Ministry of Education and Science of the Republic of Kazakhstan
Suleyman Demirel University

Adilbek Karmanov, Maxim Kolesnikov, Sergey Kirichenko, Darkhan Myrzabek

AI-service for coloring old photos

A thesis submitted for the degree of
Bachelor in Information Systems
(degree code: 6B06101)

Kaskelen, 2021

Ministry of Education and Science of the Republic of Kazakhstan
Suleyman Demirel University
Faculty of Engineering and Natural Sciences

AI-service for coloring old photos

A thesis submitted for the degree of
Bachelor in Information Systems
(degree code: 6B06101)

Author: **Adilbek Karmanov, Maxim Kolesnikov, Sergey Kirichenko, Darkhan Myrzabek**

Supervisor: **Ardak Shalkarbayuly**

Dean of the faculty:
Andrey Bogdanchikov PhD

Kaskelen, 2021

Abstract

Technology is advancing rapidly these days, and something new appears every year. Our project is aimed at creating a single new application that will contain important and popular IT technologies. First of all, the main idea is artificial intelligence.

Over the past 5 years, technology has completely changed our lives, and one of the fastest-growing areas of their activity is artificial intelligence. Artificial intelligence is data displayed by machines, just like actual data presented by humans and creatures, which implies mindfulness and emotion. During the last semester of our undergraduate program, we recreated the Colorizen project, in which we will take the opportunity to recreate old photographs using deep learning in one of the subsections of artificial intelligence. With our application, we have brought old photos to life, given them a new expression, and made a convenient web-based conversion service. In our dissertation, you will learn in detail how our web service works using proven latest technologies.

Аңдатпа

Қазіргі кезде технология тез дамып келеді және жыл сайын жаңа нәрсе пайда болады. Біздің жоба маңызды және танымал IT-технологияларды қамтитын бірыңғай жаңа қосымшаны құруға бағытталған. Біріншіден, басты идея - жасанды интеллект.

Соңғы 5 жыл ішінде технологиялар біздің өмірімізді түбегейлі өзгертті және олардың белсенді дамып келе жатқан бағыттарының бірі - жасанды интеллект. Жасанды интеллект дегеніміз - бұл адамдар мен тіршілік иелері ұсынатын нақты мәліметтер сияқты машиналарда көрсететін зейін мен эмоция. Біздің бакалавриат бағдарламасының соңғы семестрінде біз Colorizen жобасын құрдық, онда біз жасанды интеллект бөлімшелерінің бірінде терең оқытуды қолданып, ескі фотосуреттерді қалпына келтірдік. Біздің веб-қосымшамен біз ескі фотосуреттерді өмірге әкелдік, оларға жаңа өрнек беріп, ыңғайлы веб-конверсия қызметін жасадық. Біздің диссертациямызда дәлелденген жаңа технологияларды қолдана отырып, біздің веб-қызметіміздің қалай жұмыс істейтінін егжей-тегжейлі білетін боласызсыз.

Аннотация

В наши дни технологии стремительно развиваются, и каждый год появляется что-то новое. Наш проект направлен на создание единого нового приложения, которое будет содержать важные и популярные IT-технологии. Прежде всего, основная идея - это искусственный интеллект.

За последние 5 лет технологии полностью изменили нашу жизнь, и одна из самых быстрорастущих сфер их деятельности - искусственный интеллект. Искусственный интеллект - это данные, отображаемые машинами, точно так же, как фактические данные, представленные людьми и существами, что подразумевает внимательность и эмоции. В течение последнего семестра нашей программы бакалавриата мы воссоздали проект Colorizen, в котором мы воспользуемся возможностью воссоздать старые фотографии с использованием глубокого обучения в одном из подразделов искусственного интеллекта. С помощью нашего приложения мы оживили старые фотографии, дали им новое выражение и сделали удобный веб-сервис конвертации. В нашей диссертации вы подробно узнаете, как работает наш веб-сервис с использованием проверенных новейших технологий.

Contents

List of abbreviations

GAN, Generative Adversarial Network

GANs, Generative Adversarial Networks

G, Generator

D, Discriminator

SAGAN, Self-Attention Generative Adversarial Network

TTUR, Two Time-Scale Update Rule

NoGAN, No Generative Adversarial Network

CNN, Convolution Neural Network

CycleGAN, Cycle Generative Adversarial Network

ResNet, Residual Neural Network

SPA, Single Page Application

JSON, JavaScript Object Notation

APIs, Application Programming Interfaces

CORS, Cross-origin resource sharing

List of Figures

Chapter 1

Introduction

1.1 Motivation

The motivation to do this project, first of all, to show that in these 4 years of studying it technologies does not go to nothing. Every year we get enough knowledge to make little apps. Finally, we can create a huge app with those experiences. Second thing is that still we don't a have big projects in artificial intelligence, and it is also a kind of motivation for us. The third thing is getting so much experience working as a team, understanding every part, and connect all this labor.

1.2 Aims and Objectives

Aim: Make a flexible service using artificial intelligence with a user-friendly interface and well-thought-out system design.

Objectives:

1. Explore existing models in conversion from old to color photographs
2. Create a special web service for the model with a user-friendly interface.
3. Try to improve the model for better results in the production
4. Make photo conversion fast for the client's convenience

1.3 Thesis Outline

In Chapter 1 (Introduction), which you are reading now, we tell how we came to this project and what goals we pursued. In Chapter 2 (Background), you will get acquainted with the material that you will need to understand how the converter

model of our service works. In Chapter 3 (Methodology), we wrote about the development process of the project and how it was built. Finally, we wrote a Conclusion section in which we summed up our thesis work.

Chapter 2

Background

2.1 Introduction

In this chapter, we will analyze in detail how our converter from old to colorized images works using *self-attention generative adversarial networks*. Before that, we will briefly explain how neural networks and generative models work for you to understand the architecture itself. After reading this chapter, you will understand the whole principle of old image conversion.

2.2 Neural Networks

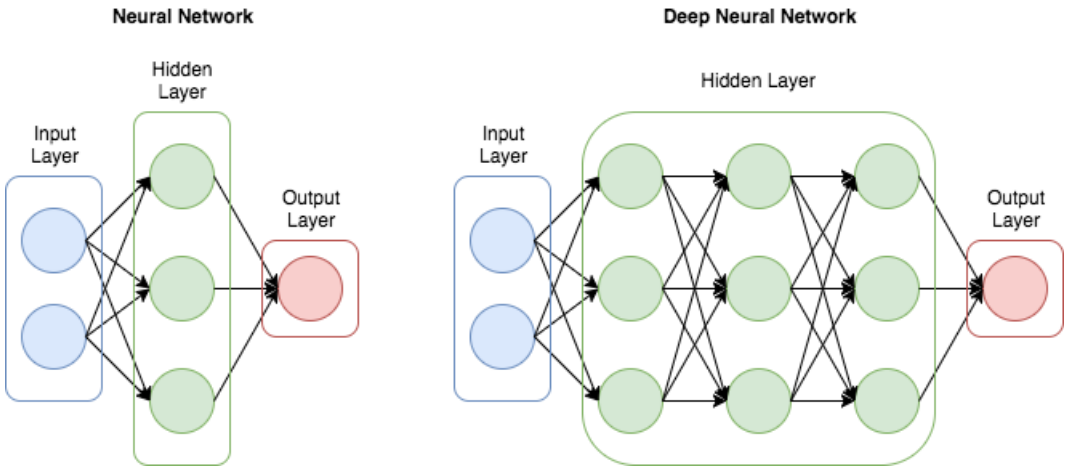

Figure 2.1: Neural Network Architecture

Neural networks are one of the machine learning algorithms created from the neural connections in the human brain. The concept of a basic neural network is a single computational node that takes a vector as input, which is multiplied by a hidden weight and transforms the vector into the desired result. The idea is to use hidden weights to teach the model to produce the needed output as shown in **Figure 2.1**. Based on this mechanism, we can build neural networks with

hundred layers. In our thesis, we took one of the neural networks as a means of estimating a probability distribution given conditional inputs. This probabilistic approach will be explained in the next section, as we use such a model to calculate the difference between original and fake images.

