero plus the logarithm of one minus. And we want the prediction of the fake images to be fake, so we want the $D(G(z)$ to be zero. One minus zero is one and the logarithm of one is zero. So all of this equation, the ideal for the discriminator is that it goes towards zero. We want to minimise it and in the case of the generator, it wants to fool the discriminator. So in the case of the real images, it

[20] Data approximation is the process of calculating approximate outcomes from existing data using mathematically sound methods

doesn't want the discriminator to output a one, but something much lower that foes towards zero. And the logarithm that goes towards zero is going to be a very large value. The generator wants the discriminator to be fooled and to think that the fake image is real so that this is one or towards one or 0.9. This is a battle, a fight where the discriminator wants to minimise the equation and the generator wants to maximise it which becomes a beautifully trained network.

Before coding, we are going to talk about the generator loss which is much easier than the discriminator loss because in the generator loss, we don't have to deal with both fake and real images but only with the fake images. We only have to deal with one of the cases and with one of the labels because we want to convince the discriminator that the fake images are real, therefore the only label that we are going to use is one that represents real. So when we use the label one the second part of the binary cross entropy equation is going to disappear and the whole equation is going to just become the logarithm of the prediction.

$$\text{When the label is real} = log(y\hat{}_i)$$

So the calculator of the loss of the generator is simple, the output of the discriminator is applied to the output of the generator because we are dealing with fake images. So we want to convince the discriminator that the fake image is real so the output of the discriminator should be a real one and the logarithm is going to be zero. So the generator is going to be trying to minimise, to push this curation as much as possible towards 0, minimising the curation.

2.4. A basic GAN architecture

2.4.1 Importing libraries and declaring a visualization function.

We will build a basic GAN and make a functional GAN architecture using Google Colab and Jupyter Notebook. The first thing we will do is import the libraries that will help us build the GAN quickly.

```
# import the libraries
import torch, pdb
import torch.utils
from torch.utils.data import DataLoader
from torch import nn
from torchvision import transforms
from torchvision.datasets import MNIST
from torchvision.utils import make_grid
from tqdm.auto import tqdm
```

```
import matplotlib.pyplot as plt
```

DataLoader is a part of a library of PyTorch that is going to allow us to build an iterable that is going to hold all of our training data and we will be able to iterate through the training data as we train the network. Now from torch import nn which is a part of PyTorch library that is going to allow us to build very quickly deep learning architectures and models. From torchvision, that is a part of PyTorch that deals with computer vision, we import transforms. That will allow us to transform our training data in different ways. From torchvision.utils import make_grid we are going to use make_grid to build a grid of images and to evaluate during the training, how the fake images created by the generator are looking versus the real images. And then we import matplotlib.pyplot as plt, where matplot is going to allow us to plot that image with all the visuals that we want.

Next, we create a function to be able to create a function to display as we are training. We want to show a grid with several generated images versus the real images.

```
# visualization function
def show(tensor, ch=1, size=(28,28), num = 16):
 #tensor : 128 x 784
 data=tensor.detach().cpu().view(-1,ch,*size) #128 x 1 x 28 x 28
 grid = make_grid(data[:num], nrow=4).permute(1,2,0) # 1 x 28 x 28 = 28 x 28 x 1
 plt,imshow(grid)
 plt.show()
```

A tensor is like a multidimensional array. That is the first parameter, tensor and the second is the number of channels. We are going to be using black-and-white images of numbers so there is only one channel, a gray scale. The size is going to be in our case 28 because the number of images of the MNIST dataset has 28 pixels of width times 28 pixels of height. Next, we want to make a grid of 16 images, which is four by four. First, we are going to apply the detach function. The detach function is going to detach that variable from the computation of the gradients. So when we are using PyTorch, different backpropagation functions[21], the gradients are being calculated when they are linked to different variables. Now we just want to visualize what is contained in that variable. We don't want any processing to take place with the gradients, so we detach it from all the gradient computations and pass it to the CPU because during the training we will be using the graphic card. And then we are going to restructure it in the following way: -1, ch, *size. What this is doing that if you start from the end, the end says size, so that means 28

[21] Backpropagation, or backward propagation of errors, is an algorithm that is designed to test for errors working back from output nodes to input nodes. It's an important mathematical tool for improving the accuracy of predictions in data mining and machine learning.

x 28. That is going to be the end of the reshaping that we are doing because the function view is going to reshape the size of the tensor. The tensor is going to have the dimensions of 128(batch size of 128, which means we are going to be processing 128 pictures at a time) times 784 (28 times 28) and we are going to reshape this structure back to 28 times 28. And then -1 means whatever remains after that is going to be the first dimension. The next thing we are going to create is the grid, so we are going to use the function make_grid and we are going to take some elements of that data structure, in this case, the first 16 elements of this multidimensional tensor that has 128 and we are going to put the second parameter of this function that his the number of rows and set it as four because we want to use 16 pictures. Now the function make_grid is going to give us a series of images that have this internal structure, 1 times 28 times 28. But if we want to display this with matplotlib the order of the channels has to be different because PyTorch is using channel and then width and height but we want to have for matplotlib width, height and then channel. In this case, we declared a visualization function that we are going to use later to visualize the aspect, the content of images generated by the generator during the training process.

2.4.2 Hyperparameters and the Data Loader

It is very important to understand what is the shape and the dimensions of each data structure, so we are now continuing with the setup of the main parameters and hyper-parameters.

```
# setup of main parameters and hyperparameter
epochs = 500
cur_step = 0
info_step = 300
mean_generator_loss = 0
mean_disc_loss = 0

z_dim = 64
lr = 0.0001
loss_func = nn.BCEWithLogitsLoss()

bs = 128
device = 'cuda'

dataloader = DataLoader(MNIST('.',download=True, transform=transforms.ToTensor()), shuffle
= True, batch_size=bs)

# number of steps = 60000 / 128 = 468.75
```

The number of cycles that we are going to perform during the training is 500, so epchos is going to be equal to 300. In each of the steps of the training, we are going to process one batch, one set of images, so the current step is going to start in zero. The info step is going to store how many steps we want to show in the screen information about the current loss values. Each step processes a batch and how many steps are we going to show the information on screen. Every 300 steps we have to accumulate that information, the generator and calculator loss and calculator the average of them. We are going to have a couple of variables, the mean generator, that is zero at the beginning, and the mean discriminator loss, that will be zero at the beginning. Some hyper-parameters, the dimensionality of the noise vector, that is the input of the generator, of what is going to be the dimensionality of that latent space is the input of the generator, which in this case is 64.

Learning rate is what is going to be the speed at which we are going to tweak the parameters of the neural network, pushing them in the direction of the negative gradient. Loss function is going to be the mathematical function that is gonna be used to calculate the loss values. PyTorch hass two possibilities, binary cross entropy, BCE loss or BCE with logits loss. The difference is that before we calculate the loss, in the output of the discriminator we want to convert the numerical values to the range that goes from zero to one. So the function BCEwithlogirtaLoss is going to take the output of the neural network that are the logits[22]. Next with batch size, it is that during each step of the training how many images are we going to process at once in the GPU? In the greater advantage of GPU is that we can send a number of images in parallel to be processed to the GPU, where in this example we are going to send 128. That is going to be our batch size. Next is going to be the device where we are going to process the data and we want to process it in the graphic card in the GPU so that the data can be processed in parallel much faster, so we set here cuda. And then finally, we are going to declare our data loader. The data loader is going to be a structure using the function dataloader from PyTorch, which holds our training data and it is going to be a type of structure called an iterator that we will be able to call. And then we will get batches, which we will give us back a batch of the training data that will contain a set of input elements for the network and a set of labels for those elements. PyTorch has a function called MNIST that gives us the data by itself. First parameter is where we do want to store the data and we ask PyTorch to download the data for us. The last line of code is us asking the PyTorch, after we download the data to put it in the root folder and transform it according to this function. Depending on the data we have, we have to do different transformations where in our case all we need to do is transform it into the format of a multi-dimensional tensor. After declaring the data is going to be on shuffle, which means at every epoch we are going to reshuffle the data and change the order which will help with the training process. The MNIST dataset has 60.000 images and divided by the number of images per batch we get 468.75, meaning there are going to be 469 steps in every epoch.

[22] Logits are the outputs of a neural network before the activation function is applied.

2.4.3 Generator and Discriminator Class

Now we can declare our models and begin with our generator.

```
# declare our models
# generator
def genBlock(Inp, out):
  return nn.Sequential(
      nn.Linear(inp, out),
      nn.BatchNorm1d(out),
      nn.ReLu(inplace = True),
  )

class Generator(nn.Module):
  def _init_ (self, z_dim=64, i_dim=784, h_dim=128):
    super()._init_()
    self.gen = nn.Sequential(
        genBlock(z_dim, h_dim), # 64, 128
        genBlock(h_dim, h_dim*2), # 128, 256
        genBlock(h_dim*2, h_dim*4), # 256, 512
        genBlock(h_dim*4, h_dim*8), # 512, 1024
        nn.Linear(h_dim*8, i_dim) # 1024, 784 (28 x 28)
        nn.Sigmoid()
    )

    def forward(self, noise):
      return self.gen(noise)

def gen_noise(number, z_dim):
  return torch.randn(number, z_dim).to(device)
```

We are going to structure the generator in different blocks and also disciminator. We define a function of the generator block, with the size of the input, output and we return an nn.sequential which allows us to set a number of blocks or elements that are going to be executed sequentially. The first one is going to be on nn.linear layer that is going to perform linear computation between the input and the output. Those linear computations involve parameters, in this case input and output size. After we are going to execute nn.BatchNorm1d, where we use 1d because we are using black and white, one dimensional images with a single channel. We are going to

apply non-linearity, where one of those non-linear functions is the value function. The value function is a function that returns whatever arrives as a negative value and sets that to zero. Whatever is positive, it lets it pass and the value that it has. This helps the neural network to learn more complex functions.

