CAIRO UNIVERSITY

FACULTY OF ENGINEERING

DEPARTMENT OF COMPUTER ENGINEERING

Retratista

A Graduation Project Report Submitted

to

Faculty of Engineering, Cairo University

in Partial Fulfillment of the requirements of the degree

of

Bachelor of Science in Computer Engineering.

Presented by

Mohamed Shawky Zaky AbdelAal Sabae

Remonda Talaat Eskarous

Mohamed Ahmed Mohamed Ahmed

Mohamed Ramzy Helmy Ibrahim

Supervised by

Dr. Mayada Hadhoud

26th July, 2021

Abstract

الملخص

Acknowledgement

Contents

List of Figures

List of Tables

List of Abbreviation

List of Symbols

Contacts

1 Introduction

1.1 Motivation and Justification

1.2 Project Objectives and Problem Definition

1.3 Project Outcomes

1.4 Document Organization

2 Market Feasibility Study

2.1 Targeted Customers

2.2 Market Survey

2.2.1 FaceAPP

2.2.2 PicsArt

2.2.3 Facetune2

2.2.4 Booth Apps

2.3 Business Case and Financial Analysis

3 Literature Survey

3.1 Generative Models

3.2 Face Modelling and Generation

3.3 Comparative Study of Previous Work

3.4 Implemented Approach

4 System Design and Architecture

In this chapter, we discuss our working pipeline and system architecture in details. Generally, our system takes a speech note, textual description or numerical attributes as an input. It processes the input description and outputs the initial human face portrait that corresponds to the given description. Afterwards, the user is allowed to manually control some facial attributes and morphological features and to rotate the face and render it in multiple poses. In the first section, we give an overview about the system. Then, we discuss the system architecture in the second section. In the subsequent sections, each module implementation is discussed in details.

4.1 Overview and Assumptions

As mentioned above, our system basically enables the user to describe a human face in words or using numerical values and turns it into a full human face portrait that can be manipulated and rendered in multiple poses. The system relies heavily on generative models and text processing, both are iteratively designed to obtain the required results. The overall flow can be described as follows :

- The input speech notes are translated to text.
- The textual description (extracted from speech input or manually entered) is processed to extract the numerical values of the required facial features.
- The numerical values are used generate a face embedding vector that encodes the facial attributes in low dimensional space ($512D$).
- A generative model is specifically designed to translate from the low dimensional embedding into the full face portrait ($1024X1024$).
- The generated face portrait can be further refined by navigating the face embedding space and re-generating the face portrait.
- Once the user settles on the final face portrait, the system can render that face in multiple poses to provide further identification.

The previous flow provides a very versatile framework to generate face portrait and adjust it to your liking. However, there is an extremely large number of facial attributes and morphological features to describe a human face. Consequently, we have to choose a descriptive subset of these attributes to consider in the face description. We consider 32 facial attributes for face description, which are listed as follows :

- Overall face :
  - Gender : Male / Female.
  - Age : Young / Old.
  - Thickness : Chubby / Slim.

  - Shape : Oval / Circular.
  - Skin Color : Black / White.
  - Cheeks : Normal / Rosy.
- Eyes :
  - Color : Black / Blue / Green / Brown.
  - Width : Wide / Narrow.
  - Eyebrows : Light / Bushy.
  - Bags Under Eyes : On / Off.
- Nose :
  - Size : Big / Small.
  - Pointy : On / Off.
- Ears :
  - Size : Big / Small.
- Jaw :
  - Mouth Size : Big / Small.
  - Lips Size : Big / Small.
  - Cheekbones : Low / High.
  - Double Chin : On / Off.
- Hair :
  - Color : Black / Blonde / Brown / Red / Gray.
  - Length : Tall / Short.
  - Style : Straight / Curly / Receding Hairline / Bald / with Bangs.
- Facial Hair :
  - Beard / None.
- Race :
  - White / Black / Asian.
- Accessories :
  - Glasses : Sight / Sun.
  - Makeup : On / Off.
  - Lipstick : On / Off.

