Doble Grado en Ingeniería Informática y Matemáticas
Curso académico 2020–2021

UNIVERSIDAD
DE GRANADA

Trabajo de fin de grado

Image lightening with deep learning techniques

AUTOR:
DAVID CABEZAS BERRIDO

Supervisado por:
Fernando Berzal Galiano
Departamento de Ciencias de la Computación e Inteligencia Artificial

Summary

Abstract

Image lightening with deep learning techniques

The second part of this thesis focuses on the development of a tool for lightening extremely dim images. We design, implement, and document a deep learning technique to achieve this goal. Our project is composed of two main parts: the production of the generative deep learning model for image lightening and the creation of a tool for non-expert users. In the first part, we introduce the necessary theoretical concepts of machine learning and neural networks and give an in-depth explanation of the construction and training of our model. We also compare the performance of the model with that of a simpler technique: histogram equalization. The second part of this project pursues making the generative model usable to non-expert users by developing a web app that applies the model to a submitted image. Then, we shall deploy the application in a cloud computing service to make it steadily available to the public.

Keywords: Neural networks; GAN; Web programming; Cloud computing; TensorFlow.

Resumen extendido

Abstract

Iluminación de imágenes basada en técnicas de deep learning

El objetivo de este proyecto es la creación de una herramienta para ajustar la iluminación de imágenes capturadas en ambientes extremadamente oscuros, donde las herramientas usuales para incrementar el brillo no logran resultados satisfactorios. Además, la herramienta debe estar al alcance de usuarios sin conocimientos técnicos.

Para corregir la iluminación de una imagen muy oscura utilizaremos *redes neuronales artificiales*, en concreto modelos generativos de deep learning. Para explicar con detalle la arquitectura del modelo que vamos a usar y el proceso de entrenamiento, dedicaremos un capítulo al desarollo de fundamentos básicos de *aprendizaje automático (machine learning)*, *aprendizaje*

profundo (deep learning) y deep learning generativo, así como al entrenamiento de redes neuronales y las estrategias para entrenar modelos generativos. Las *redes generativas adversarias (GANs)* cobrarán especial importancia, ya que utilizaremos una GAN para entrenar el modelo generativo que finalmente procesará las imágenes.

En una GAN, se utiliza una pareja de redes neuronales, una discriminativa (*discriminador*) y una generativa (*generador*). El generador produce imágenes a partir de ciertas entradas, y el discriminador recibe imágenes reales y generadas, y debe discernir de qué tipo es cada una. El generador intenta confundir al discriminador haciendo que sus imágenes parezcan lo más reales posibles, y este último trata de evitar ser confundido. Se usa una función de pérdida combinada y se entrenan en paralelo ambas redes, de forma que las mejoras de una ayuden a hacer mejor a la otra.

Durante la producción del modelo iremos abordando las tareas y dificultades propias de las estrategias de deep learning generativo para problemas de *image translation (traducción de imágenes)*, es decir, problemas en los que se transforma una imagen en otra. Revisaremos algunos casos de éxito de aplicaciones del deep learning en problemas de este tipo. También presentaremos otra alternativa más simple para iluminar imágenes oscuras, la *ecualización del histograma*, que usaremos como referencia para evaluar el desempeño de nuestro modelo.

Para poner el modelo generativo al alcance de usuarios no expertos, construiremos una aplicación web priorizando la simplicidad y la usabilidad. Seguidamente, encapsularemos la aplicación en un contenedor de Docker, que desplegaremos en un servicio de *cloud computing* como es Google Cloud para hacerlo accesible al público de forma ininterrumpida.

Tras la realización de cada una de las partes hemos conseguido resultados generalmente satisfactorios. El desempeño del modelo generativo supera con creces a la ecualización del histograma, si bien existen casos en los que nuestro modelo genera salidas indeseadas. El mayor inconveniente sería la pérdida de resolución, que es notable cuando la resolución de la imagen de entrada es relativamente baja (por debajo de 1000×1000 píxeles). Con respecto a la aplicación, hemos logrado una interfaz sencilla y eficaz para interactuar con el modelo, además de alojarla en un servidor en la nube para hacerla accesible las 24 horas a través de una URL pública.

En los párrafos que siguen, resumimos los contenidos de cada uno de los capítulos.

El primer capítulo describe dos posibles estrategias para nuestra tarea de iluminar imágenes oscuras. La primera es la ecualización del histograma ([1]), una técnica para aumentar el contraste de imágenes en blanco y negro. Describimos brevemente esta técnica y algunos de sus inconvenientes, así como variantes de la técnica para paliar éstos. Para aplicar esta técnica a imágenes a color, utilizamos representaciones alternativas a RGB, en concreto HSV y HSL ([2]), que describimos brevemente. Mostramos también un ejemplo de una imagen iluminada con esta técnica, que finalmente será la referencia del modelo generativo. La segunda estrategia es construir un modelo generativo basado en deep learning. En esta sección, justificamos el uso de esta estrategia recapitulando algunas aplicaciones exitosas de *redes convolutivas* y GANs a otros problemas que requieren procesar imágenes. Finalmente, comparamos la idoneidad de las dos alternativas atendiendo al comportamiento que debería presentar cada una en teoría.

El Capítulo 2 establece los fundamentos teóricos necesarios para proporcionar una descripción detallada de la arquitectura del modelo y del proceso de entrenamiento en el siguiente capítulo. Comenzamos con la identificación de cuándo un problema es apto para ser enfocado con técnicas de aprendizaje automático, y pasamos a redes neuronales, donde presentamos las funciones de activación más populares y los teoremas de aproximación universal (Teoremas 2.2.1 y 2.2.7), que aseguran la densidad de las redes neuronales en distintas clases de funciones y puede encon-

trarse en [3]. Este teorema supone un importante ejemplo de aplicación de resultados clásicos de Análisis Funcional, como son el teorema de Hahn-Banach, el teorema de representación de Riesz-Markov y el teorema de Lusin. Seguidamente, describimos los elementos necesarios para el entrenamiento de una red neuronal. En concreto, las funciones de pérdida, los algoritmos de optimización basados en el gradiente (enfocándonos en *gradiente descendente* y *Adam*) y el algoritmo de *retropropagación (backpropagation)* para el cálculo de las derivadas parciales de la función de pérdida. También explicamos algunos problemas que pueden afectar al proceso de entrenamiento y posibles soluciones a ellos: la *regularización* para prevenir el *sobreaprendizaje* y la *normalización por lotes* para el *problema del gradiente explosivo*. Posteriormente, introducimos las redes neuronales convolutivas y pasamos al deep learning generativo, donde presentamos la convolución transpuesta y los *autocodificadores variacionales*. Respecto al entrenamiento de modelos generativos, exponemos los problemas de las funciones de pérdida usuales e introducimos el entrenamiento por adversario. Para finalizar el capítulo, comentamos el problema de los artefactos en cuadrícula en los modelos generativos y sus posibles causas y soluciones. Las principales referencias consultadas para este capítulo son [4] y [5].

Dedicamos el Capítulo 3 a la producción del modelo generativo para ajustar el brillo de imágenes oscuras. En primer lugar, describimos el proceso de generación del dataset, que obtenemos postprocesando con distintos niveles de brillo las fotografías del proyecto [6]. Seguidamente, proporcionamos una explicación detallada de la arquitectura de la GAN que emplearemos, la cual puede encontrarse en [7]. El modelo generativo presenta una arquitectura *U-Net*, un autocodificador simétrico en el que cada capa del decodificador recibe como entradas la salidas de la capa anterior y de la capa simétrica en el codificador (ver Figura 3.3). Para la implementación utilizamos la librería TensorFlow-Keras y el entrenamiento se realiza en Google Colaboratory. Se especifican todos los parámetros tanto del modelo como del proceso de entrenamiento. A continuación, probamos el desempeño del modelo sobre el conjunto de test y lo comparamos con los resultados obtenidos con ecualización del histograma, mostrando cómo influyen diversos factores (resolución de la entrada, presencia de fuentes de luz, nivel de brillo inicial, etc.) en el rendimiento. Por último, proponemos una solución a un problema con la dimensión de las imágenes, que estará relacionada con la forma en la que se invoca al modelo.

El cuarto capítulo está destinado a la construcción y despliegue de una aplicación web que permita utilizar el modelo generativo a usuarios sin conocimientos técnicos. Tras listar una serie de características deseables, presentamos un esquema de la apariencia de la aplicación y la descripción de cómo se realiza la funcionalidad principal. Implementamos la aplicación usando Flask, y mostramos algunos test para comprobar su correcto funcionamiento. Finalmente, encapsulamos la aplicación en una imagen de Docker y la desplegamos usando Cloud Run (de Google Cloud) de modo que cualquier usuario pueda acceder al servicio en cualquier momento. Nos aprovechamos de la prueba gratuita de esta plataforma de cloud computing para ello, debido a que el modelo consume demasiados recursos como para ser desplegado indefinidamente de forma gratuita. También Aprovechamos para describir un intento fallido de aplicación web con TensorFlow para JavaScript que hubiese permitido hacer el modelo accesible de una forma diferente: se descargaría y ejecutaría el modelo en el propio navegador del usuario. De esta manera, el modelo hubiese quedado accesible en GitHub Pages indefinidamente en lugar de depender de la prueba gratuita de Google Cloud.