Now we got the block where we declare the class. The class of the generator that is child of nn.Module and we declare the init function with the parameter, the size of the input noise, latent vector, that is going to be 64 by default. The generator is going to output an image, where the size of the image is going to be 784, because 28 x 28. That is the size of the MNIST dataset. And then we declare the size of the first hidden layer of the generator which is 128. We declare a variable which is going to be called gen of generator. We create four nn.sequential structures that the network can learn. The first one is going to take as input the size of the input noise latent vector and the output is the base size of the hidden layer, moving to the other structures we are going to increase the size until we arrive to generate an image of the size of the images in the MNIST dataset. For the output we create a single linear later, a fully connected layer that we go from that previous output h_dim*8 to the size of whatever we want to generate in the output of the generator. That is going to go from 1024 to 784, where 784 is 28 times 28. Next we want to tell the generator what we want the values of the pixels of the image to be between zero and one so we add an nn.Sigmoid function. We declare our forward function which is going to be executed when we run the instance of the class and that is going to have a parameter that is the noise vector. All we need to finish is a function that generates noise. It is going to have as a parameter the number of basically noise vectors that we want , which is z_dim, and it is going to return the PyTorch function randn. Randn is function that is going to return a tensor field with random numbers from a normal distribution and this has parameters, the number of noise vectors and the size of the noise vector, where after we store it in the GPU.

```
class Discriminator(nn.Module):
 def _init_ (self, i_dim = 784, h_dim = 256):
  super()._init_()
  self.disc=nn.Sequential(
     discBlock(i_dim, h_dim*4), # 784, 1024
     discBlock(i_dim*4, h_dim*2),  # 1024, 512
     discBlock(i_dim*2, h_dim), # 512 256
     nn.Linear(h_dim, 1) # 256 1
  )

  def forward(self, image):
   return self.disc(image)
```

```
def gen_noise(number, z_dim):
 return torch.randn(number, z_dim).to(device)
```

For the discriminator class, it is the same as the generator class. Composed of a linear layer, input size to output size and in this case a non-linearity instead if a ReLU[23], so that the disciminator can learn complex mapping. So when we instantiate a variable of the class generator, and when we call that instance, we are going to call it with a noise vector. And the generator class is going to return the result of passing that noise vector through all these nn.Sequential data structures through this model. And to produce the noise vector, we are going to call it gen noise function that is going to produce a set of random numbers from a normal probability distribution that is going to initialize these noise vectors. For the discriminator, the input is going to be an image, therefore we want to know if the image is real or fake. The output is going to go towards one if it is a real image or zero if it is a fake image. And when we instantiate the discriminator variable with the input of an image the discriminator class is going to return their result of passing that image through all of this sequential model in discriminator class.

2.4.4 The optimizer and testing the generator

```
gen = Generator(z_dim).to(device)
gen_opt = torch.optim.Adam(gen.parameters(), lr=lr)
disc = Discriminator().to(device)
disc_opt = torch.optim.Adam(disc.parameters(), lr=lr)
```

Now that we have the data structures for the generator and discriminator we can declare them, we declare the variable gen for the generator as an instance of the generator class with the parameter, the size of the noise vector that is z_dim and we store it into the device that is the GPU. Declaring the optimiser next, which is going to be the charge of the optimisation, such as calculating the gradients with the backpropagation and then tweaking and updating all the parameters of both the generator and discriminator. By applying the correct learning rate and updating those parameters with the correct algorithm, we use PyTorch as a function that allows us to apply different optimizers such as Adam. When we execute the code, we see the structure that we have declared where we track the statistical characteristics of the batch during the whole training process to get the mean and the variance during the entire training process instead of focusing on the statistics of the current input batch. We have affine, where we use learnable affine parameters, that is something internal to BatchNorm, which normalizes the data and it also has some learnable parameters that can allow it to adapt to the data dynamically. And then the momentum, which is the value that is used for the running mean and the running bar

[23] Instead of giving zero to the negative numbers, is going to giv them a small negative value on a slope.

computation, which means that during the training we are calculating batch by batch. During the inference, we are going to be using running averages that are going to be impacted by this momentum.

```
# Fetch a batch of data
x, y = next(iter(dataloader))
print(x.shape, y.shape)
print(y[:10])

noise = gen_noise(bs, z_dim)
fake = gen(noise)
show(fake)
```

Now we have some fun with the iterator. In the first line we are saying take the data loader structure, cast it as an iterator and give us the next batch. So the dataloader is going to give us a whole batch of 128 images with one channel each, because its black and white and each image has a 28 times 28 pixels, 784 pixels in total and then 128 labels and each labe has a number that says what is the number inside each of the images. Now testing the visualization function, we get the noise vector with gen_noise, the number of noise vectors, bs size and the size of the noise vector, z_dim. We pass the noise vector through the generator model and show those fake images.

2.4.5 Loss values of generator and discriminator

```
# calculating the loss

#generator loss
def calc_gen_loss(loss_func, gen, disc, number, z_dim):
  noise - gen_noise(number,z_dim)
  fake = gen(noise)
  pred = disc(fake)
  targets=torch.ones_like(pred)
  gen_loss=loss_func(pred,targets)

def calc_disc_loss(loss_func, gen, disc, number, real, z_dim):
  noise = gen_noise(number, z_dim)
  fake = gen(noise)
  disc_fake = disc(fake.detach())
```

```
disc_fake_targets=torch.zeros_like(disc_fake)
disc_fake_loss=loss_func(disc_fake, disc_fake_target)

disc_real = disc(real)
disc_real_targets=torch.ones_like(disc_real)
disc_real_loss=loss_func(disc_real, disc_real_target)

disc_loss=(disc_fake_loss+disc_real_loss)/2

return disc_loss
```

We declare a function of calc_generator_loss and as parameters we will have the loss function we want to use, the generator model, the discriminator model, the number of elements that we want to processand the dimensions of the input latent vector. We obtain a noise vector, that is going to be the input of the generator. We pass that noise vector through the generator class instance and we are going to store the result in a variable called fake. We take that fake output of the generator, pass it through the discriminator and store the results in a variable called prediction. Now we are going to compare to calculate the generator loss and we are going to apply the loss function. We are going to compare two things: the output of the discriminator to the targets. What are the targets? The generator wants to fool the discriminator that the fake images are real and real means one, so the targets is going to be a value of ones. We use the PyTorch function and what its going to do is create a tensor with the dimensionality similar to the prediction and fill it with ones. That represents the real image because we want to push the generator to fool the discriminator so the discriminator thinks that the fake images are true.

The same thing is done with the discriminator. Real is going to be a set of real images, because the discriminator has to receive as input both fake images and also real images. So we need a set of real images that we are going to send as a parameter to the calc_disc_loss function and then the dimensionality of the noise vector. We get a noise, a set of noise vectors with gen_noise. We pass those noise vectors through the generator and we pass those fake images generated by the generator through the discriminator. Something very important here is the detach, because when PyTorch back propagates this loss, the loss of the discriminator, to calculate the gradients and tweak the parameters, we dont want to change the parameters of the generator when we are optimizing the discriminator. So in this case, we only want to optimize and tweak the parameters of the discriminator, that’s why we are going to detach the fake images from the calculator of the gradients. Loss applied to the fake images is going to be the loss, the result of applying the loss function, to compare the output of the discriminator applied to the fake images (targets). Fake

images are zero, therefore we are going to use torch.zeros_like(disc_fake) which is going to produce a tensor with the same dimensionality than the predictions of the fake images but full of zeros. Now we do the same thing with the real images, so disc_real passes the real images that were passed through this function through the discriminator and then the targets in this case are going to be one. One because we want the real images to be identified as real. Disc_real_loss is going to be the loss function that is going to compare the output of the discriminator applied to the real images, with the targets that are going to be one, because we want the discriminator to learn to identify the real images are real. And finally to get the discriminator loss, we just have tot ake the average of both and we return it.