4.2 System Architecture

Now, let's discuss our system architecture. The system consists of 6 modules, 3 core modules of the project and 3 auxiliary modules. These modules are deployed in a *web application* to provide an easy-to-use interface for face generation and manipulation. Figure 4.1 shows the complete block diagram of the system architecture. Meanwhile, figure 4.2 shows the application design and how the modules are deployed in a web application. The *core* modules are listed as follows :

- **Text Processing :** processes the input textual description and extracts the corresponding numerical values of facial attributes. This problem is similar to *multi-label text classification*, however the outputs are normalized scores of facial attributes, which are designed carefully to match the *face code generation* process.
- **Face Generation :**
  - **Code Generation :** converts the numerical attributes values to be low dimensional face embedding. This is the most *important* and *innovative* module of our system, because it glues the desired attributes scored with the latent space of the generative model (used to generate the face), resulting in more accurate quality outputs.
  - **Code-to-Face Translation :** translates the low dimensional face embedding into the actual face portrait. For this purpose, we use `StyleGAN2`, which is a *state-of-art latent-based generative model*, whose latent space can be manipulated easily to fit our needs.

Meanwhile, the *auxiliary* modules are listed as follows :

- **Speech Recognition :** translates the input speech to textual description.
- **Face Refinement :** uses the same generative model to manually refine the generated face portrait through navigating the latent space.
- **Multiple Head Poses Generation :** rotates the generated face portrait and renders it into multiple poses.

We discuss each module in more details in the subsequent sections. Also, these modules are organized into a web application for easier usage, as shown in Figure 4.2. The application is divided into :

- **Web (Frontend) :** which contains the user interface and, also, the *speech recognizer*. The speech recognizer is moved to the frontend to reduce the network communication overhead between the web application and the server, as transmitting text is easier than transmitting speech. Moreover, the speech recognizer doesn't require high computational power, so it can be embedded in the web application.
- **Server (Backend) :** which is separated into two servers. First server contains the *text processor* and the *generative model* and serves the requests of face generation and refinement. Second server contains the *pose generator* and serves the requests of face rotation.

The two servers can communicate with each other to exchange the generated face portraits through `TCP sockets`. Meanwhile, the web application communicates and sends requests to the servers through `HTTP REST API`.