En el quinto y último capítulo, recogemos una valoración del proyecto en términos de resultados obtenidos, cumplimiento de los objetivos y la familiarización del autor con diversas técnicas y herramientas durante su desarrollo. Finalmente, recogemos algunas otras alternativas y mejoras al software que podríamos haber implementado en caso de disponer de más tiempo o recursos computacionales.

Palabras clave: Redes neuronales; GAN; Aplicación web; Computación en la nube; TensorFlow.

Contents

Image lightening with deep learning techniques

Introduction

Adjusting the brightness of a dark image is performed on many occasions. Nearly every tool for image editing allows you to do this task. However, what happens when the image is extremely dark? Maybe we need to capture a photograph in a really dim environment and we cannot afford the use of flash or a long exposure time, so the brightness of the resulting image would be unusually low. Classic lightness adjustment may not work when the original image is nearly all black, like these two examples.

In this case, more complex techniques are needed to increase the brightness of the image without distorting it. As we show in Section 1.2, deep learning techniques usually work well for image processing problems. Thus, we shall train a deep learning model that lightens extremely dark images. Then we will compare the performance of our model with that of another common technique: histogram equalization.

Finally, we shall build and deploy an interface to make our model accessible to non-expert users, such as a web application.

The inspiration for this project comes from [6], where the authors build a pipeline for postprocessing low light images based on a convolutional network. We want a version of this software that works on standard image formats, that are already postprocessed, instead of raw sensor data. We must be aware that the quality of our images will be lower, since our model will have less information from the input and we do not have the same computational resources. On the

other hand, our tool can be applied on more occasions since raw formats depend on the sensor and the pipeline in [6] must be trained for each raw format independently, while all cameras can produce JPEG and PNG images (once postprocessed).

In the process, we expect to achieve the main goal of the project, that the author faces the experience of developing software based on machine learning and making it accessible to the public. Besides, the author will apply the knowledge acquired during the Degree in Computer Engineering and will research about several concepts and techniques that may not be explicitly covered in any of the undergraduate courses completed by the author of this capstone project, such as generative deep learning (specifically GANs), cloud computing and a few basics about computational photography and image optimization (signal-to-noise ratio, exposure time, post-processing and histogram equalization). Some frameworks will be employed for the building and training of the deep learning model (TensorFlow-Keras) and the development of the web application (Flask), with the double objective of making the implementation easier and that the author becomes familiar with these tools.

Specifically, we shall cover the following.

1. **Background of image lightening:**

   In Chapter 1, we shall discuss two alternatives for lightening extremely dark images.

   The first alternative is a simple technique that increases the contrast of grayscale images, histogram equalization. We shall describe this technique, its variations, and how it can be applied to increase the brightness of dark images in RGB format.

   The second alternative is a deep learning model. We show successful cases of deep learning applications to image processing problems.

   Finally, we provide a theoretical comparison between these two alternatives, the pros and cons of these two methods according to the behavior they are supposed to exhibit.

2. **Neural network fundamentals**

   We shall devote Chapter 2 to develop the basics that will allow us to provide an in-depth explanation of both the architecture and the training process of the model we will use for our purpose.

   We start by identifying when a problem is suitable for applying machine learning or deep learning.

   Then, we shall delve into neural networks and the universal approximation theorem. We will also present the elements needed for training a neural network, such as loss functions, gradient-based optimization algorithms (gradient descent and Adam), and the backpropagation algorithm (how the gradient of the loss function can be computed). Besides, we shall explain two common problems that may cause training to fail and possible solutions to them (regularization to prevent overfitting and batch normalization to prevent exploding gradient).

   Next, we will present convolutions and move on to generative deep learning (with transposed convolutions and variational autoencoders) and training of a generative model, where the adversarial approach proves to be of great service.

   Finally, we shall introduce the checkerboard artifacts, a common problem in generative models. We will describe possible causes and solutions to this problem.

3. **Generative model for image lightening**

The third chapter is intended for the production of the generative model. We begin by describing the dataset generation process.

Second, we take advantage of the concepts introduced in the previous chapter to provide an in-depth explanation of the architecture of the GAN and its training process. It includes the tools employed (mainly TensorFlow-Keras) and the values of the parameters.

Then, we do an evaluation of the performance of the model over a test set, along with an empirical comparison with histogram equalization.

At last, we proceed to explain how the model will be invoked, fixing a problem with the shape of the input in the process.

4. Web application development

In this chapter, we will build and deploy a web application to make our model accessible to non-expert users. We start by listing the features that we want to provide.

Next, we expose a scheme of the appearance of the application and a description of how the main functionality will be carried out. Then, we build our application using Flask, present the final result, and perform some tests to check that it works properly.

Finally, we shall encapsulate our application in a Docker container and deploy it to Google Cloud so everyone can access this service. We make use of the free trial of Google Cloud, since our application requires too many resources and no cloud computing platform would host it for free.

Additionally, we describe an unsuccessful first attempt of a web application that would allow our software to work indefinitely instead of depending on Google Cloud free trial.

5. Conclusions and future work

We shall devote the final chapter to make an appraisal of the project: the author's experience and opinion on the result. Then, we list possible alternatives that could improve it. In other words, we discuss what we could have done if we had more time or computational resources to improve the resulting software.

CHAPTER 1

Background of image lightening

We shall devote this chapter to the discussion and theoretical comparison of different alternatives for increasing the brightness of an extremely dark image. We need a method to adjust the lightness of images, and we shall explore two alternatives: histogram equalization and deep learning models or neural networks.

In a first section, we shall present a simple technique to adjust the brightness of images, histogram equalization. The performance of this method will be the benchmark for our deep learning model.

Next, we shall recap some successful cases of applications of deep learning to similar problems, and thus we will justify the usage of a deep learning model (a neural network) to lighten dark images.

Then, we shall provide a comparison between the behaviors that the aforementioned two methods would have in theory, and we will discuss their suitability for the problem.

We shall also justify the usage of a deep learning framework to create our deep learning model.

1.1 Histogram equalization

Consider a grayscale image, where the intensity of each pixels lies in $[0, 255] \cap \mathbb{Z}$. The *histogram* of the image is a graphical representation of the intensity distribution.

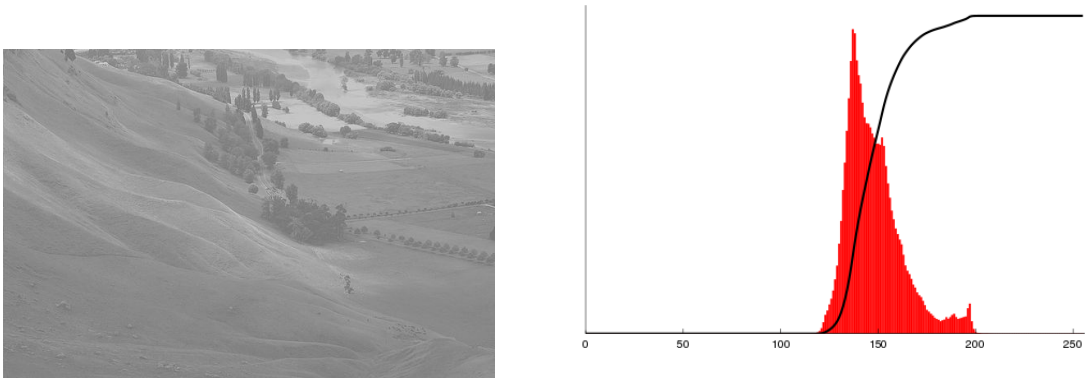


Figure 1.1: ([1]) A grayscale image along with its histogram (red). The abscissa represents the possible intensity levels for pixel, 0 for black and 255 for white; and the ordinate represents the number of pixels in the image with each intensity. The black line represents the sum cumulative sum of the number of pixels for each value, the cumulative histogram. The scale in Y-axis is omitted due to the histogram and the cumulative histogram having different scales.

As we can see in the previous figure, the intensities of all pixels of the image lie in a small range of the gray scale. *Histogram equalization* ([1]) improves the contrast in images by spreading out the intensity values of the pixels so the whole gray scale is fully covered and the cumulative histogram resembles a linear function. Consider the probability of an occurrence of a gray level $i \in [0,255] \cap \mathbb{Z}$: $p(i) = \frac{n_i}{n}$, the proportion of pixels of that level in the image. What we have called cumulative histogram is the *cumulative distribution function* (CDF). Histogram equalization applies the *inverse distribution function* (so called *quantile function*) to achieve its purpose, but we will not go into detail about how it is done.

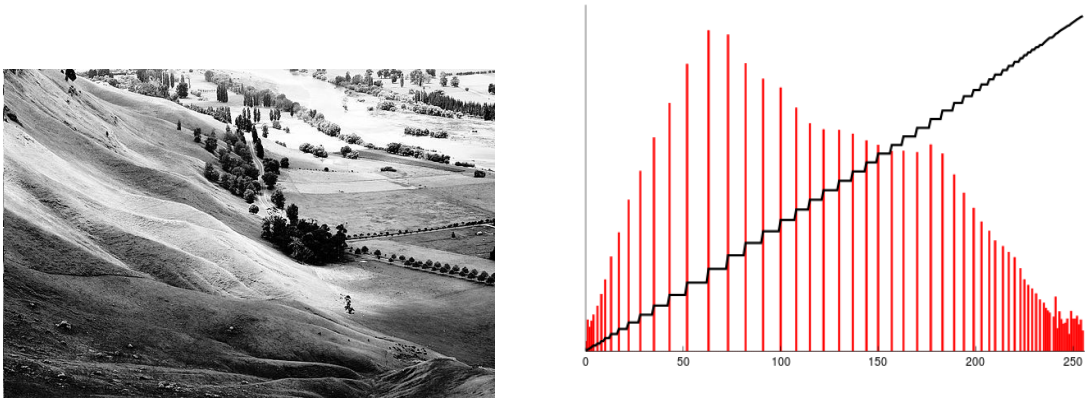


Figure 1.2: ([1]) The image contrast has greatly improved. The equalized histogram (red) covers the whole range of intensities, and the growth of the cumulative histogram (black) is more gradual.