2.4.6 Main training loop: discriminator, generator and stats

```
# Initialize the variables
mean_disc_loss = 0
mean_gen_loss = 0
cur_step = 0

for epoch in range(epochs):
    for real, _ in tqdm(dataloader):
        #### discriminator
        disc_opt.zero_grad()

        cur_bs = len(real)  # real : 128 x 1 x 28 x 28
        real = real.view(cur_bs, -1)  # 128 x 784
        real = real.to(device)

        disc_loss = calc_disc_loss(loss_func, gen, disc, cur_bs, real, z_dim)

        disc_loss.backward(retain_graph=True)
        disc_opt.step()

        ### generator
        gen_opt.zero_grad()
        gen_loss = calc_gen_loss(loss_func, gen, disc, cur_bs, z_dim)
        gen_loss.backward(retain_graph=True)
        gen_opt.step()
```

```
        ### visualization & stats
        mean_disc_loss += disc_loss.item() / info_step
        mean_gen_loss += gen_loss.item() / info_step

        if cur_step % info_step == 0 and cur_step > 0:
            fake_noise = gen_noise(cur_bs, z_dim)
            fake = gen(fake_noise)
            show(fake)
            show(real)
            print(f"{epoch}: step{cur_step}/ Gen loss: {mean_gen_loss} / Disc loss:
{mean_disc_loss}")
            mean_gen_loss, mean_disc_loss = 0, 0
        cur_step += 1
```

We are going to do a training loop that is going to go through different epochs and each epoch is going to have a number of steps, so in this case if we have 60,000 MNIST images of numbers, divided by the size of each batch (128) we get 469, so we are going to have 169 steps in each epoch. In the beginning we have our main loop which is saying that for the information that is returned at each point by the data loaded structure, which is being wrapped by TQDM, it is going to apply a beautiful progress bar to the visualization of the process and we are going to receive a set of images that we are going to store in the variable real, because these are real images. These are images from the MNIST dataset. This is generative adversarial learning, so we only need the actual input of the images. Inside the training loop, we train the discriminator first and then the generator.

Beginning with the discriminator, we have to take the discriminator optimizer and set its gradients to zero, so at the beginning of each of the steps, we have to set the gradients to zero and then we have to detect what is the current batch size, which is going to be the length of the size of those real images that are returned in each step by the data loader. Depending on the proportion between the total number of imagesand the size of the batch, we are going to reshape those real images with view into the size of the batch and concentrate the other dimensions. We create the loss function and after the loss we call the backward function of PyTorch. The backward function is going to take that loss value and back propagate it to calculate all the gradients across the neural network. We have a parameter called retain graph, of we don't use it the graph being used to compute the gradients is going to be freed and if not, it’s going to be kept in memory to be accessed by different variables all around. After we have calculated the gradients, we are going to tweak and update the parameters of the discriminator in this case.

The generator is going to be the same thing, we are going to take the optimizer of the generator and apply the zerograd function to set the gradients back to zeero at the beginning of each step and then we are going to calculate the loss. In this case we dont need to deal with the real images. First we need to calculate the mean discriminator loss and we are going to visualize every 300 steps in our case, so we are going to accumulate it and the value that we have. This calculated item is going to transform the tensor value into a standalone number. We are going to divide it by the info step and we are going to do the same thing for the mean generator loss. What this is doing, it's going to be adding the values in proportion to the period in which we are going to visualize the information so that at the end when we visualize the information, every 300 steps we are going to see the average of the discriminator loss and the generator loss. In the end, we want to show how the fake images are looking in relation to the real ones. Lets generate some random noise, we pass it through the generator and we show some fake images and also real images. We print the information, we can format it and at the beginning put the number of the epoch and where we are.

For as many epochs as we want, we are getting watches from the dataloader and then we are for the discriminator setting the gradients to zero and looking at the size of the batch. We are reshaping the size of the real image tensor and then calculate the discriminator loss. We backpropagate it with tweak and update the parameters. We do the same for the generator. We reset the gradient, calculate the loss, backpropagate the gradients, the loss and calculate the gradients/update the parameters and then we calculate the average discriminator loss. We accumulate the discriminator and generator loss values and then calculate the average every 300 steps. In this case, we visualize some example fake images and the real images and the different values.

2.4.7 Running the training, results and conclusions

When we execute the loop, we can see that each epoch has 469 steps. At the beginning the fake images look very blurry and these are the fake and the real. Every 300 steps we can see how they begin to evolve, so we will have to wait a number of epochs and we will begin to see them improve gradually as they start to become less noisy and they begin to approach more the shapes of the numbers. Of course, this GAN is a very basic GAN, but it is enough for us to see how with little code we can train it to transform pure noise into an actual image of something, in this case, images of numbers.

Chapter 3

3.1 Multimodal generation

In this advanced material, we are going to work with two cutting-edge generative architectures that will allow us to do this multimodal generation, the future that is coming, the ability to connect different modalities, in this case, text and images, text and visual elements.

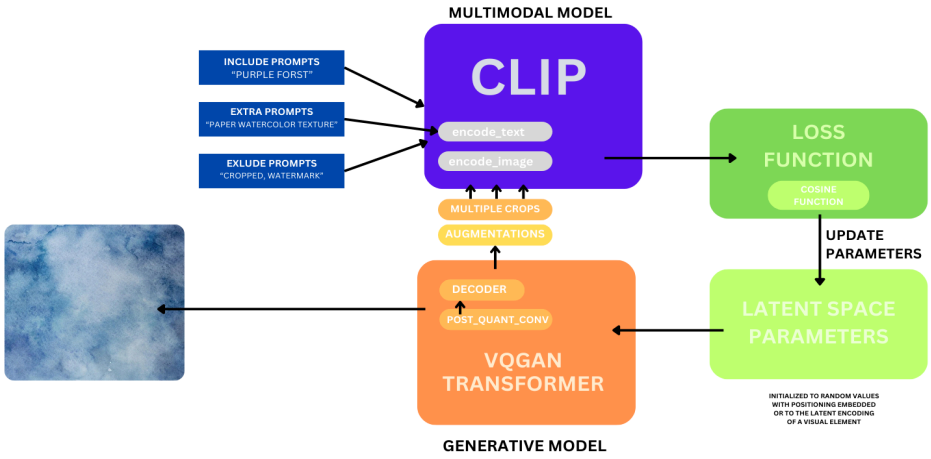

Figure 1: CLIP architecture with VQGAN transformer

First, we have the CLIP architecture by OpenAI, where a generative architecture is created, and it can connect text and images by being trained with a lot of images and text to be able to predict which image corresponds with what encoding of the text and what text encoding corresponds with what visual encoding. It creates the capability of encoding text and image elements into a common space where the network learns to connect both kinds of encoding. This is very powerful as we will use it in combination with the second architecture, a type of transformer architecture called VQGAN, which combines elements of convolutional architectures with GAN types of elements. It uses a codebook and works with patches, so instead of working with the pixels directly, they train the network to learn a sort of matrix representing parts of the image stored in this codebook. How are we going to combine these two architectures? We are going to have a text input, and on the other hand, we will have an image, and at the beginning, that image will be noise. It's going to be a random value like noise. So we have the phrase and an image that it begins with just random numbers. We continue to do an optimization process through a number of iterations, where the more iterations, the more refined the results will become. In this case, we are going to do 100 iterations. That is enough to get nice results. In each of the iterations, we are going to do the following: we are going to take that text phrase, and we are going to pass it through the CLIP architecture, encode it in 512 numbers that are going to be an encoding of that architecture, the understanding of the CLIP architecture of that text. Then we do the same with our image and our current image. First, we are going to transform the image,

augment it, rotate it, move it and create crops of it, in this case, 30 different crops of the image. We do this to help the CLIP architecture understand our image better by giving versions. So, instead of sending 30 versions of our image with different crops of a specific rotation and translation. We send those 30 crops of the image to CLIP, and CLIP is going to encode and return 30 sets of encodings of 512 values each. In the end, we compare those encodings by using a function, that is, the cosine similarity. This will help us calculate the loss value and the network's performance because our objective is to match the text's encoding and the image's encoding. When their similarity is as high as possible, it will mean that the encoding of the image will match the text. In this chapter, we will not only encode one single text prompt but also provide the capability of having multiple text prompts. So we can have an elephant on a mountain, 100 people in blue jackets, etc. Moreover, we will also have what we call the extra prompts, which we want to apply to all of the prompts we have included. The beautiful combination of architecture allows us to come up with new visual elements that we have never seen before.

3.2 Importing the libraries, helper functions and hyperparameters

The code in the first section begins by cloning two GitHub repositories: the CLIP model from OpenAI and the taming transformers from CompVis. Following this it installs several Python packages: 'ftfy', 'regex' and 'tqdm' without dependencies, specific versions of 'omegaconf' and 'pytorch-lightning' etc. It then imports a variety of libraries essential for numerical operations, learning, file operations, image processing,debugging, data visualization, etc. Additionally, it imports 'torch-vision' along with its transformation submodules for handling image data and configuration management tools' yaml' and 'omega conf'. Finally, the 'clip' module from the cloned CLIP repository is imported, preparing the environment to work with these models and tools.