4.2.1 Block Diagram

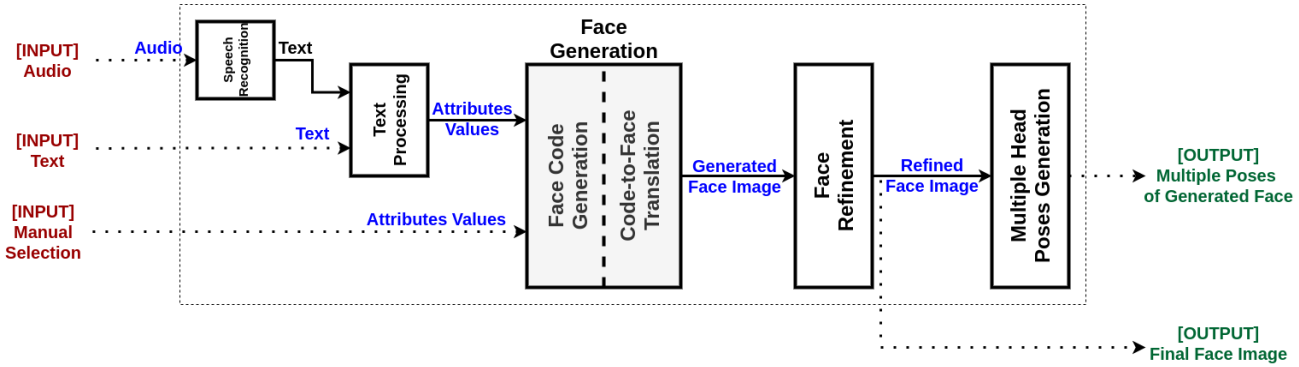

Figure 4.1: Block diagram of complete system architecture

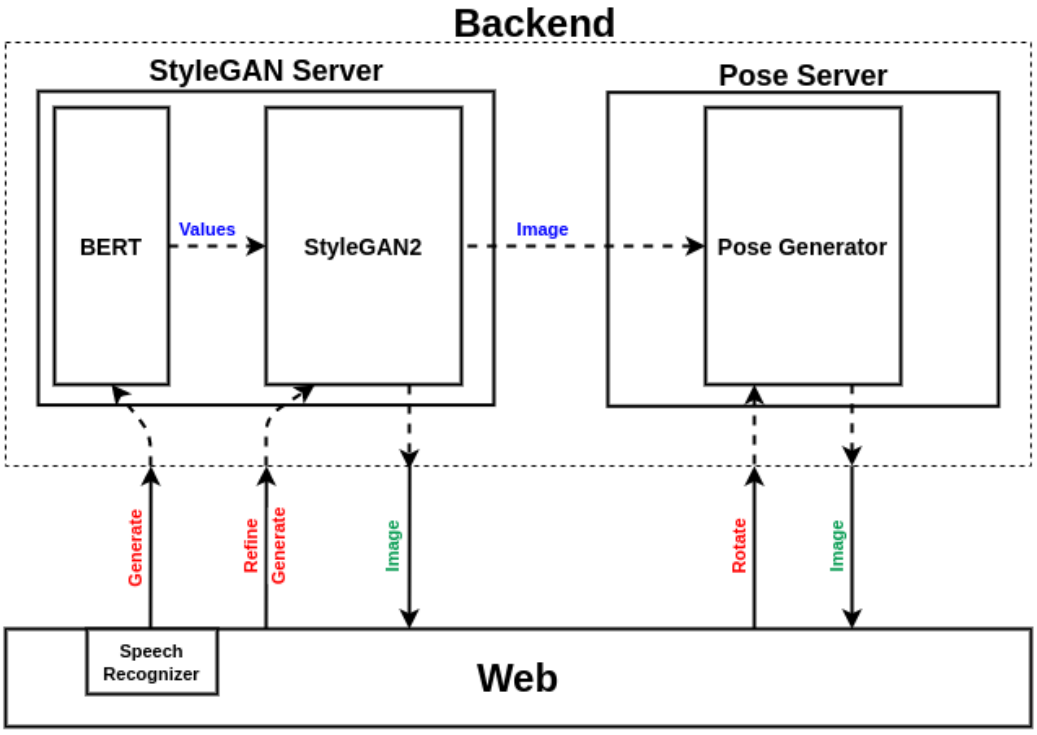

Figure 4.2: Block diagram of application design

4.3 Module 1 : Speech Recognition

4.3.1 Functional Description

4.3.2 Modular Decomposition

4.3.3 Design Constraints

4.3.4 Other Description

4.4 Module 2 : Text Processing

4.4.1 Functional Description

4.4.2 Modular Decomposition

4.4.3 Design Constraints

4.4.4 Other Description

4.5 Module 3 : Face Code Generation

Here, we discuss the face code generation from numerical values of facial attributes. This is the most important and innovative module in our system and the first stage of *face generation*. It's worth noting that we use both of the terms *"feature"* and *"attribute"* to refer to a facial attribute, like hair color or nose size.