Histogram equalization has a major disadvantage. The human perception of lightness is non-linear, it has the following form instead.

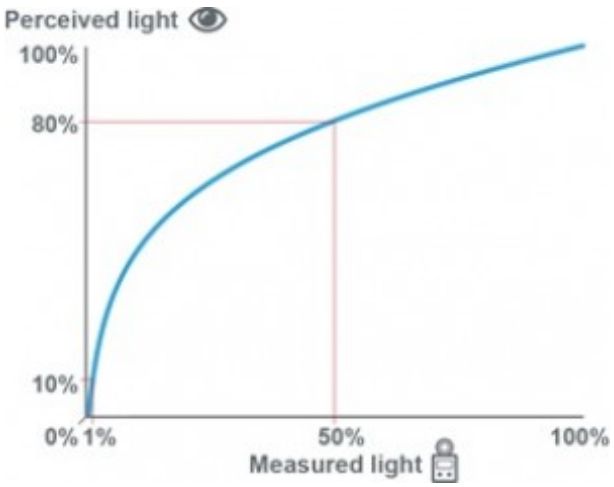
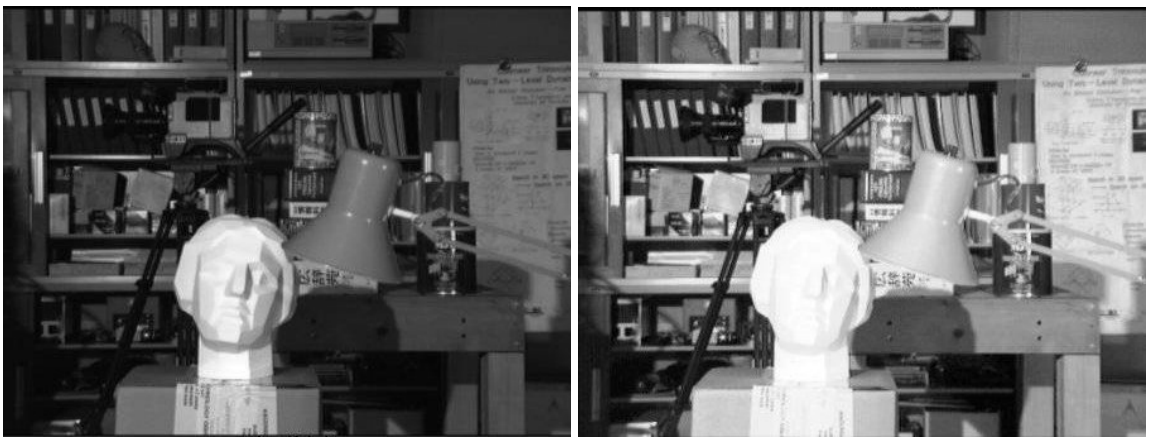


Figure 1.3: Human perception of light. When the light is dimmer, the human eye perceives changes with more ease than when the light is brighter.
Source: https://www.eldoled.com/support/learning-center/why-you-need-dimming-curves, Figure 4.

Therefore, although histogram equalization maps two shades of gray $a < b \in [0,255] \cap \mathbb{Z}$ to other shades of gray $f(a) < f(b) \in [0,255] \cap \mathbb{Z}$ such that $f(b) - f(a) > b - a$, we may not appreciate the contrast improvement. In fact, we may appreciate a worsening: $f(a)$ and $f(b)$ could look more similar to us than a and b. In other words, if an image has areas with very different brightness, histogram equalization produces a lower contrast in the lighter regions to the human eye, even if the contrast in those regions actually improves.



(a) Original image. (b) Equalized image.

Figure 1.4: ([1]) For our perception, there is a worsening of the contrast in the plaster portrait area.

To solve problems of this kind, **Adaptive histogram equalization** (AHE) is applied. The image is divided in tiles, histogram equalization is applied to each tile and then borders are bilinearly interpolated to soften the tile joints. However, this technique overamplifies the contrast in near-constant regions of the image, since the histogram in such regions is highly concentrated, this may lead to noise amplification.

Clip limit AHE (CLAHE) aims to avoid the overamplification by clipping the histogram at a

predefined value before computing the CDF. This limits the amplification of contrast and hence that of noise.

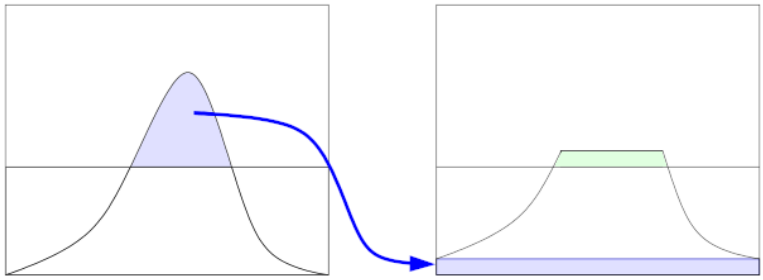


Figure 1.5: The part of the histogram that surpasses the limit is clipped and redistributed. Now the green is above the initial clip limit, this can be ignored or the process can be repeated until the excess is negligible.
Source: https://en.wikipedia.org/wiki/Adaptive_histogram_equalization.

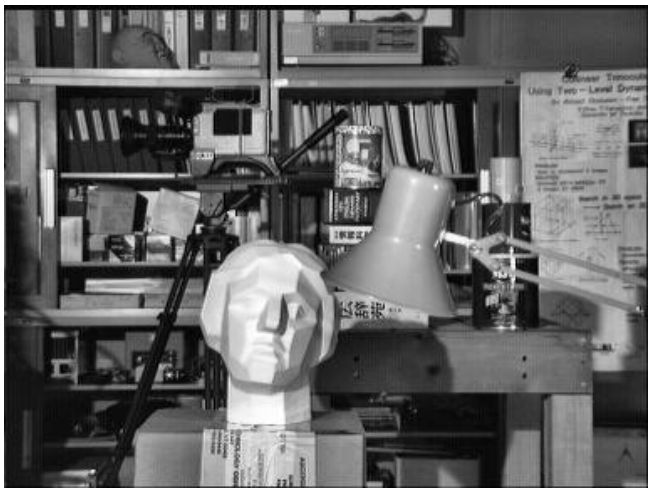


Figure 1.6: ([1]) CLAHE applied to the image in Figure 1.4(a).

CLAHE is generally better than simple histogram equalization. However, in our particular case all images will be dark and the problem about histogram equalization may not manifest. Therefore, there may be not need to use CLAHE, which entails a (limited) noise amplification.

These methods can be applied to improve the brightness of a dark image. Nevertheless, they cannot be applied to RGB images, since color balance is distorted. Therefore, we shall use different representations for our images. They can be found in [2].

HSL (hue, saturation, lightness) and HSV (hue, saturation, value) are alternative representations for the color of a pixel. HSL models how paints mix up together to form a color, and HSV represents how colors appear under light. In both representations, the hue dimension represents the "pure color" an observer perceives: blue, red, green, orange, etc. Not black, white or gray. And the saturation dimension represents the purity of that color, it may be an intense blue or it may be mixed with black or white. In HSL, the lightness dimension represents the amounts of black and white paint in the mixture, in other words, the lightness of the gray mixed with that pure color. In HSV, the value represents the brightness of the light that affects the color, it will appear darker when under a dim light.

These color spaces are better understood when represented geometrically. First, we shall recall the geometric representation of the RGB color space.

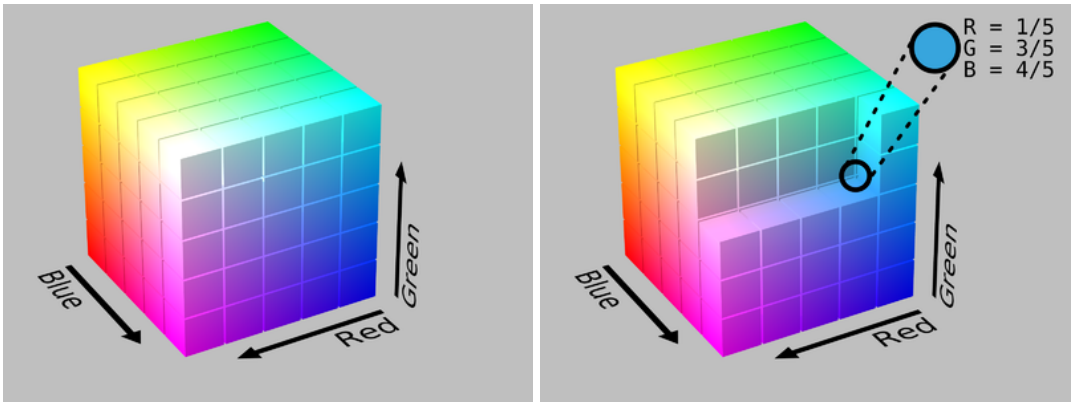
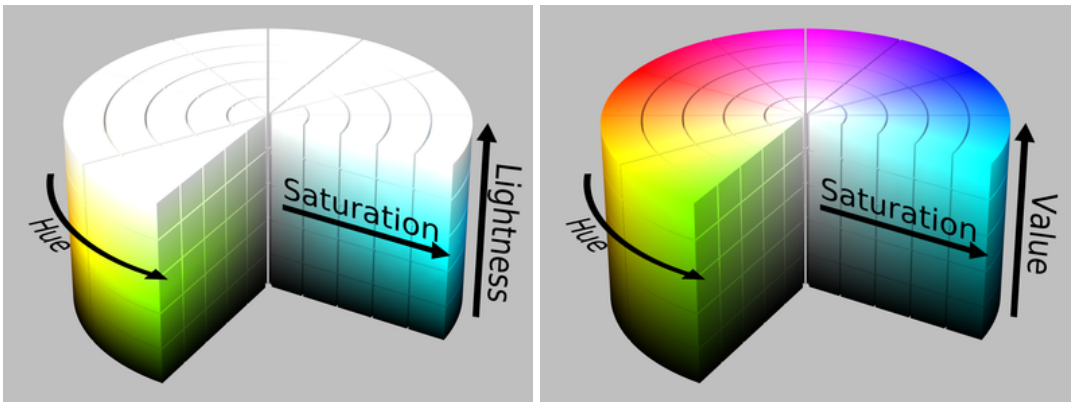