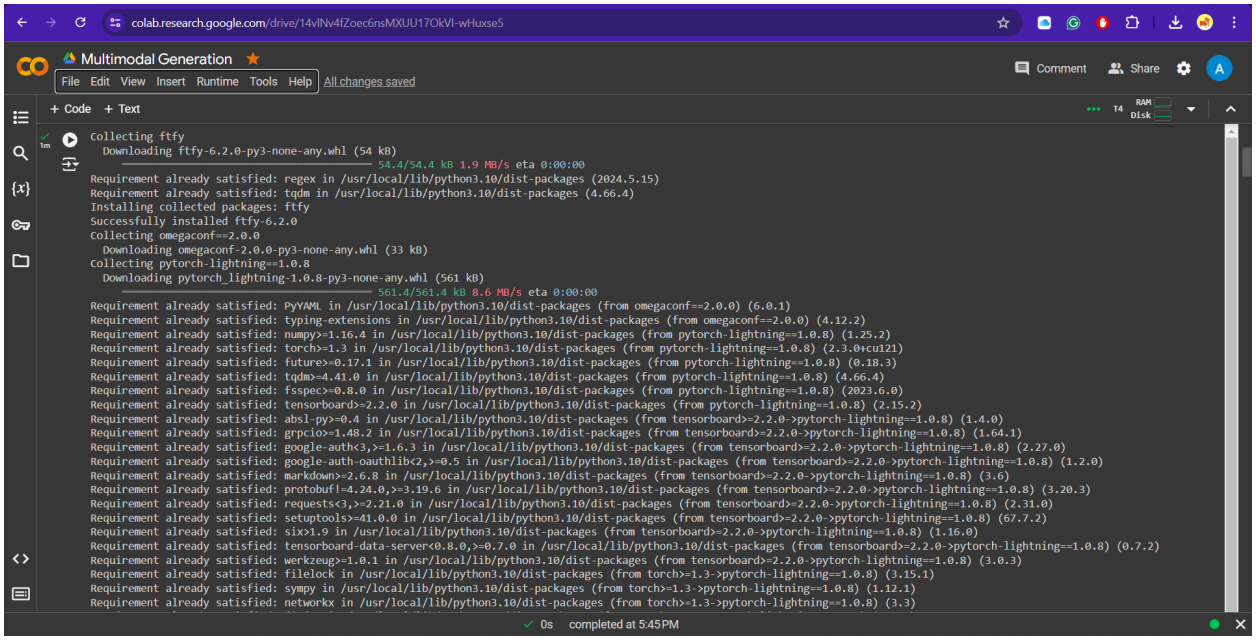

Figure 1: Importing the libraries successfully

3.3 Setting up the CLIP model

```
### CLIP MODEL ###
clipmodel, _ = clip.load('ViT-B/32', jit=False)
clipmodel.eval()
print(clip.available_models())

print("Clip model visual input resolution: ", clipmodel.visual.input_resolution)

device=torch.device("cuda:0")
torch.cuda.empty_cache()
```

We load the CLIP model and set it to evaluation mode with ‘clipmodel.eval()’. It prints the list of available CLIP models and next it prints the input resolution required bu the visual component of the CLIP model.

```
['RN50', 'RN101', 'RN50x4', 'RN50x16', 'RN50x64', 'ViT-B/32', 'ViT-B/16', 'ViT-L/14', 'ViT-L/14@336px']
```

We use the rest net architecture with 50 layers or the visual transformer and then the CLIP model visual input resolution is 224 pixels. This is very important which means that when we encode images in clip we need to set them

3.4 Genrative transformer model and latent space parameters

```
%cd taming-transformers/
!mkdir -p models/vqgan_imagenet_f16_16384/checkpoints
!mkdir -p models/vqgan_imagenet_f16_16384/configs
if len(os.listdir('models/vqgan_imagenet_f16_16384/checkpoints/')) == 0:
  !wget 'https://heibox.uni-heidelberg.de/f/867b05fc8c4841768640/?dl=1' -O
'models/vqgan_imagenet_f16_16384/checkpoints/last.ckpt'
  !wget 'https://heibox.uni-heidelberg.de/f/274fb24ed38341bfa753/?dl=1' -O
'models/vqgan_imagenet_f16_16384/configs/model.yaml'
```

When we execute this, we begin to download the pre-trained VQGAN model architecture from Heidelberg, including the checkpoint of the last trained model and the configuration information for the 'model. yaml' file. Once the checkpoints are downloaded, we can instantiate the taming transformer VQGAN architecture. To do this, we are first going to import taming models.

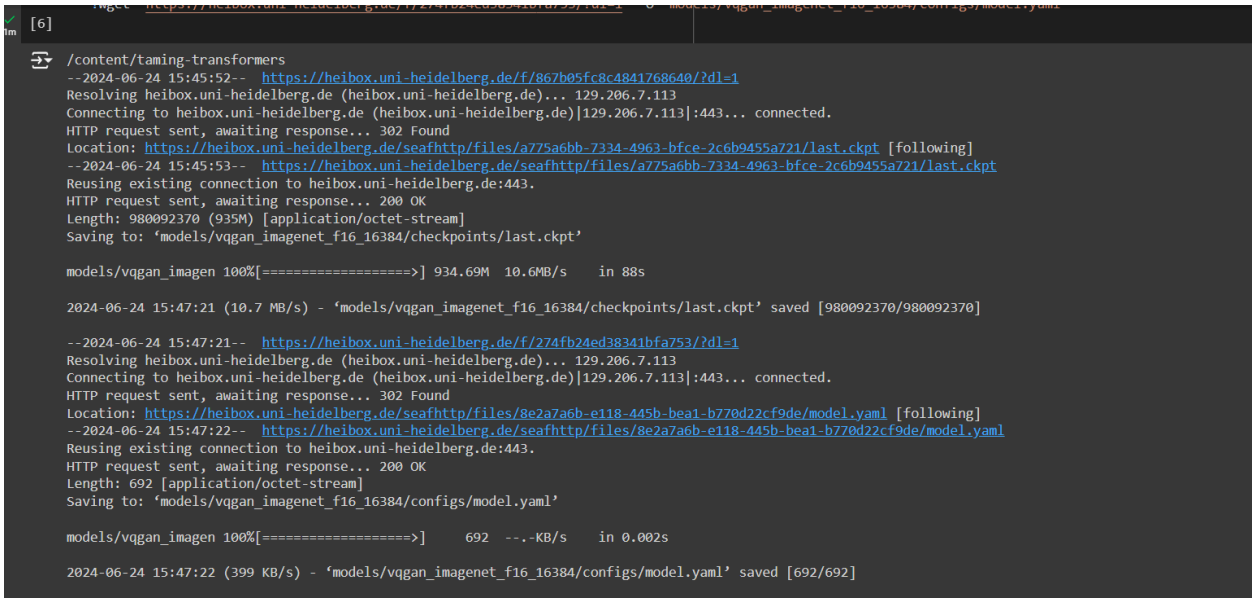

Figure 2: Taming transformer instantiation

We import the VQGAN architecture from `vqgan` as `VQModel` within the repository. We create helper functions to load configurations and the model. First, `load_config` takes a config path and an optional display flag (default `False`). It uses OmegaConf to load the configuration and, if requested, prints it using `yaml`. Next, a function to load the VQGAN model takes the configuration and checkpoint path. It instantiates the `VQModel` with the configuration. If a checkpoint path is provided, it loads the state dictionary using `torch.load` and sets the map location to CPU. The state dictionary is then loaded into the model with

`model.load_state_dict(state_dict, strict=False)`. The model is set to evaluation mode with `model.eval()`. The generator function takes an input and generates an image. The input passes through `taming_model.post_quant_conv` and the transformer's decoder, yielding the final image.

```
def load_vqgan(config, chk_path=None):
  model = VQModel(**config.model.params)
  if chk_path is not None:
    state_dict = torch.load(chk_path, map_location="cpu")["state_dict"]
    missing, unexpected = model.load_state_dict(state_dict, strict=False)
  return model.eval()
```

Within the repository's models, the VQGAN architecture is implemented, and we import the `VQModel`. We create a few helper functions to facilitate loading the configuration and the model. First, we define `load_config`, which takes a configuration path and an optional display parameter (defaulting to `False`). It uses OmegaConf to load the configuration from the path and, if the display is set to `True`, prints the configuration using the `yaml` library. The loaded configuration is stored in a container for further use. Next, a function to load the VQGAN model takes the configuration and checkpoint path. It instantiates the model by calling `VQModel` with the provided configuration. If a checkpoint path is given, it loads the state dictionary (which contains all the model parameters) using `torch.load`, setting the map location to the CPU. The state dictionary is then loaded into the model using `model.load_state_dict(state_dict, strict=False)`. The model is set to evaluation mode with `model.eval()` since it will be used for generating outputs, not training. Lastly, the generator function takes an input and generates an image. The input passes through `taming_model.post_quant_conv`, and the output is fed into the model's decoder. The generation process involves two stages: first, the convolutional network, and then the transformer's decoder. The final result, after passing through these stages, is the generated image, which is returned.

```
self.data = .5*torch.randn(batch_size, 256, size1//16, size2//16).cuda() # 1x256x14x15 (225/16, 400/16)
   self.data = torch.nn.Parameter(torch.sin(self.data))
```

After we import the VQGAN, we continue with declaring the values that we are going to optimize. We define a 'Parameters' class to initialize and optimize parameters. The class creates a random tensor, applies the sine function and wraps it as a 'torch.nn.Parameter'
The code initializes parameters with random numbers using `torch.randn` from a normal distribution. It sets the structure to meet the taming transformer VQ-GAN architecture

requirements: a batch size of 1 and 256 channels. For the desired output size of 255x400, each dimension is divided by 16 and rounded down, ensuring compatibility with transformer patch requirements. This tensor, placed on the GPU, represents random values organized as one batch, 256 channels, and dimensions derived from the division operation. Multiplying by 0.5 helps standardize initialization after random generation. Applying `torch.sin` embeds positional information across the data, which is essential in transformer architectures for context without explicit positional indicators found in earlier neural networks. Converting this processed tensor into a `nn.Parameter` optimizes its values during training.

```
optimizer = torch.optim.AdamW([{'params':[params.data], 'lr': learning_rate}], weight_decay=wd)
 return params, optimizer
```

Next, we set up our optimizer using PyTorch's built-in functionality. The optimizer handles gradient calculation and parameter updates based on these gradients and a specified learning rate. Using `torch.optim.Adam`, we instantiate an Adam optimizer. In the dictionary passed to it, we specify that the optimizer should optimize the earlier parameters, explicitly accessing the `data` attribute within those parameters. When calling `init_params` again, it resets the system, allowing us to initialize and optimize a new configuration seamlessly.