2.3 Generative Models

Generative models are one of the *neural networks* capable of generating realistic data. Research in this area has begun in recent years and requires solutions to many types of problems. At the moment, *generative models* have advanced in the generation of sounds, text, and pictures. Thus, we can direct *neural networks* to perform a certain action, in our case we will consider the conversion as generating the desired colors for the input image.

2.4 Generative Adversarial Networks

Generative adversarial networks (**GANs**) [13] are a form of *generative models*. These models consist of two neural networks, one of which is designated as a *generator* (**G**) and a *discriminator* (**D**). These models are trained against each other. The purpose of the generator is to generate fake samples from the input *Gaussian noise*. Unlike a *generator*, the *discriminator* model evaluates which data is fake and which is genuine.

$$V(G,D) := \underset{x\sim q_{\text{data}}(\boldsymbol{x})}{\mathrm{E}}[\log D(\boldsymbol{x})] + \underset{\boldsymbol{z}\sim p(\boldsymbol{z})}{\mathrm{E}}[\log(1 - D(G(\boldsymbol{z})))]$$

Figure 2.2: Adversarial Loss Function, Source: [13]

The standard objective function for *adversarial loss* is shown in **Figure 2.2**, where:

D → Discriminator model
G → Generator model
z → Gaussian Noise Vector
qdata → Data distribution
p(z) → Generator distribution

From the above function we should learn our models using *min-max optimization* as shown in **Figure 2.3**:

$$\min_G \max_D V(G, D)$$

Figure 2.3: Min-max optimization, Source: [13]

Both *neural networks* need to be trained at the same time to avoid overfitting these models. The *generator* model will fail at first, but will eventually learn to generate images that are indistinguishable from the original images. The whole training process of *Generative Adversarial Networks* is described in **Figure 2.4**:

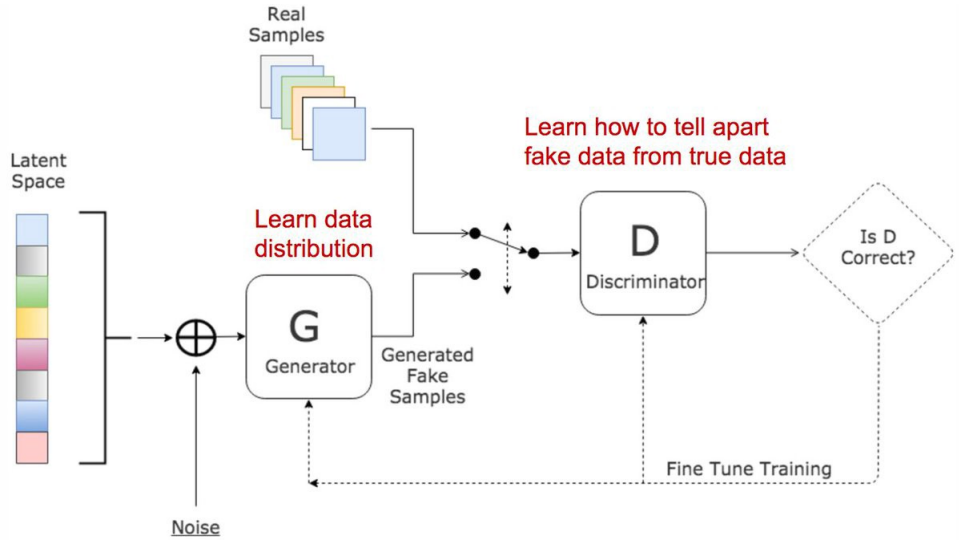

Figure 2.4: GAN model pipeline, Source: [12]

The actual training of **GANs** [13] started with low-resolution images and gradually increased in resolution, adding new layers to the network. This approach allows us to train and see the distribution of images, and then pay attention to small details to gradually train our models. Also, the *generator* and *discriminator* networks are trained synchronously and reflect each other. If we started training our models straight away from high resolution, then it would be easy for the *discriminator* to learn to recognize generated images and trained images what is causing the gradient problems so that the *generative* model will not keep up with the *discriminator*. This technique is taken from the original paper **"Progressive Growing of GANs for Improved Quality, Stability, and Variation"** [18], where all process can be presented on one scheme (**Figure 2.5**):

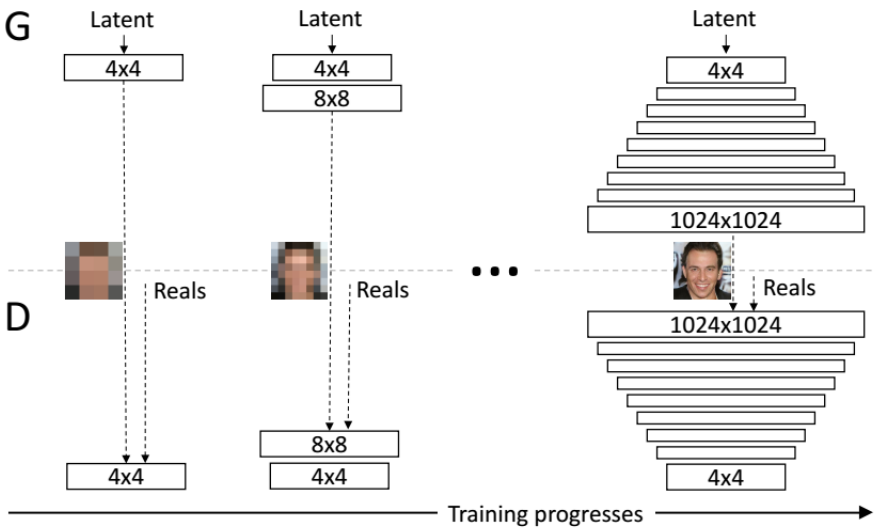

Figure 2.5: Progressive Growing of GANs, Source: [18]

In the next chapter, we will take a closer look at the architecture we used for the *generator* and *discriminator*.

2.5 Self-Attention Generative Adversarial Networks

Many **GAN** imaging models are built on convolutional layers. In convolutional layers, information is processed only at the local level, this causes problems in modeling the image on long-range dependencies. To solve this problem, a *self-attention mechanism* has been devised that balances long-range dependence and efficient computation. Thus, the *self-attention mechanism* gives an understanding of the interaction between layers at distant intervals. This mechanism is similar to the one we use in natural language processing to translate from one language to another. The *self-attention module* is shown in **Figure 2.6**, where **X** is denoted as multiplication and *softmax* will be used on each row.