Figure 1.7: The RGB color space can be represented as a cube with three dimensions: Red, Green and Blue. Each point of the cube represents a color, obtained by mixing red, green and blue lights with different proportions. Each color is unique represented by a point of the cube. Source: https://en.wikipedia.org/wiki/HSL_and_HSV, Figure 6.

The HSL and HSV color spaces can be regarded as cylinders with hue as angular dimension and saturation as distance from the vertical axis. The height represents the lightness in HSL and the value in HSV.



(a) HSL. (b) HSV.

Figure 1.8: HSL and HSV color spaces as cylinders, each point represents a color. In HSL, both black and white are non-unique represented, while in HSV, only black is non-unique represented. The rest of the colors are uniquely represented in both spaces.
Source: https://en.wikipedia.org/wiki/HSL_and_HSV, Figure 2.

With the cylinders in mind, providing a description of the color spaces is much easier.

To select a color in HSL, we first choose a hue. Then, we mix black and white into a gray tone according to lightness. Finally, we mix the hue with the gray in proportions that depend on the saturation. The only exception is when pure white or pure black results from the mixture, in such cases, the saturation and hue are ignored.

To select a color in HSL, we first choose a hue. Then, we mix that hue with white according to the saturation. Finally, we project a light on the resulting mixture, the intensity of the light

depends on the value. If we shine no light, the result will be pure black regardless of the hue and saturation.

We will not cover the color conversion between RGB and these two color representations. Clearly, the transformations cannot be bijections, since there are colors (black in HSV; black and white for HSL) that have non-unique representations. However, RGB coordinates can be converted into HSL or HSV coordinates, and that the process can be reversed. This is due to colors having unique RGB coordinates and does not work the other way around, since exact HSV and HSL coordinates cannot be recovered from RGB (black can have different HSV and HSL coordinates).

Therefore, we shall apply histogram equalization and CLAHE on the lightness or value channel (depending in the representation) to improve the brightness of an image. Next, we shall show how both techniques in a test example for both color representations. We intend to transform the short-exposure image so it looks similar to the long-exposure one.

(a) Input, short exposure image. (b) Ground truth, long-exposure image.

Figure 1.9: Test example `sony-20005-x6.jpg`.

(a) HSL equalization. (b) HSV equalization.

Figure 1.10: Histogram equalization.

(a) HSL equalization. (b) HSV equalization.

Figure 1.11: CLAHE.

We appreciate the CLAHE produces images more similar to the ground truth in terms of contrast between the foreground and the background, while histogram equalization creates more contrast between different parts of the image. On the other hand, CLAHE creates more noise than histogram equalization. Besides, equalizing the L channel in HSL produces excessive illumination.

Nonetheless, this example is just meant to show how these methods can be used to light up dark images. We later encounter a problem of color noise amplification than both CLAHE and histogram equalization have for both representations. In fact, the just shown example is one of the few test examples where strong color noise does not appear. This problem affects both techniques, and there is not much difference in the performances of histogram equalization and CLAHE. We decide to use CLAHE as a benchmark for the performance of the deep learning model.

1.2 Deep learning models: neural networks

Neural Networks (NN), specifically Convolutional Neural Networks (CNN), have been proven appropriate for processing images as two-dimensional signals. Facial recognition or identification of certain elements are examples of problems with images as inputs.

Some examples of applications of convolutional neural networks to problems of this kind are listed below:

- LeNet-5 ([8]) outperformed current digit recognition methods, see [9].
- AlexNet ([10]) was one of the first machine learning models to perform better than human being in object recognition problems, see [11].
- YOLO, You Only Look Once ([12]). CNN for real-time object detection.

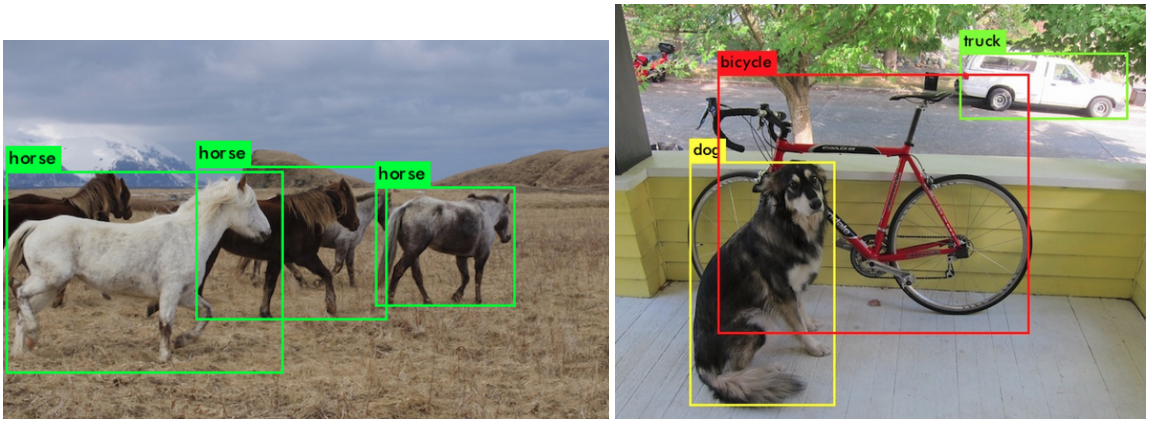


Figure 1.12: Objects detected by YOLO, it works at video rate.
Source: https://pjreddie.com/darknet/yolo.

Things get more complicated in image optimization problems or other problems when the outputs are images, in which case we need a generative model. Solving a problem with a Machine Learning approach implies training a model, which generally requires a loss function, that is, a criterion to measure how well the model is performing during training. We can use a wide variety of loss functions when the output of the problem is simple, like a probability for the existence of a face. However, designing a loss function to determine how satisfactory an image is is a non-trivial task. Adversarial neural networks (GANs) are created to solve this problem by incorporating another convolutional neural network to serve as a loss function for the generative model. Nevertheless, training a GAN requires a dataset of ground truth images that we would consider satisfactory outputs, which severely limits the usage of GANs in problems of this kind. Fortunately, we are able to get such a dataset, so a GAN may be a suitable approach for our problem.

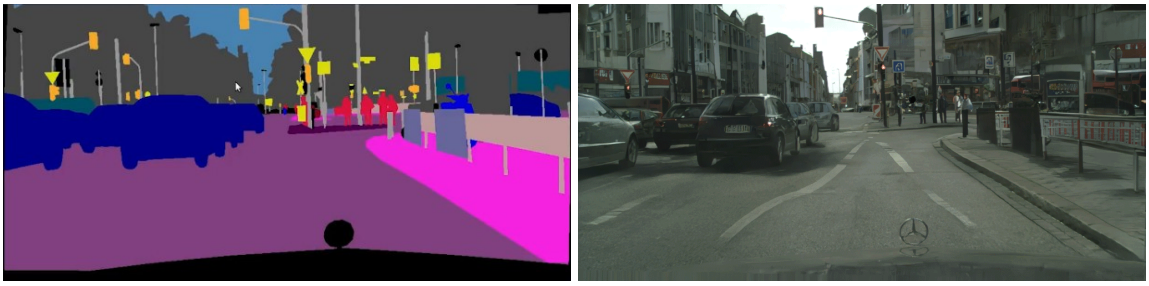
Some examples of applications of deep learning techniques to problems of this kind are listed below:

- NVIDIA researchers developed a GAN that synthesizes images from label maps, see [13].

(a) Input. (b) Output. (c) Edited style.

Figure 1.13: The GAN generates realistic images from label maps. It is also possible to change the style of the generated output interactively. Both outputs correspond to the same label map.
Source: https://tcwang0509.github.io/pix2pixHD.