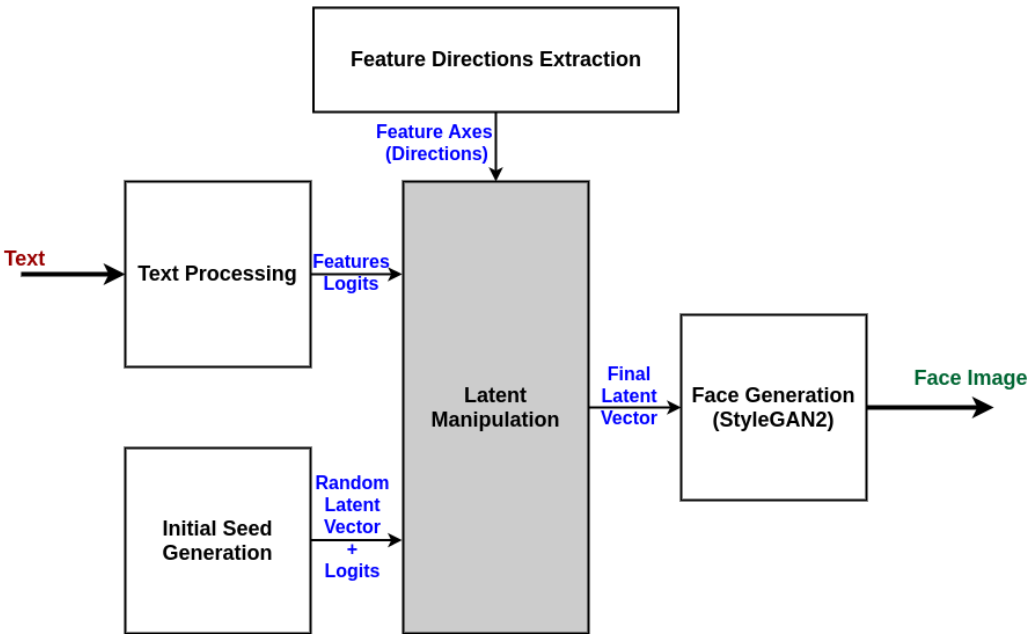

Figure 4.3: Detailed block diagram of the three core modules workflow

4.5.1 Functional Description

Figure 4.3 shows a block diagram of the interaction between the 3 core modules. We can see that the code generation module is the main driver of our face generation process. Generally, it converts the numerical attributes values (a.k.a. *logits*) into a face embedding vector

that matches the design of the latent space of the face generator (*StyleGAN2*). Basically, it starts from an initial vector and uses the *required feature values* and *extracted feature directions* to transform this vector into the final latent vector, which is passed to the generative model.

- **Input :**
  - Numerical values of facial features (logits).
- **Output :**
  - Low dimensional face embedding vector (latent vector).

4.5.2 Modular Decomposition

As figure 4.3 tells, the code generation module can be torn down into 3 sub-modules, which are **latent manipulation**, **initial seed generation** and **feature directions extraction**. Each sub-module is discussed in details to show how they integrate to each other to achieve the desired goal.

Feature Directions Generation Since, we use `StyleGAN2` [1] as our generative model, we have a full $512D$ latent space that is used to encode the whole face attributes. The changes in this latent space maps to the generated face image and similar features occupies the same area in the latent space. Consequently, we have to come up with a way to extract the axes (*hyperplanes*) in this latent space to define each of our 32 facial features. These feature directions are, then, used to manipulate the latent vector, in order to map to the required face image.

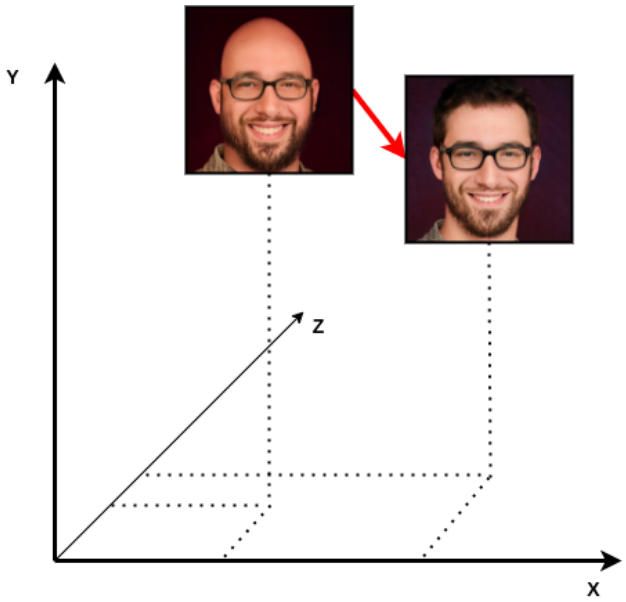

Figure 4.4: Illustration of feature directions in latent space

Figure 4.4 further illustrates the idea of feature directions in the latent space. Here, we plot two face images in a $3D$ latent space. We can see that the difference between the

two images in the existence and the absence of the hair, thus the red arrow represents the *baldness* feature direction in that $3D$ latent space (moving along this particular vector causes hair density to change).