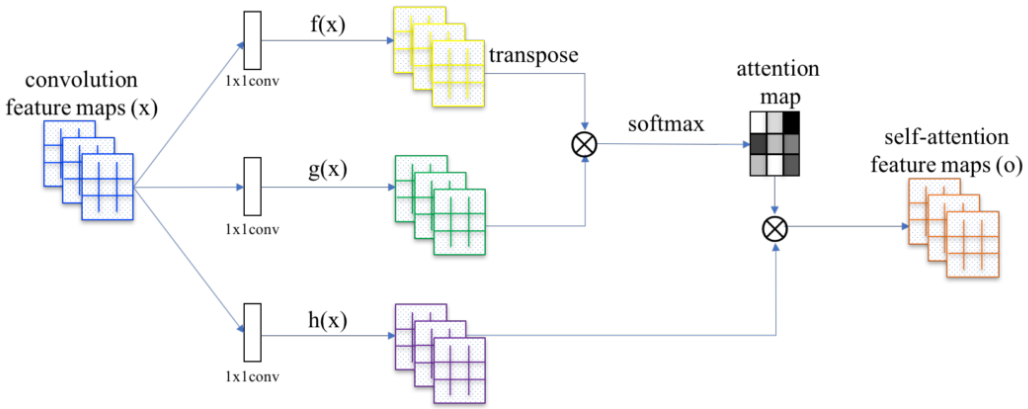

Figure 2.6: The proposed self-attention module for the SAGAN, Source: [39]

Thus, using this module, we create a **GAN** [13] with a new architecture called **SAGAN** [39]. Let's take a closer look at the *self-attention module*. At the input, we have **x** which represents features from the past hidden layer. This vector is transformed into two features spaces **f** and **g**, where **Wf** and **Wg** being the weights of the features spaces respectively (**Figure 2.7**):

$$\boldsymbol{f(x) = W_f x},\ \boldsymbol{g(x) = W_g x}$$

Figure 2.7: 1st step of self-attention module, Source: [39]

Next, we calculate the *softmax* function from the multiplication of the **f** and **g** functions, from what we get the *attention map* at the output (**Figure 2.8**):

$$\beta_{j,i} = \frac{\exp(s_{ij})}{\sum_{i=1}^{N} \exp(s_{ij})}, \text{where } s_{ij} = \boldsymbol{f(x_i)}^T \boldsymbol{g(x_j)},$$

Figure 2.8: 2nd step of self-attention module, Source: [39]

The final step is to multiply our *attention map* on **h(x)**, which is also feature space from our input **x** (Figure 2.9):

$$\boldsymbol{o_j} = \sum_{i=1}^{N} \beta_{j,i} \boldsymbol{h(x_i)}, \text{where } \boldsymbol{h(x_i) = W_h x_i}.$$

Figure 2.9: 3rd step of self-attention module, Source: [39]

The **SAGAN** models [39] with the *self-attention mechanism* at the middle-to-high level feature maps achieve better performance than the models with the *self-attention mechanism* at the low-level feature maps [39]. In addition to the *self-attention mechanism*, the **SAGAN** model also uses the *Two Time-Scale Update Rule* (**TTUR**) [16]. It is used to converge our *discriminator* model with a *generative* model for which it is used as the main objective. Generally speaking, this rule is that we start training our **G** and **D** models with separate learning rates. The main problem is that if you start with the same learning rate, the *discriminator* can get stuck at a local minimum when the *generative* model is already fixed. In our case, the learning rate of the *discriminator* model is five times higher than that of the *generator* model.

2.6 NoGAN

A special technique called **NoGAN** [4] was used to train the **SAGAN** model [39]. The authors of **NoGAN** have created a whole new form of **GAN** [13] training. It gives you the benefits of **GAN** training while taking up very little of your time. Instead, we spent the majority of our time separately training the *generator* and *critic* using more straightforward, fast, and reliable traditional methods. The procedure is as follows:

1. To begin, we traditionally train the *generator* using only the feature loss.

2. Then, as a simple binary classifier, we produce images from the trained *generator* and train *the critic* to differentiate between those outputs and real images.

3. Finally, train the *generator* and *critic* together.

All of the crucial **GAN** preparation takes place in a very short period. There's a stage when it seems that the critic has passed all of the valuable information to the generator. After the model reached the inflection point, there appears to be no productive training. Finding the inflection point tends to be the most difficult aspect, and the model is very unstable, so we must build a lot of checkpoints. Another advantage of **NoGAN** is that after the initial **GAN** training, we can pre-train the critic on produced images and then replicate the **GAN** training in the same way. Using **NoGAN** our *generator loss* now divided into two parts: One is a basic *Perceptual Loss* (or Feature Loss) based on **VGG16** [30] - this just biases the *generator* model to replicate the input image. The second is the loss score from the *critic* model.

Chapter 3

Methodology

3.1 Introduction

In this chapter, we will tell you in detail about the process of creating a project, the division of responsibilities between team members, our requirements, and the implementation of the project itself.

3.2 Development process

Before starting to develop the project, we chose the *Spiral* strategy as a software development process [31]. *The Spiral software development process* combines testing and risk assessment with the incremental nature of Iterative, Incremental, and Agile (**Figure 3.1**). It consists of four steps: Planning, Risk Assessment, Development, and validation, Evaluate results and plan next "loop". The main principle of this software development process is that we reduce the risks of project failure since we planned to use machine learning in our work, which entails great risks in our project.

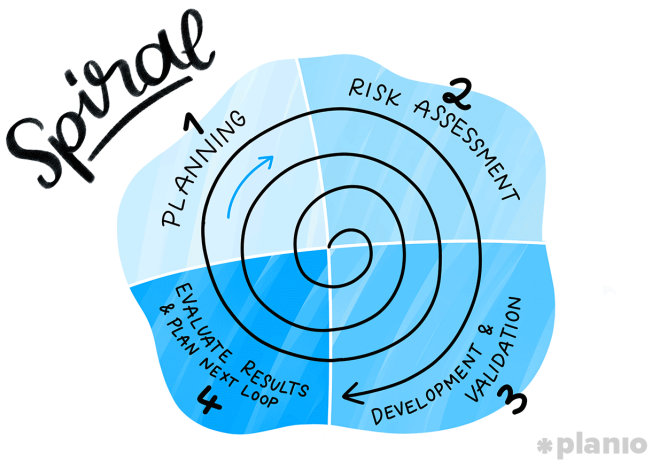

Figure 3.1: The Spiral software development process, Source: [31]

On the next step, we began to do a brainstorm on the ideas of what kind of project we should do. After a long selection, we chose ***Colorizen.cc*** - *AI-service for coloring old photos*. Next, we divided the roles in the team in this way:

- Adilbek Karmanov - Machine learning Engineer
- Maxim Kolesnikov - Machine learning Engineer
- Sergey Kirichenko - Frontend Developer
- Darkhan Myrzabek - Backend Developer

Stack of the project: **PyTorch, Fast.ai, OpenCV, Celery, Flask, Redis, Docker, Angular.js, Node.js, Tailwind, MongoDB**

3.3 Requirements and risks

Non-functional requirements:

- The qualities of conversion must be good
- The application must have feedback part
- The application must be hosted on services with good CPU qualities
- The application must convert black-white to photos to colorized in 5-10 secs
- The application must be responsive (work well and look good on all screen sizes)
- The application must be able to support 20 simultaneous users

Functional requirements:

- The application must send the result to the user's email
- The application must have a landing page
- The application must save result images on the database
- The application should send requests asynchronously
- The application must use **Docker** [9] to containerize the project
- The application should do compression of input photos for a faster process

Risks:

- A trained model will not be able to show high results
- Conversion black-white videos may slow down website performance
- Our server GPUs will not be able to withstand the load of the site

3.4 System design

3.4.1 Introduction

In this subsection, we describe the system design and architecture part of our service. The main purpose of system design is the continuous work of the app. The system architecture is the part where we don't have the right answer, all options will be right but we need to choose the fastest-acting from this one.