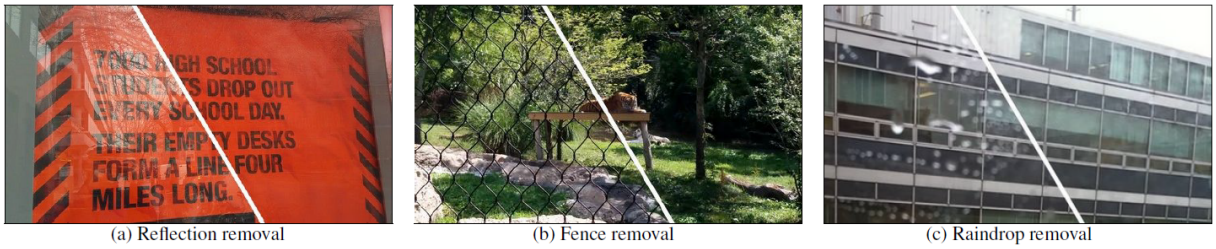
In addition, the label map can be edited interactively and the synthesized image applies the changes on the spot.



(a) Input. (b) Output.

Figure 1.14: The trees in the label map have been replaced by buildings, and the output reflects it.
Source: https://tcwang0509.github.io/pix2pixHD.

- Learning to See Through Obstructions ([14]). Although it is not a GAN, it uses convolutional neural networks to separate an image into two layers: background layer and obstruction layer. It allows to remove obstructions from images such as reflections and fences.



(a) Reflection removal (b) Fence removal (c) Raindrop removal

Figure 1.15: ([14, Figure 1]) This model removes several kinds of obstructions from images.

Using a deep learning *framework* for the production of the generative model has two important advantages. First, the framework offer building blocks for designing, training and validating deep neural networks through a high level programming interface ([15]). This makes the coding of the architecture much easier. Furthermore, deep learning frameworks generally provide automatic differentiation, which greatly simplifies the training process.

Second, most frameworks rely on GPU-accelerated libraries. A GPUs (graphics processing unit) allow faster image processing, and are one of the two main reasons why the popularity of deep learning approaches for image-related problems has increased in recent years. The other reason is the ease of obtaining and storing large datasets.

Therefore, we shall use a deep learning framework with GPU support. Among the list provided by NVIDIA in [15], TensorFlow and PyTorch are the most popular frameworks. TensorFlow is designed to program at a lower level than PyTorch is. However, it makes use of Keras, which is written in Python, as a high level API. We shall use TensorFlow-Keras for the development of our generative model because Google Colab allow us to use it online in Python3 notebooks, and also puts GPU at our disposal.

1.3 Theoretical comparison

We shall discuss advantages and disadvantages of both approaches: GAN and histogram equalization (specifically CLAHE), based solely on the behavior that these methods should have in

theory. We will provide an empirical comparison further in this document, once we get the results of both strategies on a battery of test examples.

On the one hand, the GAN entirely reconstructs a new image from the input. Potentially, we could obtain as output an image that is completely different to the ground truth (how the image should look with suitable lightness). This will not happen in practice, but blur, noise and undesired artifacts could appear in the GAN outputs (see Section 3.4(Failure cases)). Histogram equalization, on the other hand, only modifies the light (HSL) or the color intensity (HSV). Thus, the outputs will maintain the exact position of the edges and they will be sharper than the images generated by the GAN. This suggest that histogram equalization will have a better performance than GAN.

Conversely, histogram equalization only aims to increase the contrast in lightness and value channels, which may also increase the noise in the image. The GAN is not trained to increase the lightness explicitly, it is fed with correctly lit images (ground truths) and it attempts to replicate their quality. Hence, the GAN could learn to avoid noise amplification, since ground truth images do not present a high level of noise. This is a plus point for the GAN compared to histogram equalization but, as we show below, this becomes a specially important factor in the task of lighting up images. The references for what follows can be found in [16, Noise and ISO].

Signal-to-noise ratio (SNR) is a measure that compares the level of a desired signal to the level of background noise. In the case of images, the SNR for each component of a pixel is defined as the quotient

$$\frac{\text{mean component value}}{\text{standard deviation of component value}} = \frac{\mu}{\sigma}.$$

It is always desirable for the inputs of a problem to have a high SNR, which means "more information compared to noise".

During the exposure time in which a photography is captured, certain number of photons arrive to the lens of the camera, providing signal. Such number varies from exposure to exposure and from pixel to pixel, but we shall assume that is governed by the Poisson distribution. Fixing the exposure time, let λ be the average number of photons that provide signal to a pixel (or a component of a pixel) of the photo. The exact number of photons that arrive during the exposure time is a discrete random variable ruled by the following density distribution (Poisson distribution):

$$P_\lambda(k) = \frac{\lambda^k e^{-\lambda}}{k!}.$$

It is known that both the expectation and the variance are $\mu = \sigma^2 = \lambda$, so the standard deviation is $\sigma = \sqrt{\lambda}$. Therefore, we have

$$\text{SNR} = \frac{\mu}{\sigma} = \frac{\lambda}{\sqrt{\lambda}} = \sqrt{\lambda}.$$

Since the square root is a monotonically increasing function, we conclude that the signal-to-noise ratio grows with λ, the average number of photons that arrive during the exposure time. Clearly, the value of λ will be higher the more light in the environment. Therefore, photographs taken in extremely dim environments, which is our case, will lead to input images with extremely low SNR.

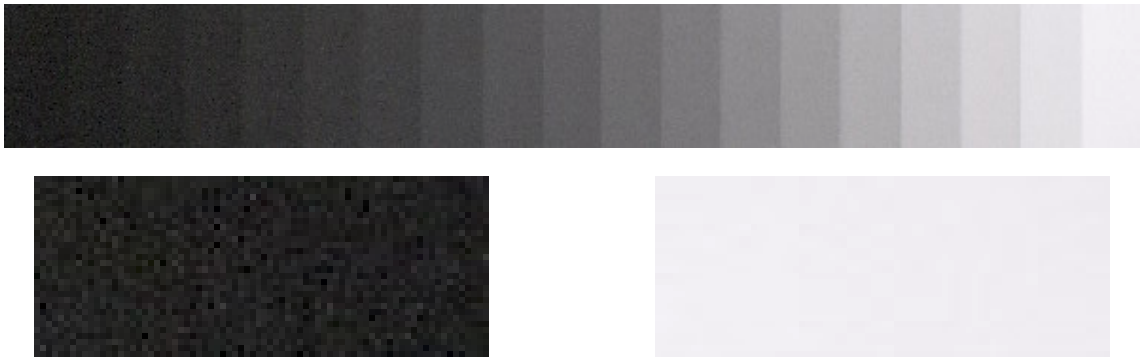


Figure 1.16: Q-13 Stepchart[1] captured with the Canon EOS-10D at ISO 1600. We include zoom of both ends. Even if the bright tile has more noise, it has much more signal, which makes the dark tile look more noisy. Actually, the pixels in the bright tile have more SNR than the pixels in the dark tile. Furthermore, the Canon EOS-10D provides software noise reduction, so the difference would be greater if the image was captured with a poorer camera.
Source: https://www.imatest.com/docs/noise.

The amount of noise in low SNR images, such as dark images, is a severe problem. A way to prevent noise amplification is highly advisable. Therefore, the neural network is more appropriate for this problem.

[1] Gray scale chart used to analyze the tonal response, noise, dynamic range and ISO sensibility of digital cameras and scanners. For more information, see https://www.imatest.com/docs/plain/q13.html.

CHAPTER 2

Neural network fundamentals

Throughout this chapter, we shall develop the basics that will allow us to provide an in-depth explanation of the architecture of the neural network we will use for our purpose, just as the algorithm used for the training process. We shall start from the concepts of *machine learning* and *deep learning*, and get to the adversarial approach for the training of a generative model.

2.1 Machine learning and deep learning

Our first step should be to position the concepts two concepts within the field of *artificial intelligence* (AI). About AI, we will limit ourselves to saying that it aims to simulate human intelligence in machines, specifically learning, reasoning and perception. Having in mind the following diagram, we shall proceed to introduce the concept of *machine learning*.

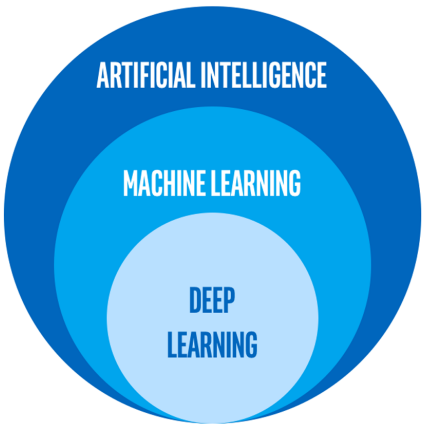


Figure 2.1: ([17, Figure 23]) *Deep learning* encompasses a subset of techniques that belong to *machine learning*, which is a discipline included within the field of computer science known as artificial intelligence.
Source: https://www.intel.es/content/www/es/es/artificial-intelligence/posts/difference-between-ai-machine-learning-deep-learning.html, [18].

According to [19], *machine learning* (ML) is the study of computer algorithms that improve automatically through experience. Those algorithms build a model based on sample (training) data in order to make predictions or decisions without being explicitly programmed to do so. This is specially useful in problems for which an explicit solving algorithm or method is really difficult to provide, while is relatively easy to solve concrete instances of the problem.

For example, is very complicated to give precise instructions to tell whether a email is spam or not, but we usually realize when a spam email manages to sneak into our inbox. As another example, try to provide a method that tells chihuahuas from muffins given a photography.

Figure 2.2: Chihuahuas and muffins can look really alike.
Source: https://www.freecodecamp.org/news/chihuahua-or-muffin-my-search-for-the-best-computer-vision-api-cbda4d6b425d.