3.5 Encoding the text prompts through CLIP

It is time to begin coding the functions that will use the clip architecture to create encodings of our text prompts. The first thing we will do is to begin testing the generation of an image from the parameters we will optimize, encoding prompts, a few more things, etc.
First of all, we need to declare a variable called normalize. There will be a function where we can apply some of the data, and these architectures require this. We will use torch-vision.transforms. normalize, and within this, we have to put these values for the mean and variance already prepared for us. This is important because this is related to the type of data used to train the architecture. We create a function called encodeText that receives a text. We will call the clip model, and first, we will tokenize the text. Tokenize the text will take each prompt and represent it with the tokens suitable for the model. After we take the result, we pass it through the encode function of the model.

```
def encodeText(text):
 t=clip.tokenize(text).cuda()
 t=clipmodel.encode_text(t).detach().clone()
 return t
```

This function is designed to generate an encoding using the CLIP architecture from a given prompt. It takes a `prompt` and `text` as inputs. The `t = clipmodel.encode_text(t).detach().clone()` line executes the encoding process: `t` represents the result of encoding `text` through the CLIP model. `.detach()` ensures the result is detached from gradient computation, and `.clone()` creates a copy in our own memory space. Moving forward, we define a more versatile function named `createEncoding` that can handle various parameters related to prompts. This function includes `include`, `exclude`, and `extras` options for specifying what to include, exclude, or add to the prompts. For each text included in `include`, we call `encodeText` to obtain its encoding and store it in an array accumulating all these encodings.

If `exclude` or `extras` are specified (and not empty), their respective encodings are also calculated using `encodeText`. If they are empty, they return zeros. This setup ensures that the function can dynamically handle different configurations of prompts while generating encodings effectively. Additionally, when generating an image, we send multiple crops of the image to CLIP for encoding. This approach provides CLIP with diverse views of the image, aiding in better understanding and matching with text encodings. Before encoding, the image undergoes augmentations and transformations stored in `augTransform`, defined as `torch.nn.Sequential`. This sequence includes random horizontal flips (`torchvision.transforms.RandomHorizontalFlip`) and affine transformations (`torchvision.transforms.RandomAffine`) such as rotation and translation, with areas outside the image boundary filled with zeros. These transformations help augment the data presented to CLIP, enhancing its ability to interpret and match images with textual descriptions effectively.

```
with torch.no_grad():
 print(Params().shape)
 img= norm_data(generator(Params()).cpu()) # 1 x 3 x 224 x 400 [225 x 400]
 print("img dimensions: ",img.shape)
 show_from_tensor(img[0])
```

Let's proceed with a simple test to generate an image using our transformer generator. Since the initial parameters are random, we use `torch.no_grad()` to skip gradient computations. First, we print the shape of the parameters. Then, we call the generator function, passing these parameters and transferring the resulting image back to the CPU. Currently, the generated image may not look visually appealing as it's based on random noise, but the size of the image is the key focus here. This test helps ensure that the generator function is correctly set up to produce an output image based on the initial parameters provided.

Figure 3: Encoding prompts

3.6 Creating crops from the generated image

We will create a function called `create_crops` to prepare multiple image crops with augmentations, rotations, and translations for encoding with the CLIP architecture. This function takes an `image` as input and the desired number of `crops`. We add padding around the image to ensure transformations can be applied effectively. This padding size is set to half the image height using `torch.nn.functional.pad`. Next, the previously defined transformations (`augTransform`) are applied to the padded image. Now, let us generate the crops. We initialize an empty list `crop_set` to accumulate the generated crops. For each crop we aim to generate (specified by the `crops` parameter), we proceed with the following steps:

1. *Calculate Offsets:* We define `gap1` and `gap2`, which determine the offset amounts along the x and y axes. These values are sampled from a normal distribution with mean and variance constraints, ensuring they fall within specified ranges and are proportional to the image size.

2. *Apply Offsets:* We compute `offset_x` and `offset_y` for each crop using random integers within the image dimensions (`size1` and `size2`). This introduces randomness, ensuring each crop is offset differently.

3. *Create Crops:* We extract a cropped section from the transformed image using these offsets. This process is repeated for the specified number of crops, each with unique offsets and cropping positions.

By applying these steps, we generate multiple variations of the original image, each tailored to provide diverse perspectives to CLIP during encoding. This diversity enhances CLIP's ability to understand the essence of the image and align it more effectively with textual descriptions.

When cropping the image, we preserve the first dimensions while adjusting the last two dimensions (height and width). Specifically, the crop spans from `offset_x` to `offset_x + gap2` for width and from `offset_y` to `offset_y + gap2` for height. This approach selects a fragment of the image within specified dimensions and then resizes it to 224x224 pixels. This resolution is necessary because our CLIP model requires inputs of this specific size for optimal processin

```
crop = torch.nn.functional.interpolate(crop,(224,224), mode='bilinear', align_corners=True)
  crop_set.append(crop)
```

We utilize this function to resize each crop, setting the mode to bilinear and aligning corners, which ensures accurate resizing without distortion. All generated crops are accumulated and prepared for further processing or encoding.

```
img_crops=torch.cat(crop_set,0) ## 30 x 3 x 224 x 224
```

We will use torch cut to concatenate all of those crops because we will send them all at once to CLIP.

```
randnormal = torch.randn_like(img_crops, requires_grad=False)
 num_rands=0
 randstotal=torch.rand((img_crops.shape[0],1,1,1)).cuda() #32

 for ns in range(num_rands):
  randstotal*=torch.rand((img_crops.shape[0],1,1,1)).cuda()

 img_crops = img_crops + noise_factor*randstotal*randnormal

 return img_crops
```

We are adding noise to the `img_crops` variable after interpolating it. We have a randnormal function and we are going to take a different type of noise that is going to be taken from a unifrom distribution and apply it in a uniform way to all the parts of the content of each of the crops, the same random value. We are going to do different cycles of this as we are going to repeat it over and over as much as we like. The effect that this produces in th eend is going to be amazing and striking. We are going to create a new variable called “num_rands” and we can say 2 or 4 or 6 whatever we want which will be the number of these extra cycles that we are going to create. We initialize a new variable called randstotal and do torch.rad. The difference between rand and randint is that randint takes values from a normal distribution and rand from a uniform mathematical distribution. We are going to set the first dimension as the shape of the first dimension of the image crops, which in our case is 30. What we are going to do by doing this is that we are going to get this variable only and ser it in the GPU.
For ns in range(num_rands) we do randstotal and we are going to multiply the content of that, the uniform noise, with another independent and new version of the same noise. In our last line we multiply the noise factor times the randstotal times randstotal. When our code starts to generate, we have created this sfumato effect that Leonardo Davinci used to use, as this is a wau if blurring the transitions between the colors and produce a more smokey effect. It also moderates, because of the RGB mode, the saturation of the colors etc. Sfumato is done by layering, creating a treatment that is done through layering and it works well for prompt that are related to paintings and sketches.

3.6.1 A function to display generated images and crops

The function `showme` is designed to visualize the generated images and their augmented crops if specified, using the parameters provided (`Params`).

1. *Generator Invocation:* Inside a `torch.no_grad()` context, the function first generates an image using the `generator` function with parameters initialized by `Params()`. This ensures that no gradients are calculated during this process, which is suitable for inference or visualization purposes.

2. *Showing Augmented Crops:* If `show_crop` is `True`, the function proceeds to print and display an augmented crop of the generated image. It converts `generated` to a float tensor (`aug_gen`) and applies `create_crops` to generate a specified number of crops (`num_crops=1`). Each crop is then normalized (`norm_data`) and displayed using `show_from_tensor`.

3. *Showing Original Generation:* Regardless of whether crops are shown, the function continues by printing and displaying the original generated image (`latest_gen`). It converts `generated` to CPU (`generated.cpu()`), normalizes it (`norm_data`), and displays the first image from the resulting tensor using `show_from_tensor`.