3.4.2 System requirements

As we have a deep learning model with large weights (2GB), our system needs to contain at least 16 RAMs, and a powerful CPU with at least 4 cores starting from Intel-Core i7. Also, we need about 20 GB of hard disk. Our app is wrapped in **Docker-Compose** [9] file so the operating environment doesn't matter.

3.4.3 Architecture

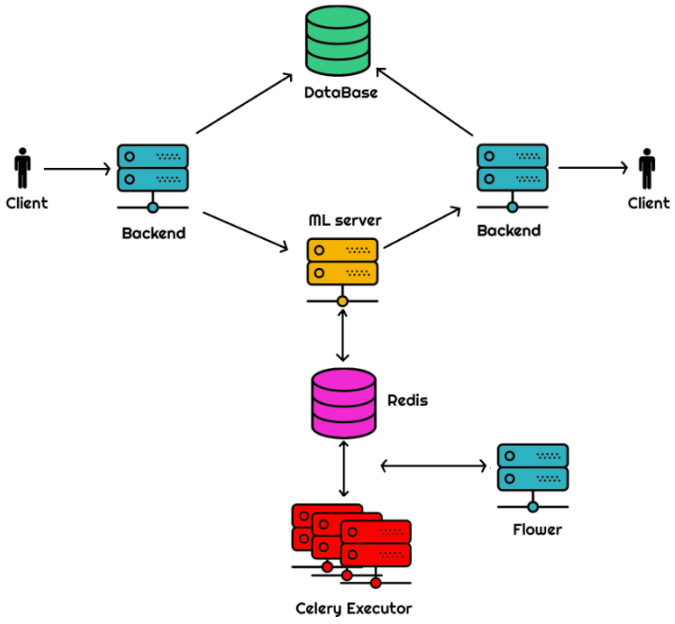

Figure 3.2: System architecture

At the start of the cycle, client need to push some image and e-mail. Backend service redirects this image and email to the machine learning server and saves it in **MongoDB** [2]. Machine learning server works asynchronously and after getting task it saves it in **Redis**, if we have freely workers **Redis** [27] assign a task to it. When the worker finished the task it send it back to the Backend. Backend after getting colorized image sends it by e-mail to user and update this image in **MongoDB** (**Figure 3.2**). We can freely monitor task executions in the **Flower** [11] (**Figure 3.3**).

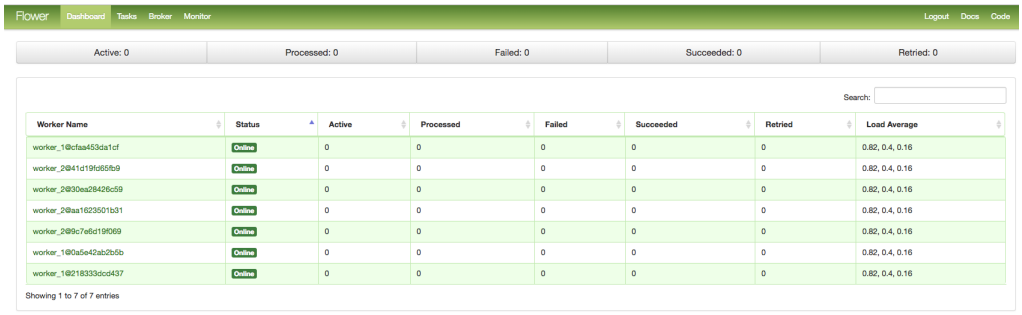

Figure 3.3: Flower UX/UI

Advantages of this architecture is

- Good basis point for other projects
- **Redis** with **Celery** [6] executor makes platform scalable and reduces costs for GPU
- **Flower** monitoring system helps manage complexity
- **Docker-Compose** with images and **Docker** files reduces deployment to production

3.4.4 Input Output Format

Input format consists of image files with png, jpg, jpeg format. On output, we send a colorized images to the user's email. Email option is great for this project because if a client uploads a lot of images with high quality, the model will colorize it very long, so the client won't wait for readiness.

3.5 Implementation

3.5.1 Frontend

Introduction

In this subsection, we describe how to create the client side of our service. As the service is a complete single-page application (**SPA** [17]), we have refused to write the project in pure **JavaScript**. There are a lot of reasons for that, but the main ones are the abundance of code. Writing in native **JavaScript** is always a lot of code. That is why we immediately started looking for the framework we needed.

Framework

What tasks should our framework solve?

- Simple work with the markup. Most of the site is static code. It was very critical for us that this code could not depend on other parts of the site (in our case, the loading form)
- Working with **API** on a native level. We didn't want to use any third-party **API** services, because they could be closed or unsupported at any moment. It was also important that working with the **API** was very comfortable.
- Speed and scalability. ***Colorizen*** is not a very large project in terms of client-side, however, it would be desirable to be able to scale (for example, add the ability to process video)

We started to choose between 4 modern frameworks: **React** [26], **Vue** [34], **Angular** [3], **Svelte** [32].These four frameworks are the most popular and functional, but there are many differences between them. As a result, it was decided to build the client-side on **Angular** and there are many reasons for that:

- Component-based approach
- Popularity
- Completeness
- Support of **Typescript** [33]

Architecture

The concept of **SPA** (Single Page Application) was taken as the basis. We have a promotional page (landing page) consisting of 8 main sections:

- The welcome screen. This is the first screen that immediately tells the user where he has arrived.
- A photo upload form. The most basic element of the page.
- Features section - a brief narrative about our service in which we highlighted its main advantages.
- Section with examples of processing. Users can look at the "Before/after" of some photos and evaluate the quality of image processing.
- Block with video. In this video, we talk about how our project works and highlight some key indicators: accuracy, processing time, and a number of processed photos.
- Team. Introduction to the development team with an opportunity to go to their social networks (GitHub and LinkedIn)
- Blog. It hosts three articles where we talk about our service.
- The footer and feedback form. For easier navigation, we duplicated the menu with links and placed a feedback forms with comments and suggestions.