Our method of extracting the feature directions (*hyperplanes*) consists of 3 steps :

1. **Code-Image Pairs Generation and Classification :** First, we use `StyleGAN2` to generate a large number of synthetic faces from random latent vectors. After so, we cluster the synthetic images (along with their latent vectors) according to each feature. The clustering can be based on discrete categories (like *hair color* or *race*) or continuous values (like *hair length* or *nose size*). We randomize the synthetic images in each clustering process to have better generalization and to cope with potential generation noise. For classification and regression, we use one of three possible methods, which are **manual labelling**, **classical image processing techniques** and **neural networks**. Thus, the output of this process is different groups of synthetic images sharing common facial features, along with their latent vectors.

2. **Feature Directions Fitting :** Now, we have a set of latent vectors (`X`) and their corresponding feature values (`Y`). It's required to find a set of feature directions that satisfies the mapping between feature vectors and values. This problem can be formulated as :

$$Y = A_f \cdot X \tag{1}$$

   Where A_f is the axis (direction) of feature f.
   We can obtain the solution to this equation in a closed form. However, due to the noise in both generation and classification, along with the non-linear nature of the problem, we opt to use *ML* methods, specifically **Logistic Regression** and **SVM**. Meanwhile, we cannot see any difference between both methods, as they yield almost the same results. Finally, the generated feature directions are normalized to unit vectors :

$$A_{unit} = \frac{A}{||A||} \tag{2}$$

3. **Directions Orthogonalization :** Facial features entanglement is one of the most difficult challenged of face generation. Some attributes in the human face tend to be extremely entangled by nature. For example, Asians rarely have curly hair, a woman cannot have beard and a man cannot put on makeup. Since `StyleGAN2` is trained and tuned on **FFHQ** dataset [2], which contains real human faces, it is normal to notice some entanglement between some features. Consequently, the feature directions have to be further disentangled by using *orthogonalization*. The orthogonalization process is done iteratively, starting from the most accurate feature directions. We orthogonalize other feature directions on the accurate ones, so that we have completely independent feature directions, where tuning one direction doesn't affect the others. The directions are orthogonalized as follows :

$$A_{proj} = (A \cdot B_{unit})B_{unit} \tag{3}$$

$$A_{orthogonal} = A - A_{proj} \tag{4}$$

To ensure convergence to reasonable set of feature directions, we use a threshold margin to stop the orthogonalization process, which is from 85 to 90 degrees (5 degrees on each side of normal angle).

Initial Seed Generation In order to avoid noise and discontinuities in the latent space, we generate an initial random latent vector. This is done by generating a random $512D$ vector and then passing it through the *mapping network* of `StyleGAN2`, which is not invertible. This initial vector is, then, manipulated by sequential navigation along each feature direction (axis) with certain amounts. To get these amounts, we should know the component of the initial vector along each feature direction. We do that by simply performing a dot product between the initial vector and the unit vector of each feature direction. Thus, we have an initial latent vector and the numerical attributes values, it presents.

Latent Manipulation This sub-module ingests all the inputs and produces the required latent vector (*face embedding*) that describes all of the required facial attributes. The inputs to this latent manipulation sub-module are *initial random vector* along with its logits, *text logits* and *feature directions*. The latent manipulation, simply, wants to realize the following transformation on the *initial random vector* :

$$E_{final} = E_{initial} + (l_{text} - l_{rand})D \tag{5}$$

Where E_{final} is the final latent (*embedding*) vector of dimensions $1X512$, $E_{initial}$ is the initial random vector of dimensions $1X512$, l_{text} is the text logits vector of dimensions $1X32$ (remember that we consider 32 facial features), l_{rand} is the logits vector of the initial random vector of dimensions $1X32$ and D is the feature directions matrix of dimensions $32X512$. The transformation includes calculating the difference between the required logits and the random logits and, then, use this difference to move the initial random vector along the feature directions to reach the final latent vector.