4. *Return Value:* The function returns the normalized tensor (`latest_gen[0]`), which represents the original generated image.

This function serves to visually inspect the output of the generator function, facilitating understanding and validation of the generated images. It integrates features for displaying both the full image and augmented crops, enhancing the assessment of the generated outputs.

3.6.2 Optimizing the latent space parameters

We arrived at a very exciting part, that is the part of tweaking those parameters that we are gradually changin so that the resulting generated image matches better and better the text prompt. The provided code defines a two-step optimization process to generate and refine images based on text prompts using CLIP model encodings.

```
def optimize_result(Params, prompt):
  alpha=1 ## the importance of the include encodings
  beta=.5 ## the importance of the exclude encodings

  ## image encoding
  out = generator(Params())
  out = norm_data(out)
  out = create_crops(out)
  out = normalize(out) # 30 x 3 x 224 x 224
  image_enc=clipmodel.encode_image(out) ## 30 x 512
```

In the first step, the `optimize_result` function handles image and text encodings along with loss calculation. It starts by generating an image using the `Params` function, followed by preprocessing the image through normalization and cropping steps, and then encoding the image using the CLIP model. It involves a PyTorch optimizer, where we set the gradients to zero, then do backpropagation from the loss value and then tweak and update with step and return the loss. For text encoding, the function combines the text prompt with extra encodings, normalizes the result, and defines a final text encoding for exclusion. We are declaring a couple of variables, alpha and beta, 0.1 and 0.5. And these are going to be the importance of the include encodings (alpha) and the importance of the exclude encodings (beta). With this we can regulate how you want to exclude encodings to be in comparison with the include encodings[24]. After this we generate a new image from our parameters, to calculate the loss we will need to compare the encodings of the text prompts with the encodings of the text prompts with the encodings of the image crops generated from the current state of our latent space parameters. That's why we first

[24] Encodings are strings containing the encoding name

need to pass our parameters through the generator to generate an image that we can encode. We create our image encoding calling the clip model and the encode image method.

```
 ## text encoding  w1 and w2
  final_enc = w1*prompt + w1*extras_enc # prompt and extras_enc : 1 x 512
      final_text_include_enc   =   final_enc   /   final_enc.norm(dim=-1,
keepdim=True) # 1 x 512
  final_text_exclude_enc = exclude_enc
```

For each experiment that we are going to do, we will declare W1 and W2 that is going to be the weight of the include text encoding versus the extras text encoding. The prompt is the current part of the include encoding plus W2 times the extra encoding. Both the prompt and the extra encoding have a dimensionality of 1 times 512. So again the encodings returned by clip have a dimensionality of 512. In the case of the image we have 30 because we have 30 crops of the image, but in the case of the text each one of them is going to have a 1 times 512 dimensionality. Now we need the final text include encoding which is going to be the same thing but we have to normalize it. We call the normlaized function in the last dimension, keep dimension equal true and this is going to produce 1 times 512, so we divide those values by their normalization to get them on the correct range.

```
## calculate the loss
   main_loss  =  torch.cosine_similarity(final_text_include_enc,  image_enc,
-1) # 30
      penalize_loss   =   torch.cosine_similarity(final_text_exclude_enc,
image_enc, -1) # 30

  final_loss = -alpha*main_loss + beta*penalize_loss

  return final_loss
```

The loss calculation involves computing the cosine similarity between the image encoding and both the include and exclude text encodings, and then combining these losses with weighted factors to get the final loss. We are using a function called `torch.cosine_similarities` which applies a mathematical function that is great to compare vectors and to really find out how similar to mathematical vectors or structures are.
In the second step, the `optimize` function focuses on optimizing the model parameters. It calculates the average loss by calling `optimize_result`, then performs gradient zeroing and backpropagation, and finally updates the model parameters using the provided optimizer. This iterative process aims to refine image generation so that the resulting images closely match the desired text prompt while penalizing features associated with undesired attributes.

3.6.3 Training loop

In the training loop, we pass the parameters, the optimizer and say if we want to show the crops or not. We create two structures, result images and result the latent space, which we will call z. These two structures are going to accumulate, that we are going to save in them the different images that we generate and also the different sets of parameters as we optimize them. We want to store them here so we can use them later to do things such as interpolations.

```
def training_loop(Params, optimizer, show_crop=False):
  res_img=[]
  res_z=[]
```

For each prompt and phrase in the include encodings, we begin with the iteration set to zero. For the iteration in range, we said we are going to do 100 iterations of evolution of optimization. For each of the text prompts we can do 100,200,500 but the more we set, the more time and memory. So with 100 we already see nice results. Next we calculate the loss, calling our optimized function, passing the parameters, the optimizer and the prompt. T he loop periodically generates images using the `Params` parameters and evaluates them. If the current iteration is 80 or more and a multiple of `show_step`, the `showme` function generates a new image with the current `Params` values, which is then appended to the `res_img` list. The current parameters are also saved in the `res_z` list. The loop prints the current loss value and iteration number for progress tracking. After each iteration, the iteration counter is incremented, and unused GPU memory is cleared with `torch.cuda.empty_cache()`. Finally, after the loop completes, the function returns the lists `res_img` and `res_z`, containing all generated images and their corresponding parameters.

```
   if iteration>=80 and iteration%show_step == 0:
        new_img = showme(Params, show_crop)
        res_img.append(new_img)
        res_z.append(Params()) # 1 x 256 x 14 x 25
        print("loss:", loss.item(), "\niteration:",iteration)

      iteration+=1
    torch.cuda.empty_cache()
  return res_img, res_z
```

Finally, we can return the structure that accumulates the images and the latent parameters.

3.6.4 Running the training

At the beginning, we make sure that the GPU cache is empty. We want to include text prompts and exclude things that can cause confusion and incoherent generation, such as a cut and blurry image. We also want to add in extras such as a watercolour paper texture. We are creating w1=1

and w2=1 to represent weights used within the training loop, where setting them to 1 indicates that both components are equally weighted in the calculations. We create the encoding and call the loop to start the image generation process by running the 'training_loop' function. This function uses 'Params' to initialize the parameters, which are then optimized using the specified 'optimizer'. As the loop iterates, it refines the images and shows intermediate results. The final outputs store the generated images and their corresponding parameters.

```
include=['A painting of a pineapple in a bowl']
exclude='watermark'
extras = ""
w1=1
w2=1
noise_factor= .22
total_iter=110
show_step=10 # set this to see the result every 10 interations beyond
iteration 80
include_enc, exclude_enc, extras_enc = createEncodings(include, exclude,
extras)
res_img, res_z=training_loop(Params, optimizer, show_crop=True)
```

Before we do another one, we will study the dimensionality of the results. In the first line, we show the length of this structure and then the dimensionality of one of these resulting images and one of these resulting latent parameters. We also print what is the minimum and maximum of this latent structure. In the output, the length is two and two as we had two prompts, one as a sketch of a lady and a sketch of a man in a horse. Therefore, we are getting two images/two sets of latent parameters. Then, the dimensionality of each image is three channels and dimensions of 400 in width and 224 in height because of the architecture's adjustments. We have batch one, which has 256 channels and a 14 x 25 width and height divided by 16 because of the 16 x 16 patches used by the generative architecture.

```
print(len(res_img), len(res_z))
print(res_img[0].shape, res_z[0].shape)
print(res_z[0].max(), res_z[0].min())
```