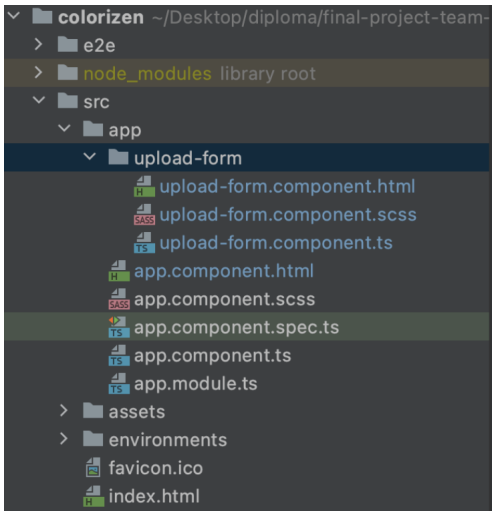

Figure 3.4: ***Colorizen*** client architecture

We used a component approach. Since most of the project is static. So we decided to make a separate component only for the loading form because it's an important functional object that has its logic. So ***Colorizen*** - has only two components, but later it will play a significant role in scaling the project. More about each component:

- *app.component.html* - the main and root component of the application. This is where the layout, styles, and logic of the whole page are described. We have used some third-party libraries for the layout:
  - **Bootstrap** [5] (for grid layout of site elements).
  - **particles.js** [24] - for background animation
  - **wow.js** [38] - for animations and smooth page scrolling
  - **Two-up** [14] - for gallery creation with service results (before/after photos)
- *Upload-form* - separate component for upload-form.

How does the image upload form work?

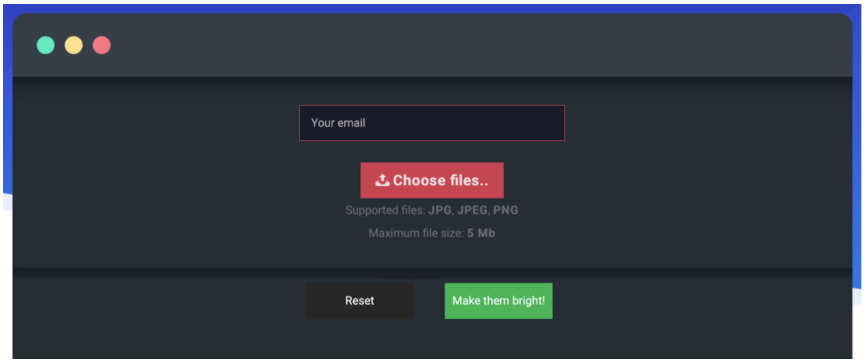

Figure 3.5: Upload form

1. The user has to enter the email to which the photos will arrive when they are ready. There is a check for the correctness of the input data to prevent the application from malfunctioning.
2. The user then chooses the photos he wants to process. There are some limitations. Only JPG, JPEG, and PNG photos are accepted. There is also a limit on the maximum size of a single file - 5 Mb.
3. If all the data is correct, the user can send photos for processing. After sending, the user sees a modal window that says that the photos have been sent for processing.

Upload form work logic

The *handleUpload()* method - this method is called when the upload form becomes active. This is where the conversion of the incoming file into the base64 format takes place. Base64 is a format for representing an image as text using 64 ASCII

characters. We chose it for convenience of storage. We store images as text in **MongoDB** [2], which allows us not to store image files, making the process of retrieving an image much faster. This approach has a disadvantage: the weight of the resulting image is on average 15-20 percentages. But this is not a big problem since we do not store the image for more than two weeks due to privacy policy.

The *sendPhotos()* method is responsible for sending photos to the server. Thanks to the built-in **RxJS** library [29] we can manipulate the state of sending photos. We send an array of objects in which we have the user's email address and images in base64 format along with a sequence number (ID). The serial number is needed for the model to iterate through the array and process the photos in the order in which it received them. After sending the photos successfully, the form is cleared with the *resetForm()* method.

3.5.2 Design

Detailed work was carried out on the design of the user interface. Design development began with a search for references. It is important to understand that the search for references is an important design process through which we can understand the design trends of similar apps or look for already reflected and working solutions. The main idea was to create a design that would be very comfortable, bright, catchy, and effective at the same time. The search for references was successful and we noted the main points that we wanted to present:

- **Relevance**. Using trendy techniques in design is a very effective strategy, but it's not our approach. We create a site that serves a specific function and we would not want to scatter the user's attention with various unnecessary but beautiful elements.
- **Simplicity**. We need to make the application easy to use. Minimum steps, minimum actions, maximum results.
- **WOW-effect**. Our task is to make a cool service. That's why this point is very important.

After discussing all the concepts and defining the structure of the site, we created a Wireframe. All in all, we have allocated 6 sections: welcome screen, download form, gallery, video section, FAQ, payment plans (**Figure 3.6**):

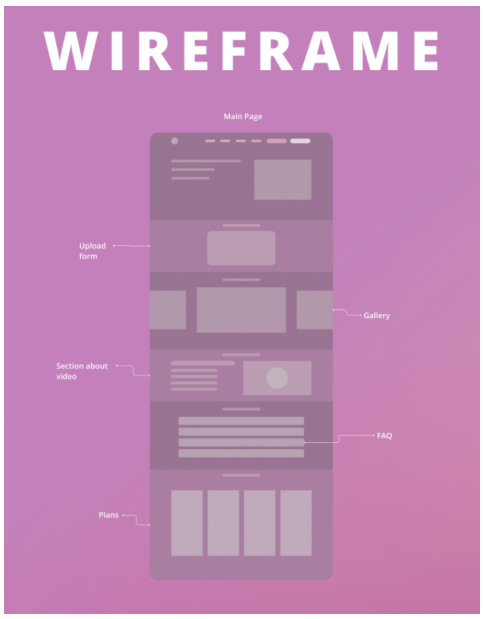

Figure 3.6: First version of our Wireframe

After approving the wireframe, we started to create the visual part. However, we did not like this design. We did a little research, interviewed a small group of people, and found out the weaknesses of our design. The main complaints were that the colors were too bright and the form was uncomfortable to use because it was not in the center of the user's attention. We took these remarks and put together a new wireframe (**Figure 3.7**):

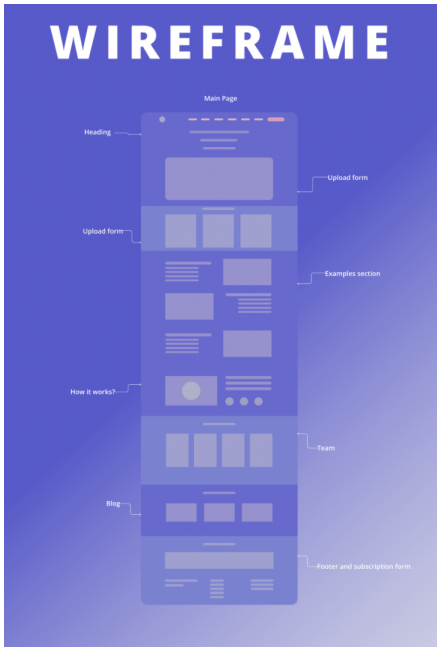

Figure 3.7: Final version of our Wireframe

Fixed points:

- Added/removed/corrected some sections of the site. The examples section has been changed. Now the user can see the result of photo processing as a before/after
- The block "How colorizen works". A small informative block describing the main points of the site.
- The new "Team" block. Added a block with the developers of the site.
- A new "Blog" block. Three articles with a short narration about some points of colorizen work.
- Added a footer and a form to subscribe to updates.

This time the focus group was satisfied with the new design. We freshened up the structure, added more space, and updated the color scheme. The focal color adds lightness and freedom to the design and doesn't strain the eye at all. The colors remained bright, but the styles of their use have changed (**Figure 3.8**).