The problem of deciding whether a photography has a suitable illumination or not fits perfectly with the aforementioned description, hence ML could help us solve our problem.

We shall focus on supervised learning, where the training samples comprise input-output pairs. Since we will provide the desired outputs for a group of input dark images to the algorithm, our problem belongs to that category.

A rule of thumb to determine if a problem is suitable to be solved by a ML approach could be: "The more data we have, the better the solution we can achieve". If a model trains with a vast and assorted set of examples, it is more likely to learn more details about the nature of the problem. Imagine that a ML model must decide whether or not an email is spam, if the model has trained with many (and varied) examples, then there is a high chance that the email resembles to other emails that the model has seen during training.

In [4, Section 1.1.1], we can find a more precise description of the components of a supervised learning problem, specifically a classification task. Simultaneously, we shall identify such components in the example provided in that same reference: "A client requests some credit to the bank. Should the bank give its approval?" There is no magical formula to resolve whether or not the bank should give the credit, but the bank has data of prior credit requests already assessed by experts, and it also keeps track of whether approving credit was profitable for the bank.

The input $\mathbf{x}$ is the credit request, with information such as annual salary, years in residence, outstanding loans, requested amount, etc. Given an input $\mathbf{x}$, we need to decide if the credit should be approved or denied. Therefore, we seek a function $f : \mathcal{X} \to \mathcal{Y}$, where $\mathcal{X}$ is the input space (all possible requests), Y the output space ($Y = \{\text{Approved}, \text{Denied}\}$) and f is the ideal formula for credit approval, known as target function. There is a data set $\mathcal{D}$ of input-output examples $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)$ where $y_n = f(\mathbf{x}_n)$ for $n = 1, \ldots, N$, credit requests corresponding to previous customers and the decision the bank considers correct. Finally, we need a learning algorithm $\mathcal{A}$, that uses the data set $\mathcal{D}$ to pick a formula (model) $g : \mathcal{X} \to \mathcal{Y}$ that approximates f. The algorithm chooses g from a set of candidates called the hypothesis set $\mathcal{H}$. The function

g will be used by the bank to base decisions about future credit requests, the decisions will be good only if g faithfully replicates f. To achieve that, the algorithm chooses the function g among the hypothesis in $\mathcal{H}$ that best matches f on the training samples, hoping it will continue to match f on new customers. Whether or not that hope is justified depends on many factors, such as the quality of the dataset (it should contain enough quantity and variety of examples), and the suitability of the learning algorithm and hypothesis set.

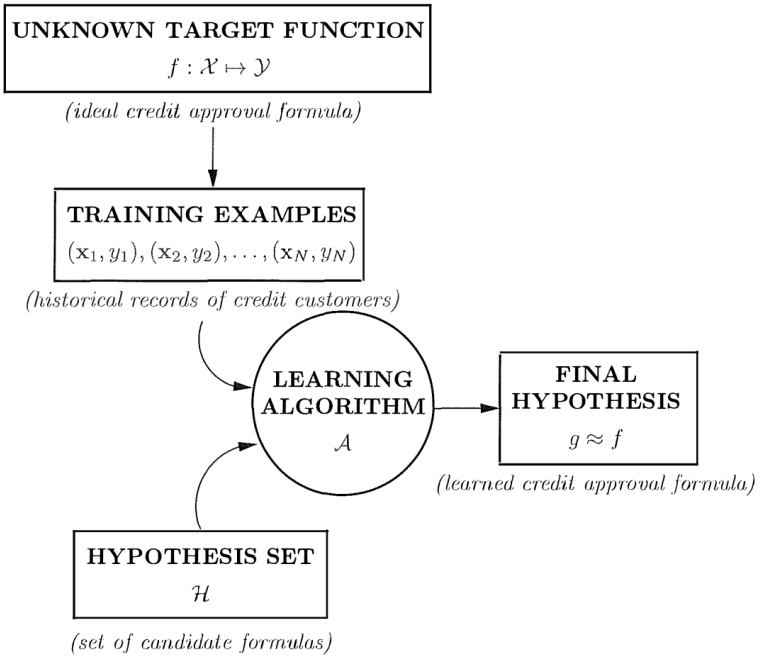


Figure 2.3: ([4, Figure 1.2]) Basic setup of a learning problem.

The choice of $\mathcal{H}$ can be crucial. Generally, we want to a family of functions as simple as possible, but containing functions complex enough to match the vast majority of training samples. The hypothesis set can include a wide variety of functions. We shall review some examples, for which we will consider the annual salary in, x_1 (in tens of thousands of euros), and the requested amount, x_2 (in hundreds of thousands of euros) as our only information about the requests. A simple hypothesis set to consider could be the set of all linear functions, whose elements would be of the form

$$g_{a,b,c}(x_1,x_2) = \begin{cases} \text{Approved} & \text{if } ax_1 + bx_2 + c \geq 0 \\ \text{Denied} & \text{if } ax_1 + bx_2 + c < 0, \end{cases}$$

with $a, b, c \in \mathbb{R}$. The training would consist of finding the values of a, b, c for which the function $g_{a,b,c}$ best matches f on the training samples. Here is an example with $a = \frac{1}{2}$, $b = -1$ and $c = 1$.

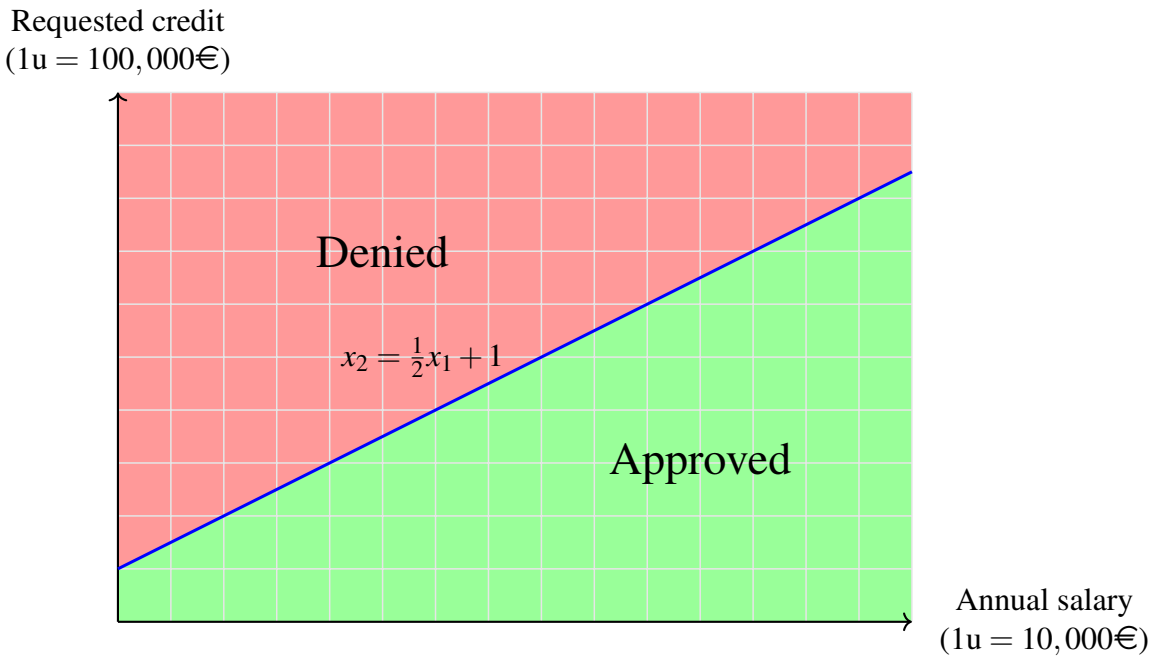


Figure 2.4: Example of a linear formula to approve or deny credit. Given values x_1 and x_2 for the annual salary and the requested credit respectively, the formula approves or denies the request depending on which region of the plane the point (x_1, x_2) is.

The consideration of a more complex class of functions as the hypothesis test leads to more complex formulas to make the decision. For example, the criterion: "Approve credits up to 400,000€, and if the customer earns 100,000€ or more a year, approve credits up to 800,000€" cannot be turned into a linear function, but can be expressed by the following decision tree.

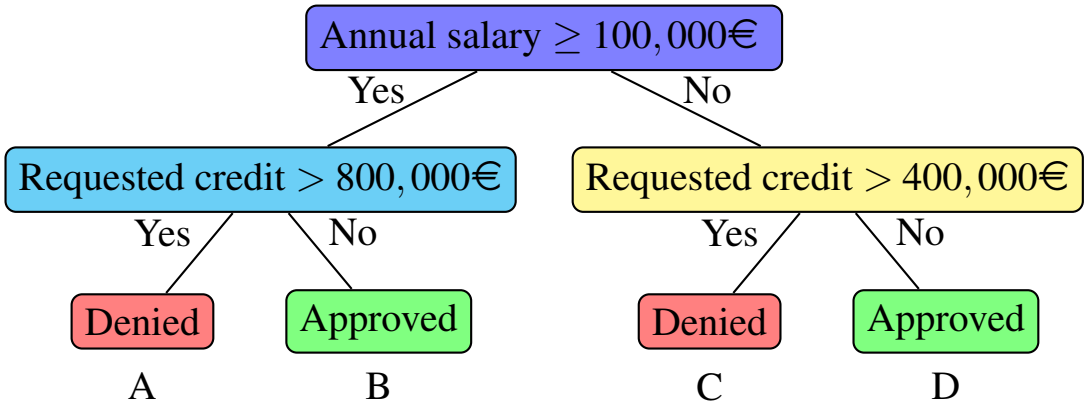


Figure 2.5: This decision tree provides a rule to approve or deny credit requests according to the aforementioned criterion.