It might seem straight forward to perform this transformation. Unfortunately, it's not feasible to perform the transformation using direct matrix multiplication, mainly due to heavy *entanglement* between direction vectors even after *orthogonalization*. Also, the latent space of `StyleGAN2` can be very noisy in certain regions, so transformations have to be done carefully.

Consequently, the processing in this sub-module is done iteratively as follows :

- Both random and text logits are scaled from 0 to 1, which cannot have significant effect, when navigating using unit directions. Consequently, the inputs logits are scaled with the directions scale, which is obtained empirically to be from -4 to 4, as shown in figure 4.5.

- The next step is to get the *difference* between *text logits* and *random logits*, which is of dimensions $1X32$. We call that **differentiated logits**. It's worth noting here that the input text usually contains a *subset* of the facial features. Consequently, *not all* the text logits are set to specific values. So, when doing the *differentiation*, we set the differentiated

logits of the *unmentioned facial features* to 0. So, it can be summarized as follows :

$$l_{diff} = \begin{cases} l_{text} - l_{rand} & l_{text} \neq None \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

- Loop over each direction in feature directions :
  - Multiply the *differentiated logit* corresponding to the *current feature* with its *direction*.
  - Add the product to the current latent vector (starting with the *initial random vector*). The following equation summarizes these steps :

$$E_{next} = E_{prev} + l_{diff}[j] * D[j] \tag{7}$$

  - Finally, to ensure that every transformation is independent of the subsequent transformations and that they are applied sequentially, we re-compute the *produced latent vector logits* and re-differentiate it with the *text logits*. This is done on every iteration of the latent manipulation process.

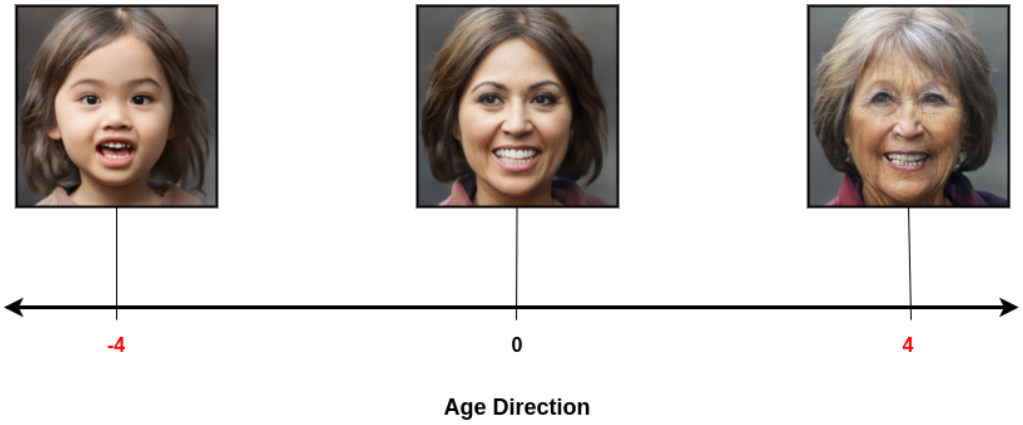

Figure 4.5: Illustration of directions scale using age direction

By applying the previous process, the *numerical value* of facial features extracted from text or manually entered by the user can be converted into a complete face embedding (*latent vector*) matching the required facial attributes. This vector can be passed to `StyleGAN2` to translate it to a complete human face image.

4.5.3 Design Constraints

The design constraints of this module are enumerated as follows :

1. **Facial attributes entanglement** is the main challenge of the face code generation module. Naturally, human face attributes are related to each other. For example, Asians barely have curly hair, no woman cannot have beard and most women have long hair and wear makeup. We mentioned before that `StyleGAN2` is trained and refined on `FFHQ`

dataset, which contains real human faces. Consequently, it's normal to see heavy entanglement between features in the latent space. Due to this *entanglement*, we have to perform extra computations to get decent results.