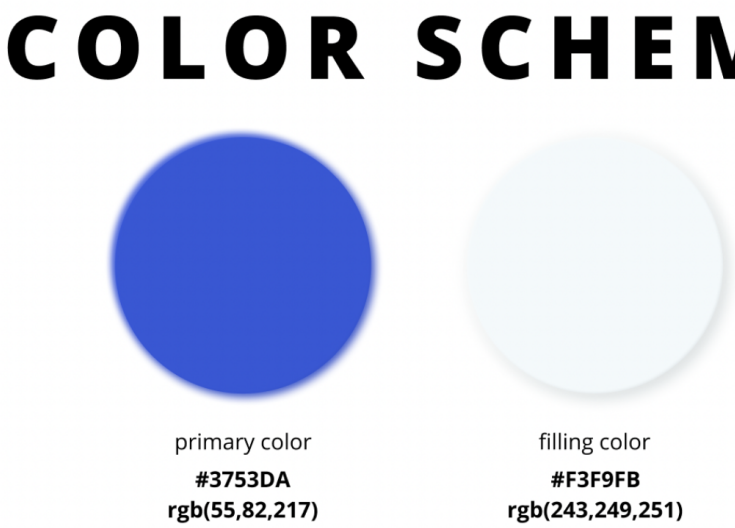

Figure 3.8: Color scheme

You can get acquainted with the old and new designs at the link:
`https://github.com/kdiAAA/Architectures/blob/main/design.md`

3.5.3 Backend

Framework

When writing the backend part of our project, we chose **Node JS** [21] along with the **Express JS** [10] framework. It was chosen because it is the best tool for our purposes. We also used several third-party libraries, such as:

- **mongoose** [20] - it's a library that makes working with **MongoDB** [2] very convenient and efficient
- **CORS** - it fixes the Cross-origin resource sharing problem. In our project, we use two server-side parts and this causes a conflict because browsers do not encourage the use of two data sources for security reasons.
- **Nodemailer** [22] is a library for sending emails using **Node JS**. We use it to send finished pictures to our users.
- **Request-promise** [1] is an add-on for the Request library. It allows us to send a request inside a request. To send the photos to the database and the model for processing at the same time, we use this library.
- **Base64-IMG** is a library for processing base64 strings back into an image.

Database

As a database, we used **MongoDB** [2], specifically its cloud version, **MongoDB Atlas**. We use a scheme from **mongoose** consisting of three keys (**Figure 3.9**):

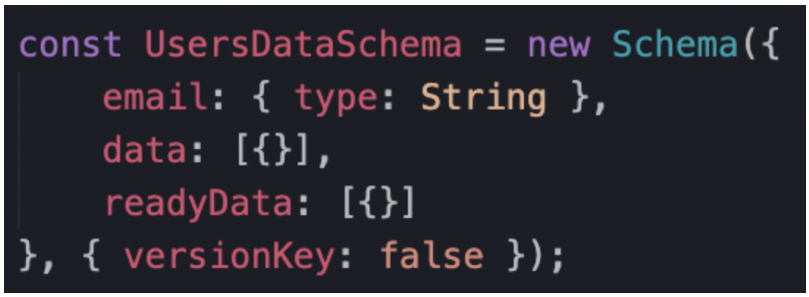

Figure 3.9: Database structure

- **email** - the user's email, to which we will send the finished color photos
- **data** - an array of objects that stores all the photos received from the user in base64 format, as well as the ID with the order number.
- **readyData** - an array of objects that stores the finished photos, ready to be sent

APIs

/api/newPhotoUpload is a **POST** request that receives data from a user. We get the data in **JSON** format as a body parameter and save it to a new **JSON** for further submission. As soon as the data arrives, we save it to the **MongoDB Atlas** cloud storage. We then send the same data to the model for processing using a 2way request. This has been achieved with the request-promise library.

/api/photosDone/:email is a **PUT** request, which is responsible for receiving new data from the model. It takes two parameters: email as params and readyData as body parameter. It's done so we can use the email to look for a user in the database and update the readyData key with the new data we received after processing. Also, inside this method, we convert base64 into a proper jpg-image file using base64-img. We are converting all the photos to the same format. Tests have shown that conversion from base64 to JPG is most efficient because less detail is lost during reverse processing.

Once the photo files have been created, an email is generated with these photos using the nodemailer library. Once everything is done, the email is sent to the address specified by the user.

3.5.4 Machine learning

Introduction

In this subsection, you will learn about the development process for the machine learning part of the project and deployment in productions. We conducted research on other technologies for converting pictures from black and white to color. Among them were models: simple **GAN** [13], **CycleGAN** [40], **DeOldify** Artistic model [4], Evolutionary **GAN** [35] and **DeOldify** Stable model [4]. From this list, we have selected **DeOldify** Stable model and described this model in the second chapter. Next, we will talk about the architecture and training of this model, as well as how we took the model out to production.

Training and fine-tuning models

We used a ready-made trained model since we did not have the resources to train this model. We used the already trained model on **ImageNet** dataset [8], where all the images were converted to black and white with some noise and trained to reverse them using the methods described in the second chapter. In general, the **SAGAN** generative model [39] is trained first, starting with a low resolution and with a constant increase up to 192px. Next, we train the critic model to

determine the original and fake images to the same resolution. Then we train these models together as in a normal **GAN** process using *Two Time-Scale Update Rule* (**TTUR**) [16]. We tried to fine-tune this model to get the best result on small datasets in the same way, but after many hours of training, the model did not change the results, it was decided to leave the model unchanged and deploy it to a production environment.

Generator Architecture

As a generator architecture was used a **ResNet 101** [15] backbone architecture with **U-Net** [28] with an emphasis on the depth of layers on the decoder side. **U-Net** and **ResNet** are types of *Convolutional Neural Networks* [19]. **U-Net** consists of Convolution Operation, Max Pooling, ReLU Activation, Concatenation, and Up Sampling Layers and three sections: contraction, bottleneck, and expansion section. **U-Net** uses a loss function for each pixel of the image. This helps in the easy identification of individual cells within the segmentation map. The basic architecture of **U-Net** is shown in **Figure 3.11**. This model is mainly used for semantic segmentation and medical image analysis. In our case, we use **ResNet** to downsampling our image (Feature extraction) and use the **U-Net** up-sampling architecture to get a colorized version of our image. **ResNet** consists of 3x3 filters, CNN down-sampling layers, global average pooling layer, and a 1000-way fully-connected layer with softmax in the end. **ResNet** solves the problem of the vanishing gradient problem, it's when weights of the first layer won't be updated correctly through the back-propagation. **ResNet** solves this problem by using the identity matrix. This preserves the input and avoids any loss in the information. This identity matrix is used in *Residual Block* (**Figure 3.10**), where each layer feeds into the next layer and directly into the layers about some hops away. Also on some level of the **U-Net** block, we add a *self-attention module* for avoiding gradient problems.