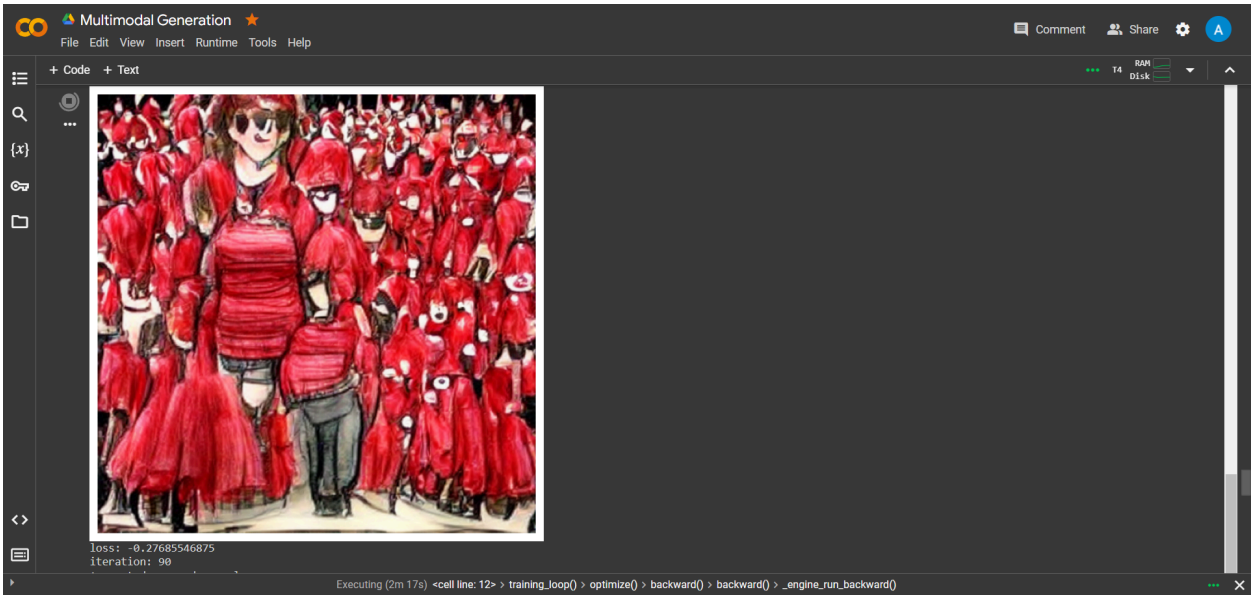

Figure 4: 100 people in a red jacket prompt generation

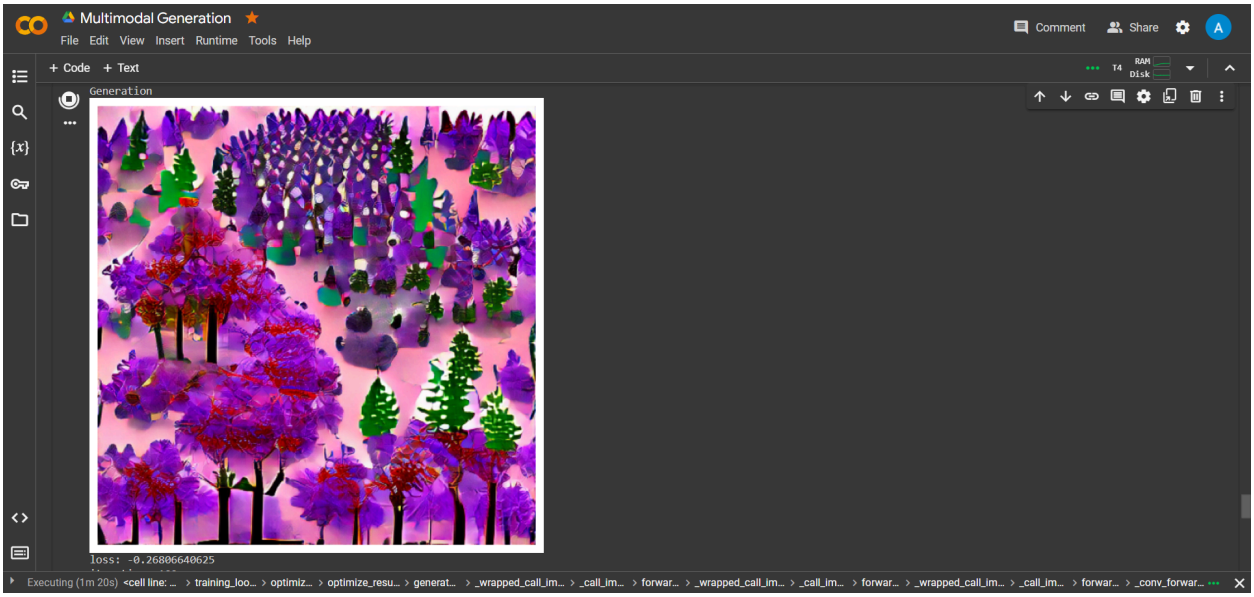

Figure 5: Purple forest prompt generation

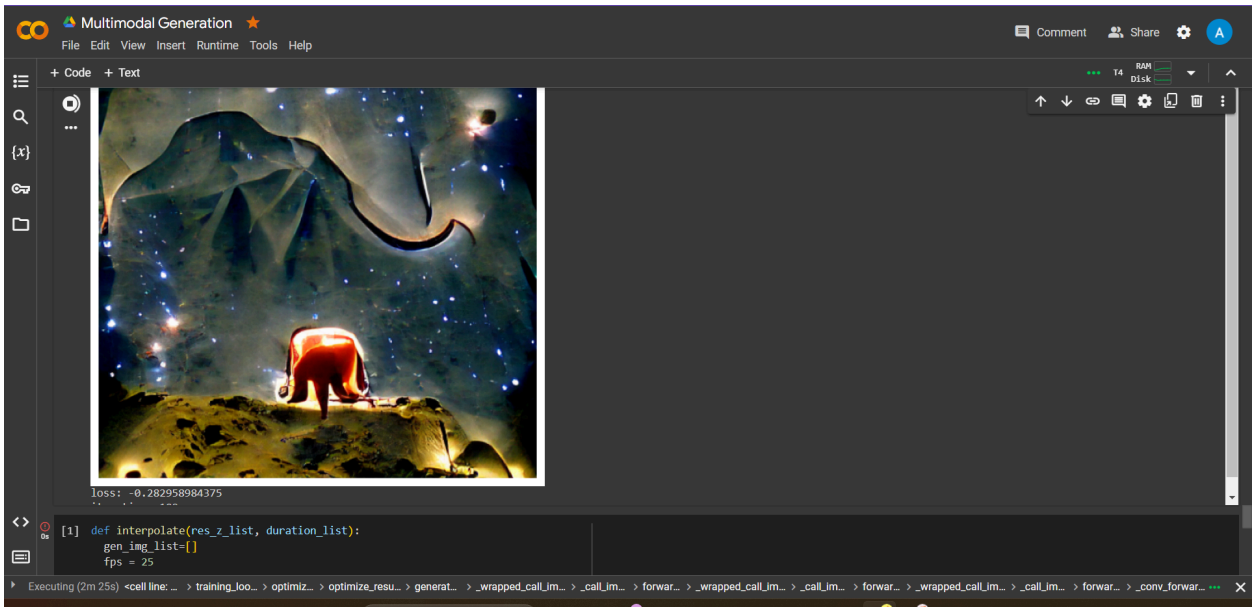

Figure 6: Boy on top of a mountain prompt generation

3.7 Creating a video of the interpolations and general review

One enjoyable thing we will create is an interpolation between the latent values of the parameters of each of these representations so we can create a video between them. We need to create a new function, where interpolate will receive our list of the latent parameters of the different images and a list of durations, how long we want each of them to last on the video. We will create a data structure to hold the generated images and how many frames per second for the video we want.

```
def interpolate(res_z_list, duration_list):
  gen_img_list=[]
  fps = 25
```

Now we make a loop, as we will loop for IDX and then the latent values and the duration in enumerate. We will package the list of the latent values and the durations together. What the loop is doing is that we are taking the list of the latent values, the parameters and the list of the durations all together with zip. It enumerates them, and then in z, the duration is going to go through each of the set of latent parameters and each of the durations. The number of steps that we will need to produce is the integer of duration*fps. If we are saying that one of these images should last three seconds, 3 seconds times 25 frames per second, it is going to be 75 frames that we need to generate in the interpolation. The beginning will be the latent point, where we are at the moment, that is, z, and the destination will be the next point, which will be in the list of patents. We are also going to do a module "%len(res_z_list)" that will interpolate from the first to the second, second to third and from the third to the first one again. So this is the initial point. So, we calculate the number of steps we need for each of them depending on the duration we want and calculate the origin and destination of the interpolation.

```
  for idx, (z, duration) in enumerate(zip(res_z_list, duration_list)):
    num_steps = int(duration*fps)
    z1=z
     z2=res_z_list[(idx+1)%len(res_z_list)] # 1 x 256 x 14 x 25 (225/16,
400/16)
```

To create the steps, we will create an interpolation in which the new point in the latent space will be alpha times the destination plus a one minus alpha times the origin. This is how we always do an interpolation, but in this case, what an alpha does is that it produces a value of the interpolation that is going to be faster in the middle, and then it is going to slow down towards the end. The bigger the exponential we put, the bigger the contrast between the slowing down of that number as it changes at the end in comparison with the previous moments. .

```
 for step in range(num_steps):
      alpha = math.sin(1.5*step/num_steps)**6
      z_new = alpha * z2 + (1-alpha) * z1
```

Now, we will pass through the generator that new point in the latent space that will generate a new image that we will pass to the CPU. We are going to normalize that to be in the range from 0 to 1 and pick the first dimension, so what we are going to pick is 3 x 224 x 400, which is next stored into new_gen. Now, we convert it to a PIL image and store it on a list that we declared before.