The decision tree splits the plane into 4 rectangular regions (A, B, C, D). Requests in regions A and C will be denied by the formula, and requests in regions B and D will be approved. Also, the higher number of nodes in the tree, the higher number of regions the plane can be split into.

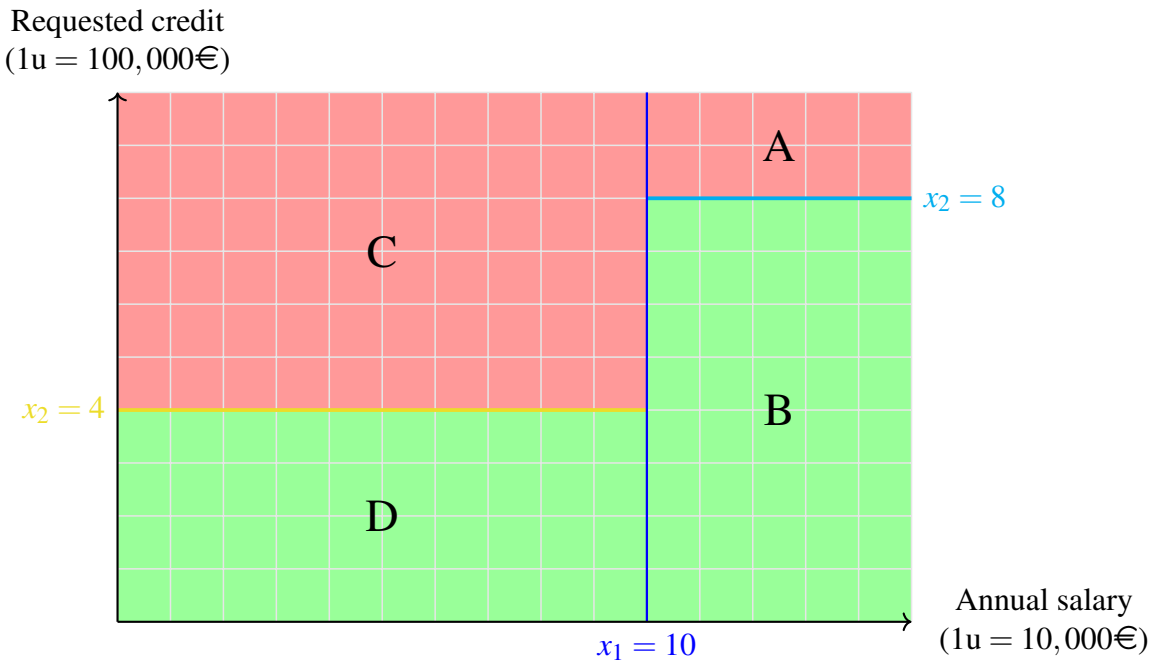


Figure 2.6: Given values x_1 and x_2 for the annual salary and the requested credit respectively, the formula approves of denies the request depending on which region of the plane the point (x_1, x_2) is. The dark blue node of the tree corresponds to the division of the plane produced by the dark blue line, and the same happens with the yellow and light blue nodes.

Now that we have seen examples of two different models (linear functions and decision trees), it is time to introduce a class of more complex models called *neural networks* (NN) or *artificial neural networks* (ANN). As stated in [17, Chapter 5], this models are inspired by the connections of the human brain. They are made up of several simple processing elements called neurons, that are connected and interact with each other, name. Neural networks have proven to be effective in a great variety of problems, some of which were previously almost unapproachable. This efficacy is due to their implicit representation of knowledge, through weights that model the connections between neurons. Even though this representation of knowledge makes neural networks hardly interpretable, it endows them with an ability to detect features and patterns in data that other models lack. The branch of machine learning devoted to the study of neural networks is known as *deep learning*.

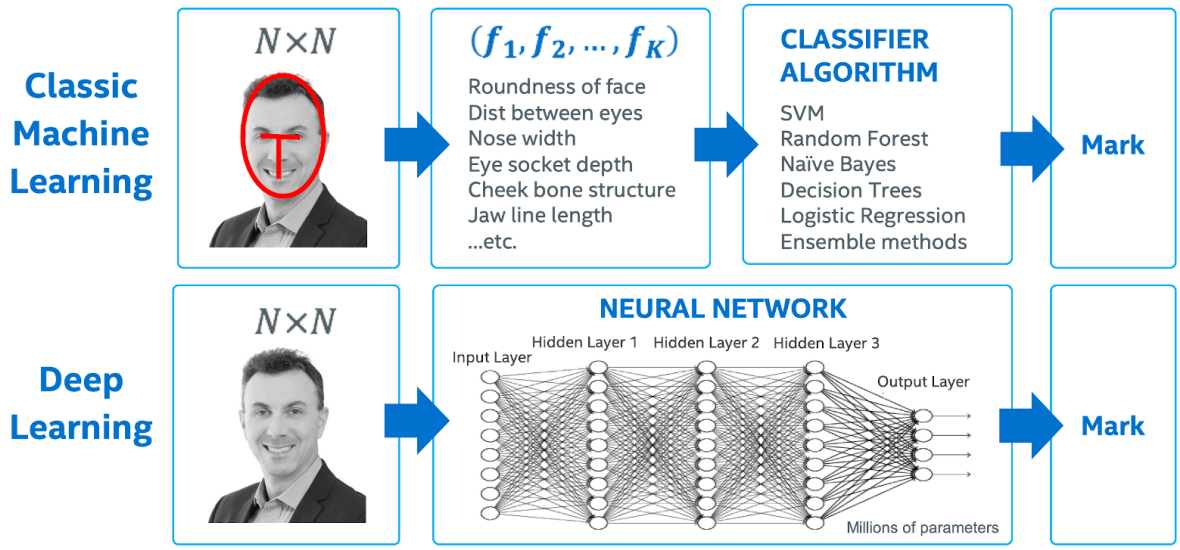


Figure 2.7: ([18]) Classic machine learning approaches for face recognition would require the data scientist to identify the set of features that uniquely represent a face (feature engineering), like the roundness of the face or the distance between the eyes. Whereas a neural network would only need the face images themselves as inputs.

2.2 Neural networks

So far, we have introduced an abstract idea of what deep learning means. Now, we shall describe an actual neural network in detail.

A *neuron* is a function that has several inputs and one output, an affine transformation is applied to the inputs by multiplied them by parameters called weights and adding them together plus a constant named bias, finally, a function h, known as activation function, is applied to the result, obtaining the output.

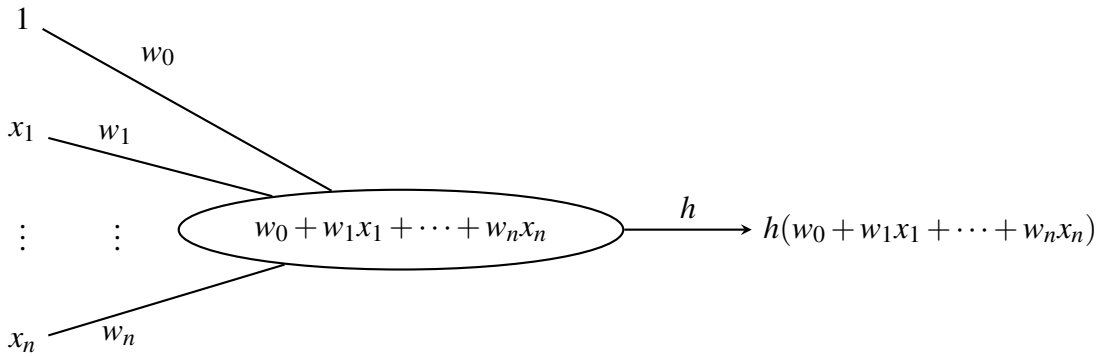


Figure 2.8: This neuron receives n inputs, to which it applies an affine transformation. The transformation depends on the weights $w_0, w_1, \ldots, w_n$, which are tunable parameters of the neuron. The bias can be regarded as a constant input, that is also multiplied by a weight. Finally, the function h is applied.

Neurons are meant to be connected to each other, using some neurons as inputs to others, which is equivalent to compose them as functions.

A *neural network* is a composition of neurons as functions, usually arranged in layers. The outputs of the neurons in the kth layer act as the inputs of the neurons in the $(k+1)$th layer. The first layer, is known as the input layer, where the inputs of the neural network are placed, typically this layer is not counted as a layer layer (it may be called the layer 0), since its neurons just output the values so they become the inputs of layer 1. The outputs of the last layer (known as output layer) are the outputs of the neural network itself. The rest of the layers are called hidden layers.