2. **Random initialization** of the latent vector can cause some issues with the final output, as the vector can be initialized in a noisy area of the latent vector. We try to *limit* the initialization of latent vector to a certain set of random vectors to avoid this effect. This method significantly reduces the *random initialization effect*, however it's not fully cured.

3. **Directions accuracy** can be a challenge as well. Some factors can negatively affect the feature directions accuracy. These factors include **synthetic image clustering** and **directions fitting** process. We already discussed our solutions to this problem.

4.6 Module 4 : Code-to-Face Translation

Here, we discuss the process of converting the face embedding (*latent vector*) to a complete face image using `StyleGAN2` (our chosen generative model). We discuss our reasons for choosing this particular architecture and how we use it.

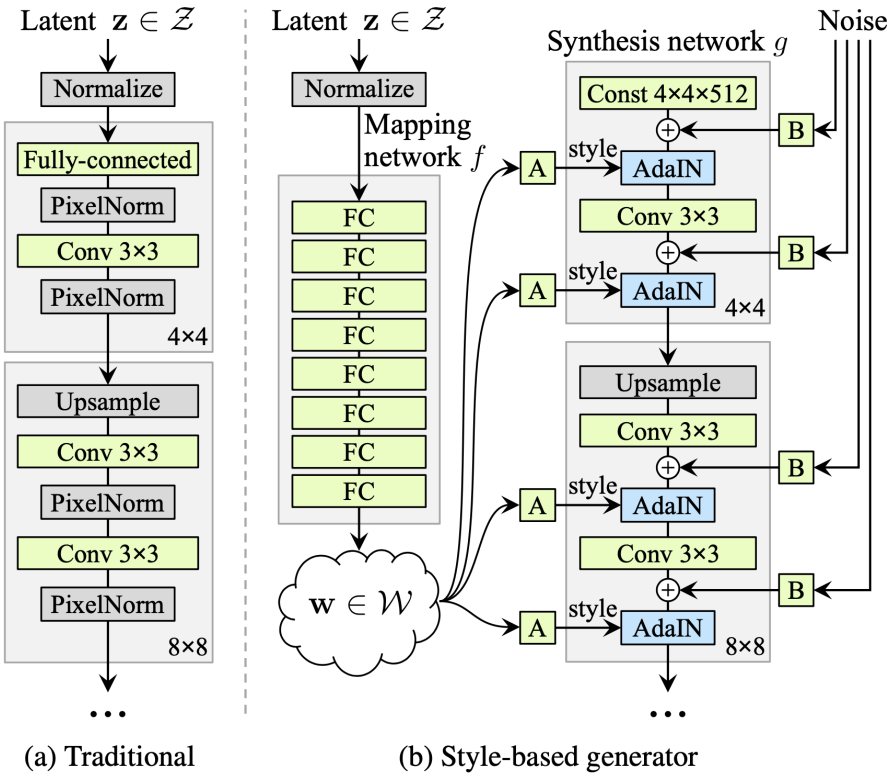

Figure 4.6: Style-based GAN architecture against traditional GAN

4.6.1 Functional Description

This module utilizes the power of *style-based* generative models, specifically `StyleGAN2` [1], to translate the required *latent vector* to a complete *human face image*. *Style-based GANs* (sometimes called *latent-based GANs*) can exert artistic control over the generated content (images, videos, text ..., etc). Consequently, we can tune it to fit our need and,

iteratively, design it along with *code generation* 4.5 and *text processing* 4.4, in order to have a complete end-to-end pipeline for *text-to-face generation*.

- **Input :**
  - Low dimensional face embedding vector (latent vector).
- **Output :**
  - Complete human face image (portrait).