We are now going to create a video from those interpolated images. We declared a writer with the image io library; we got a writer to write a file in the "out_video_path" with 25 frames per second. Now, for each of the images in the result list, we are going to convert those images to an NP array with the type set to an integer of 8 bits. That is what we need to write the values for this MP4 video. We write it, append the data to what is being written in the file, and finally close the file. If we refresh, there is a video, Res1.mp4, where we have an MP4 video. How do we display this video? We use the Ipython library, import HTML and then use the "b64encode" function to encode binary data into Base64 format. We do MP4 equal and open the file saved on the disk in read and binary mode. Ultimately, we do this formatting with a height of 800 with controls and the source, and we set the placeholder as a video MP4.

```
from IPython.display import HTML
from base64 import b64encode

mp4 = open('../video.mp4','rb').read()
data="data:video/mp4;base64,"+b64encode(mp4).decode()
```

```
HTML("""<video width=800 controls><source src="%s"
type="video/mp4"></video>""" % data)
```

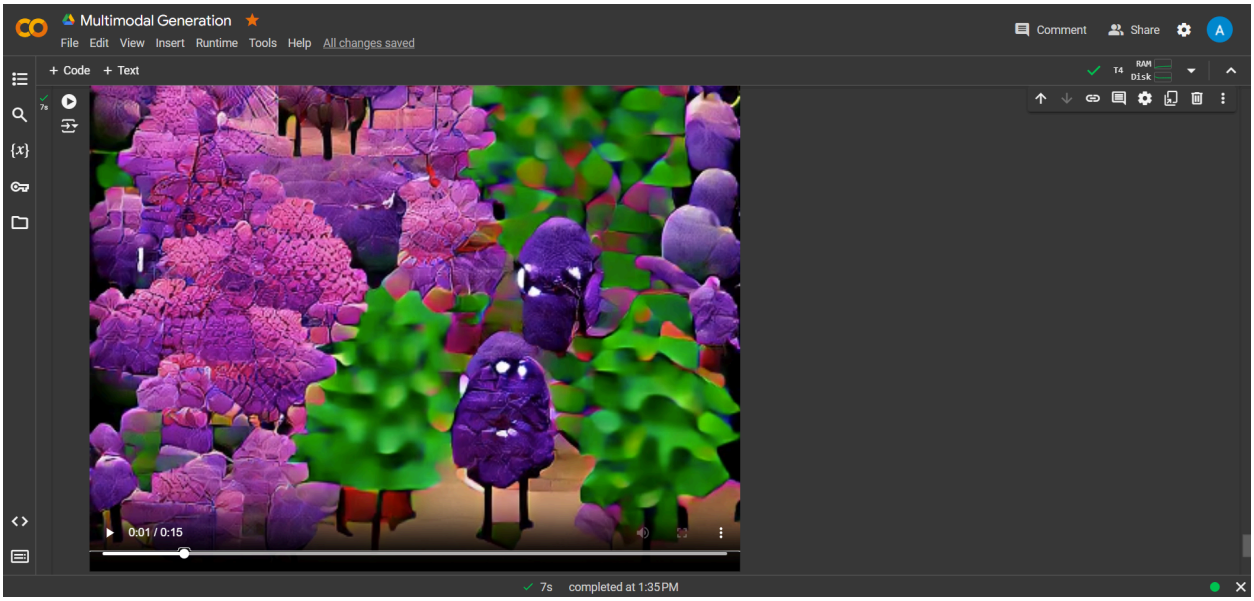

Figure 8: Video from interpolated images

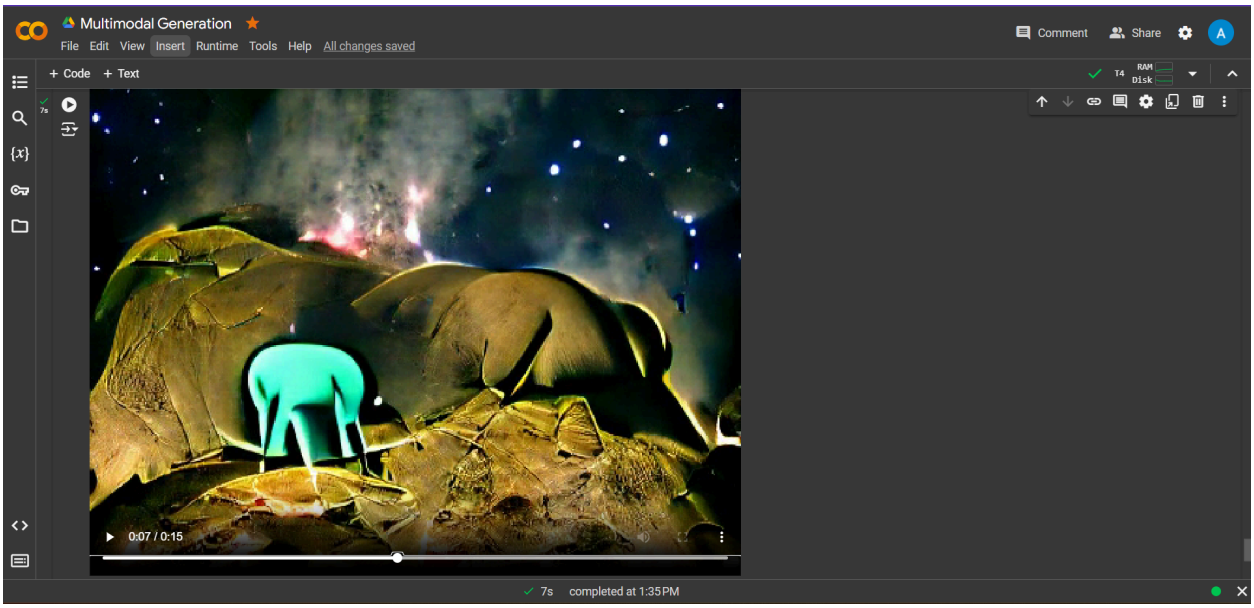

Figure 9: Video from interpolated images

Conclusion

4.1 Conclusion of the Multimodal Generation

This thesis conducts a comprehensive analysis of the application of generative adversarial networks (GANs) in the realm of image processing and optimization, highlighting the main features and benefits of contemporary methods. The introduction of Contrastive Language–Image Pre-training (CLIP) methods and Vector Quantized GAN (VQGAN) models presents effective solutions for tasks such as image classification and image recovery, showcasing the extensive applicability of GANs in the field of image processing. Furthermore, the implementation of global and local image patching techniques effectively addresses the challenge of filling in missing areas within images, yielding impressive results and offering novel approaches for image restoration. These findings underscore the significant potential of GAN-based methods in addressing practical image processing issues, serving as a valuable reference for future research and applications.

Looking ahead, advancements in technology and continued research on GANs are expected to give rise to even more innovative methods and techniques. For instance, integrating deep learning approaches with traditional image processing techniques could result in more efficient and accurate image repair algorithms. Additionally, expanding the application of GANs to other domains such as medical image processing and natural image synthesis represents a promising direction for future research. By persistently exploring and innovating in this field, we can further enhance the application of GANs in image processing and optimization, ultimately providing more effective solutions to a wide range of practical problems.

The ongoing evolution of GAN technology, coupled with interdisciplinary collaboration, holds the potential to revolutionize various aspects of image processing. As we push the boundaries of what is possible with GANs, we anticipate the development of new methodologies that will not only improve existing techniques but also pave the way for groundbreaking applications in diverse fields.

References

Lin, Lequan & Li, Ruikun & Li, Xuliang & Gao, Junbin & Li, Zhengkun. (2023). *Diffusion models for time-series applications: a survey. Frontiers of Information Technology & Electronic Engineering.* 25. 1-23. 10.1631/FITEE.2300310.

Katsumata, Kai & Vo, Duc & Liu, Bei & Nakayama, Hideki. (2024). *Revisiting Latent Space of GAN Inversion for Robust Real Image Editing*. 5301-5310. 10.1109/WACV57701.2024.00523.

Jaffri, Zain Ul Abidin & Chen, Meng & Bao, Shudi. (2024). *Understanding GANs: fundamentals, variants, training challenges, applications, and open problems. Multimedia Tools and Applications.* 1-77. 10.1007/s11042-024-19361-y.

Paul, O. A. (2021). *Deepfakes Generated by Generative Adversarial Networks.* https://core.ac.uk/download/483351921.pdf

Truong, L. V. (2022). *Generative Adversarial Nets: Can we generate a new dataset based on only one training set? ArXiv* (Cornell University). https://doi.org/10.48550/arxiv.2210.06005

Artificial Intelligence and Machine Learning. https://mmcalumni.ca/blog/the-advancements-and-applications-of-artificial-intelligence-ai-and-machine-learning-ml-in-todays-world

[1607.02748] Adversarial Training For Sketch Retrieval. https://www.arxiv-vanity.com/papers/1607.02748/

Text to Image Synthesis Using Multimodal (VQGAN + CLIP) Architectures - Hashnode. https://hashnode.com/post/text-to-image-synthesis-using-multimodal-vqgan-clip-architectures-896b8a6588ef-ckxlztr7x00f571s187kcc8ud

Yang, X., Li, Q., Zhao, X., Liu, D., & Shi, N. (2024). The Applications of Generative Adversarial Networks in Architecture Design: A Systematic Review. Journal of Design Service and Social Innovation. https://doi.org/10.59528/ms.jdssi2024.0130a14

Merrick, L., & Gu, Q. (2018). Exploring the use of adaptive gradient methods in effective deep learning systems. https://doi.org/10.1109/sieds.2018.8374740

What Boards Should Know About AI and Generative AI - C-Suite Insider. https://www.c-suiteinsider.com/what-boards-should-know-about-ai-and-generative-ai/

Gupta, R. (2023, June 25). Generative AI for beginners part 4: Introduction to generative AI. *Medium*. https://medium.com/@raja.gupta20/generative-ai-for-beginners-part-4-introduction-to-generative-ai-bb70f0854128