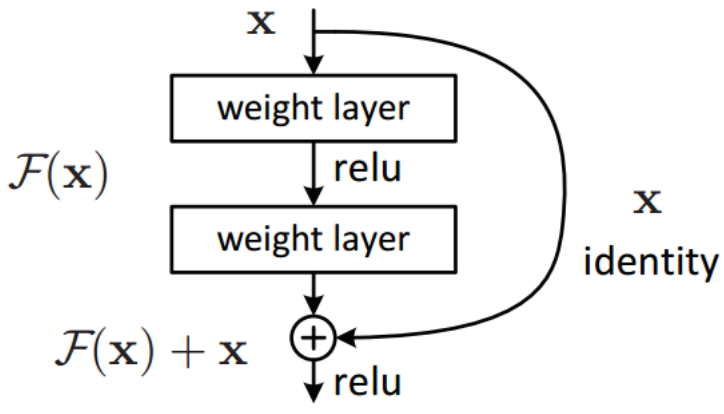

Figure 3.10: Residual Block, Source [19]

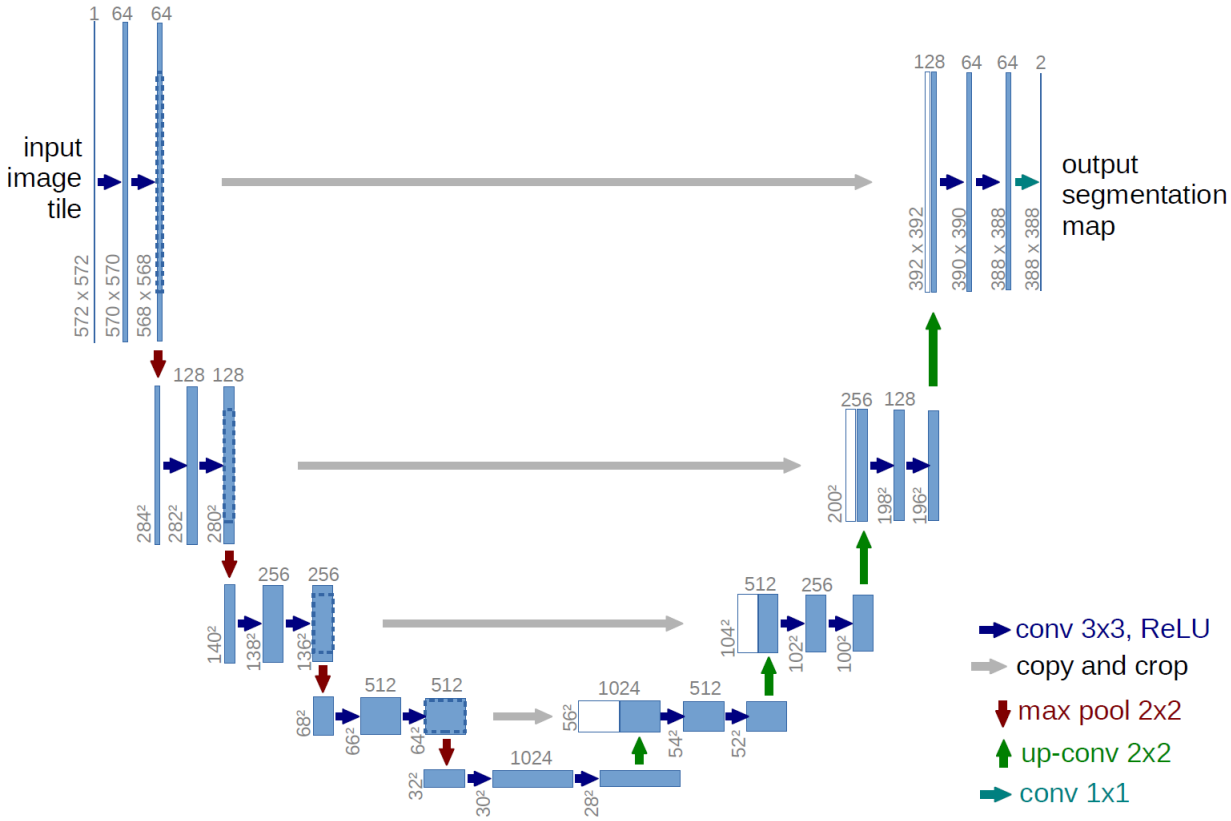

Figure 3.11: U-Net architecture, Source: [28]

You can learn more about our architecture in detail by clicking on the link below:
`https://github.com/kdiAAA/Architectures/blob/main/gen_arch.txt`

Discriminator Architecture

For the critic of the model was used the architecture of a conventional *Convolutional neural network* [19], which consists of Convolutional layers with LeakyReLU, Dropout layers to prevent over-fitting the model, and *Self-attention module*. As an output, we have a binary classification that determines whether the image is real or fake. We do not use the discriminator in production, since we have already trained the generator to convert images.
You can learn more about our architecture in detail by clicking on the link below:
`https://github.com/kdiAAA/Architectures/blob/main/crit_arch.txt`

Files and description

In this subsection, we will analyze the main files that are used in the machine learning part of the project:

- **device.py** - the file that configures what computer calculations the model will run on GPU or CPU

- **filters.py** - the file that makes preprocessing and postprocessing using **OpenCV** [23] and **Fast.ai** [36] before letting the image into the generator
- **generators.py** - the file that contains the generator itself uses a **ResNet** from a ready-made library of **Fast.ai** and implementation of **U-net** in **PyTorch** [25]

Model in Production

As we want our model to work asynchronously we used **Flask Server** [37] with **Redis** [27] and **Celery** [6] executor. To monitor executions of workers we also have **Flower** server **Flask** server have 3 **APIs**:

- **GET** method which ends with localhost:8000/. It's a health check method to verify if our model works correctly. It returns a response with **JSON** data.
- **POST** method which ends with localhost:8000/colorize. It's the method that takes data from **POST** query, parsed it to get email and images, and sends it to the **Redis** database. It returns task id.
- **GET** method which ends with localhost:8000/colorize/taskid. It's method check if the **Celery** executor done his work. If the task is done, it returns that task is done in **JSON** format, and inProgress otherwise

Redis database, Celery executor, Flower [11], and Flask app we run with Docker-Compose [9].

3.6 Results and discussion

As a result, we have an AI-service converter of old images to color. The model converts well in landscape and portrait photography (**Figure 3.12 - 3.14**). As you can see, after conversion, some moments in the photographs give off yellowness, but it is not directly reflected in the eyes. We also tested this model on small datasets with 25,000 photos to get an idea of the conversion quality and made sure that, among other options, this one model is more stable on different types of photos. We also checked all the photos for colors using *the colorfulness function* [7] and then made sure that the model translates to the desired color correction.