4.6.2 Modular Decomposition

As mentioned before, we opt to use *Style-based GANs* to be able to artistically control the output and designed the whole pipeline for *text-to-face generation*. Moreover, we choose `StyleGAN2`, because it's one of the most popular and robust Style-based GANs in research literature. Also, it's relatively lightweight compared to other GANs used for the same purposes, but most importantly, `StyleGAN2` excels at human face generation based on latent space. Figure 4.6 shows the original architecture of `StyleGAN` and how it is compared to traditional GANs. `StyleGAN` generator has two networks as follows :

- **Mapping network** creates nonlinear transformation to the input latent vector z ($512D$). This transformation is not invertible and results in a $512D$ latent vector w. This latent vector w is expanded into several $512D$ vector using affine transformation, which gives the *extended latent vector* $w+$. The extended latent vector $w+$ dimensions depend on the dimensions of the output image.
- **Synthesis network** generates the synthetic image from *normally-distributed noise* guided by the extended latent vector $w+$.

`StyleGAN2` [1] is a newer version that follows the same architecture, but with some modifications to further improve the control over latent space and the quality of the outputs.

So, let's discuss how we adapted `StyleGAN2` to our work :

- To provide a high fidelity results, we target $1024X1024$ synthetic images. To achieve this, we have to use an extended latent vector $w+$ of dimensions $18X512$, meaning we repeat the latent vector w 18 times with *affine transformation* for each.
- To further improve feature directions disentanglement, we fine-tune `StyleGAN2` using a subset of `FFHQ` dataset with increasing the weight of *perceptual path regularization* in the loss function. *Perceptual path regularization* in `StyleGAN2` loss encourages the smooth mapping between latent and image spaces. So, when increasing it in certain directions, it highly penalizes the deviation between latent and image spaces in these directions giving more organized latent space. To avoid using the whole dataset, we use `StyleGAN2` *adaptive discriminator augmentation* (**ADA**) [3] training methodology.
- Finally, we opt to remove the *mapping network* of *StyleGAN2* generator and only use the *synthesis network*. This is mainly because :

- The mapping network doesn't satisfy the *path length regularization*, so there is no smooth mapping between latent space z and image space (only with latent space w). This is discussed in the original paper [1] and our experiments support that.
- The mapping network is not invertible, unlike the synthesis network. So, we cannot reverse the transformation from image space to latent space z. That's why we only work with latent space w to test the consistency of the results.
- Removing the mapping network reduces the computations and the memory footprint, which is crucial in our case.

After the previous modifications, the output model can directly translate the face embedding vector (*latent space*) into a complete human face portrait.

4.6.3 Design Constraints

The basic design constraints for this module can be enumerated as follows :

1. **Network size** is surely one of our challenges. `StyleGAN2` has a large memory footprint, as the case with most *deep generative models*. This constrains its training and deployment. We managed to remove the *mapping network*, which gives us a change to use the full *synthesis network*.
2. **Faces dataset** is, also, a constraint, because real images of human faces contains entanglement between facial features (*as discussed before*). So, we have to exert extra effort in the **code generation** module to solve some of this entanglement, emerging from real human faces datasets.

4.7 Module 5 : Face Refinement

4.7.1 Functional Description

4.7.2 Modular Decomposition

4.7.3 Design Constraints

4.7.4 Other Description

4.8 Module 6 : Multiple Head Poses Generation

4.8.1 Functional Description

4.8.2 Modular Decomposition

4.8.3 Design Constraints

4.8.4 Other Description

4.9 Other Approaches

5 System Testing and Verification

5.1 Testing Setup

5.2 Testing Plan and Strategy

5.2.1 Module Testing

5.2.2 Integration Testing

5.3 Testing Schedule

5.4 Comparative Results to Previous Work

6 Conclusions and Future Work

6.1 Faced Challenges

6.2 Gained Experience

6.3 Conclusions

6.4 Future Work

References

[1] Tero Karras, Samuli Laine, Miika Aittala, Janne Hellsten, Jaakko Lehtinen, and Timo Aila. Analyzing and improving the image quality of stylegan, 2020.

[2] Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks, 2019.

[3] Tero Karras, Miika Aittala, Janne Hellsten, Samuli Laine, Jaakko Lehtinen, and Timo Aila. Training generative adversarial networks with limited data, 2020.

A Development Platforms and Tools

A.1 Hardware Platforms

A.2 Software Tools

B Use Cases

C User Guide

D Code Documentation

E Feasibility Study