Figure 3.12: Elvis Presley

Figure 3.13: Grant's Bar, New York City, 1956

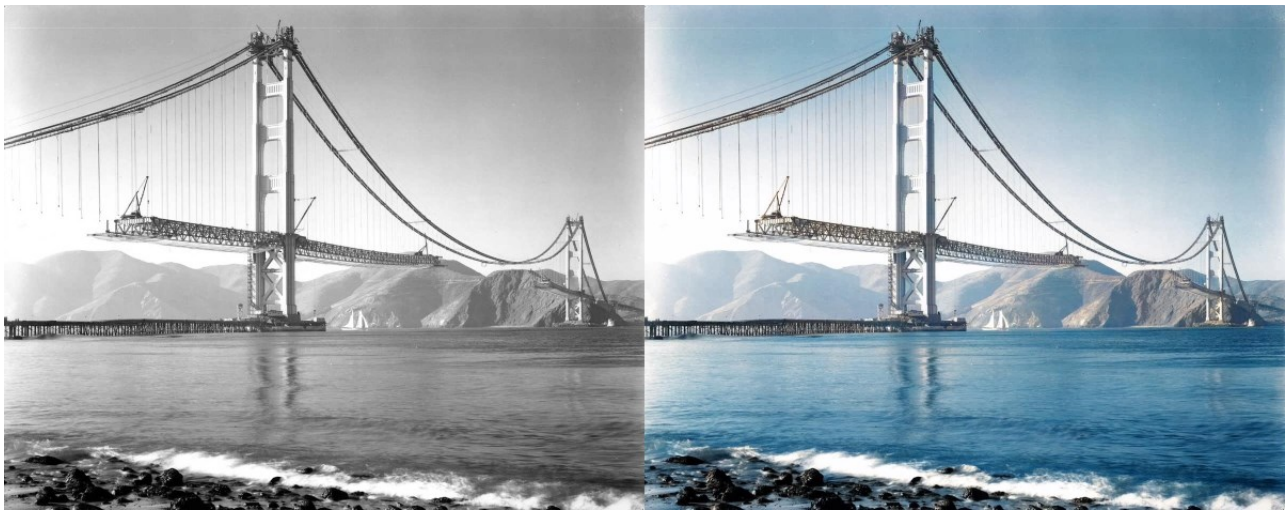

Figure 3.14: Building the Golden Gate Bridge, 1937

Chapter 4

Conclusion

So, finally, we have created a fully thought out web application called "Colorizen". This application, as we said, converts black and white photos to color using artificial intelligence. First, we learned how to build a model that will understand what color to use to complete any kind of photography. Then, by training the model, we got better-colorized photographs. To do this, we needed the use of popular libraries for deep learning such as **PyTorch** [25] and **Fastai** [36]. We also turned the **APIs** to our model for asynchronous requests and speed of service execution. As a front end, we came up with the decision to use **Angular** [3] because it is maintained and controlled by Google, and a large community of developers from all over the world has contributed to this library. In the back, we start using **Node.js** [21] and **Flask** [37] due to their high performance. We used **MongoDB** [2] as the database because it works great with **Node.js**.

In conclusion, we will say that the work done during the semester was perfect. New technologies were applied, and we found ways to solve different problems. There were also times when we had a lot of mistakes, but that gives us more passion for this project.

References

[1] *"Request-Promise." Npm.* Accessed 14 May 2021. URL: https://www.npmjs.com/package/request-promise.

[2] *"The Most Popular Database for Modern Apps." MongoDB.* Accessed 14 May 2021. URL: https://www.mongodb.com.

[3] *Angular.* Accessed 14 May 2021. URL: https://angular.io/.

[4] Jason Antic. *DeOldify.* 2018. URL: https://github.com/jantic/DeOldify.

[5] *Bootstrap.* Accessed 14 May 2021. URL: https://getbootstrap.com/.

[6] *Celery - Distributed Task Queue — Celery 5.0.5 Documentation.* Accessed 14 May 2021. URL: https://docs.celeryproject.org/en/stable/.

[7] *Computing Image 'Colorfulness' with OpenCV and Python.* 5 June 2017. URL: https://www.pyimagesearch.com/2017/06/05/computing-image-colorfulness-with-opencv-and-python/.

[8] Jia Deng et al. "Imagenet: A large-scale hierarchical image database". In: *2009 IEEE conference on computer vision and pattern recognition.* Ieee. 2009, pp. 248–255.

[9] *Empowering App Development for Developers | Docker.* Accessed 14 May 2021. URL: https://www.docker.com/.

[10] *Express - Node.Js Web Application Framework.* Accessed 14 May 2021. URL: https://expressjs.com/.

[11] *Flower - Celery Monitoring Tool — Flower 1.0.0 Documentation.* Accessed 14 May 2021. URL: https://flower.readthedocs.io/en/latest/#.

[12] *Generative Adversarial Networks | TJHSST Machine Learning Club.* Accessed 13 May 2021. URL: https://tjmachinelearning.com/lectures/1718/gan/.

[13] Ian J Goodfellow et al. "Generative adversarial networks". In: *arXiv preprint arXiv:1406.2661* (2014).

[14] *GoogleChromeLabs/Two-Up.* 2018. URL: https://github.com/GoogleChromeLabs/two-up.

[15] Kaiming He et al. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[16] Martin Heusel et al. “Gans trained by a two time-scale update rule converge to a local nash equilibrium”. In: *arXiv preprint arXiv:1706.08500* (2017).

[17] *HUSPI. Software Development IT Consulting.* Accessed 14 May 2021. URL: https://huspi.com/.

[18] Tero Karras et al. “Progressive growing of gans for improved quality, stability, and variation”. In: *arXiv preprint arXiv:1710.10196* (2017).

[19] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems* 25 (2012), pp. 1097–1105.

[20] *Mongoose ODM v5.12.9.* Accessed 14 May 2021. URL: https://mongoosejs.com/.

[21] *Node.js.* Accessed 14 May 2021. URL: https://nodejs.org/en/.

[22] *Nodemailer.* Accessed 14 May 2021. URL: https://nodemailer.com/about/.

[23] *OpenCV Library.* Accessed 13 May 2021. URL: https://opencv.org/.

[24] *Particles.Js – A Lightweight, Dependency-Free and Responsive Javascript Plugin for Particle Backgrounds.* Accessed 14 May 2021. URL: https://marcbruederlin.github.io/particles.js/.

[25] *PyTorch.* Accessed 13 May 2021. URL: https://www.pytorch.org.

[26] *React – A JavaScript Library for Building User Interfaces.* Accessed 14 May 2021. URL: https://reactjs.org/.

[27] *Redis.* Accessed 14 May 2021. URL: https://redis.io/.

[28] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. “U-net: Convolutional networks for biomedical image segmentation”. In: *International Conference on Medical image computing and computer-assisted intervention*. Springer. 2015, pp. 234–241.

[29] *RxJs | Angular.* Accessed 14 May 2021. URL: https://angular.io/guide/rx-library.

[30] Karen Simonyan and Andrew Zisserman. “Very deep convolutional networks for large-scale image recognition”. In: *arXiv preprint arXiv:1409.1556* (2014).

[31] *Software Development Process: How to Pick The Process That's Right For You.* Accessed 13 May 2021. URL: https://plan.io/blog/software-development-process/.

[32] *Svelte • Cybernetically Enhanced Web Apps.* Accessed 14 May 2021. URL: https://svelte.dev/.

[33] *Typed JavaScript at Any Scale.* Accessed 14 May 2021. URL: https://www.typescriptlang.org/.

[34] *Vue.Js.* Accessed 14 May 2021. URL: https://vuejs.org/.

[35] Chaoyue Wang et al. "Evolutionary generative adversarial networks". In: *IEEE Transactions on Evolutionary Computation* 23.6 (2019), pp. 921–934.

[36] *Welcome to Fastai.* Accessed 13 May 2021. URL: https://docs.fast.ai/.

[37] *Welcome to Flask — Flask Documentation (2.0.x).* Accessed 14 May 2021. URL: https://flask.palletsprojects.com/en/2.0.x/.

[38] *Wow.Js — Reveal Animations When Scrolling.* Accessed 14 May 2021. URL: https://wowjs.uk/.

[39] Han Zhang et al. "Self-attention generative adversarial networks". In: *International conference on machine learning.* PMLR. 2019, pp. 7354–7363.

[40] Jun-Yan Zhu et al. "Unpaired image-to-image translation using cycle-consistent adversarial networks". In: *Proceedings of the IEEE international conference on computer vision.* 2017, pp. 2223–2232.