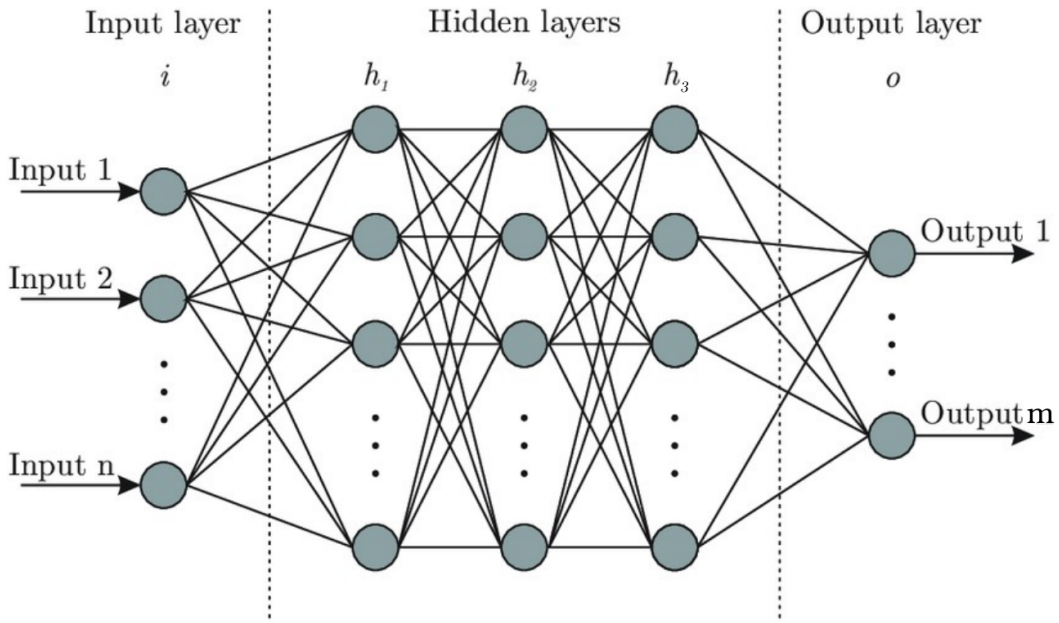


Figure 2.9: An example of neural network with n inputs and m outputs. The network has three hidden layers plus the output layer. The bias of each (non-input) neuron is omitted in the diagram.
Source: https://ai.plainenglish.io/my-notes-on-neural-networks-adf3e49657f8.

The greater the number of hidden layers and the more neurons they have, the more complex the neural network will be. The number of neurons in the input and output layers depends on the the problem that the network intends to solve. For instance, a network that receives a 64×64 pixels RGB image and determines if it is an image of a muffin or a chihuahua would have $64 \cdot 64 \cdot 3$ neurons in the input layer (3 channels for pixel, RGB) and could have one neuron in the output layer, which would tell the probability of the image being from a muffin.

Usually, the term *deep learning* is used to refer to *deep neural networks* (DNN), which are those with multiple hidden layers. There is no agreement in the number of layers a network needs to have in order to actually be considered deep. In [20], the author states that researchers in the field agree that deep learning models have more than one hidden layer, while a neural network with only one hidden layer would be considered a classic machine learning model.

2.2.1 Nonlinear activation functions

Since the composition of affine functions is an affine function, the entire network would be equivalent to a single layer in the absence of a non-affine function applied in each neuron. For

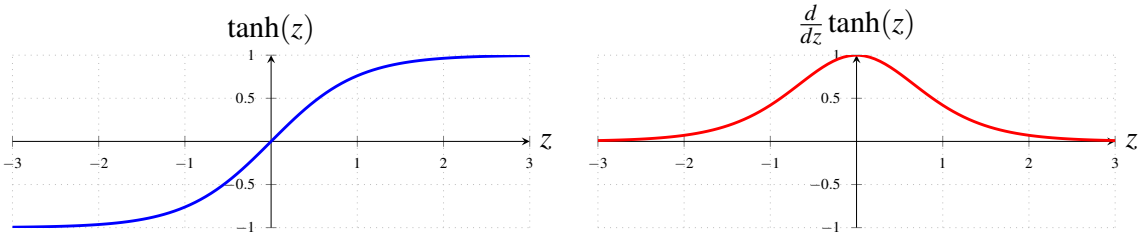
this reason, the activation function h is necessary, and it should not be an affine transformation.

We present some examples of activation functions, restricting ourselves to the most used. The main reference followed for this section is [17, Chapter 6], where a wider battery of examples of activation functions and a classification of this functions can be found.

The first example of activation function we will expose is the **hyperbolic tangent**:

$$h(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}, \qquad z \in \mathbb{R}.$$

This function is odd $\big(\tanh(-z) = -\tanh(z)\big)$, and is a *sigmoid function*, that is (as claimed in [21, Section 1]), a bounded differentiable, real function that is defined for all real input values and has a positive derivative at each point (which makes it monotonically increasing) and exactly one inflection point. This properties are quite desirable in an activation function, for instance, it is necessary to calculate the derivative of the activation function for training, and bounded outputs are usually easier to interpret. Sigmoid functions have a characteristic "S" shape and bell-shaped derivatives.



(a) Hyperbolic tangent.

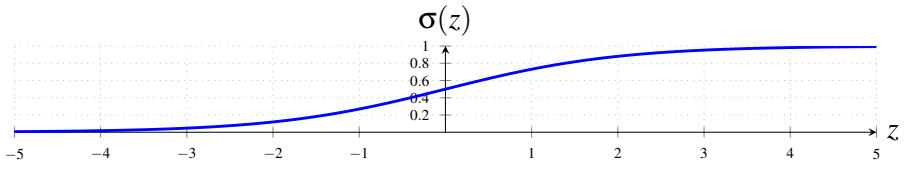
(b) Its derivative, $h'(z) = 1 - \tanh^2(z)$.

Figure 2.10: Graphs of hyperbolic tangent and its derivative.

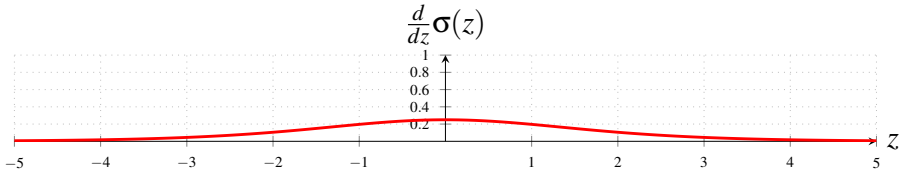
In some cases, it is desirable for the output to lie lies in the interval $[0,1]$, for example, when we want the output to be a probability. This could be achieved by simply composing the hyperbolic tangent with a homothecy and a translation $\big(h(z) = \frac{\tanh(z)+1}{2}\big)$, but there is a popular activation function that fulfills this requirement, the **logistic function**:

$$h(z) = \sigma(z) = \frac{1}{1+e^{-z}}, \qquad z \in \mathbb{R}.$$

The logistic function also happens to be a sigmoid function, and is symmetric in the sense that $\sigma(-z) = 1 - \sigma(z)$ for all $z \in \mathbb{R}$.



(a) Logistic function.



(b) Its derivative, $h'(z) = \sigma(z)(1-\sigma(z)) = \sigma(z)\sigma(-z)$.

Figure 2.11: Graphs of the logistic function and its derivative.

These are the most classic examples of activation functions. There is a close relation between them, since $\tanh(z) = 2\sigma(2z) - 1$ for all $z \in \mathbb{R}$. Besides, the derivative of both these functions can be easily worked out from their values using the equations in Figures 2.10(b) and 2.11(b) derivative, which is useful when it comes to tuning the network parameters during training (see 2.3.2).

On the other hand, both the hyperbolic tangent and the logistic function have a disadvantage. When they are evaluated at points far from 0, they resemble constant functions, equivalently, their derivatives are close to 0. This saturation causes these functions to only be sensitive to changes around their midpoints, which entails an issue during training, when the value of the derivative of the activation function becomes relevant, this is called the *vanishing gradient problem*. See [22].

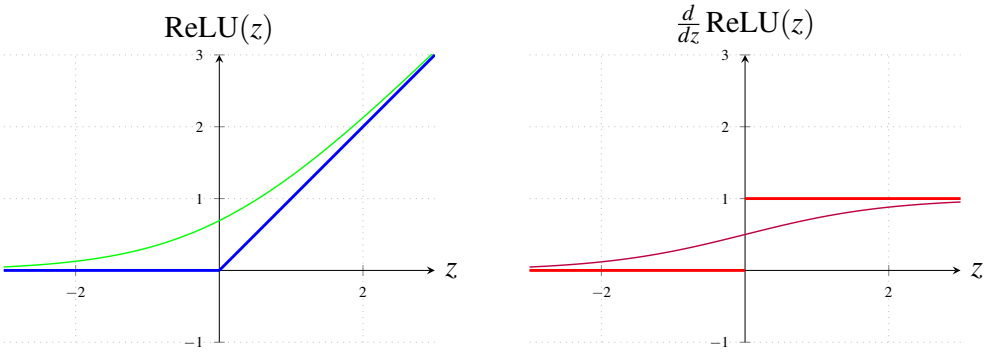
To solve this problem, other activation functions can be used, the **rectified linear activation function** or ReLU (*REctified Linear Units*) is one of the most popular among them. The training speed of networks that use this activation function for their neurons can be significantly higher than that of the networks using sigmoid functions.

$$h(z) = \mathrm{ReLU}(z) = \max(0,z) = \begin{cases} z & \text{if } z \geq 0 \\ 0 & \text{if } z < 0, \end{cases} \qquad z \in \mathbb{R}.$$

Its derivative is not defined at the point $z = 0$, but this is not a concern in practice, due to the negligible probability of the input actually taking that exact value. If fact, we could approximate this function with the **softplus** (also known as **smooth ReLU**) function, $h(z) = \ln(1+e^z)$, which is analytic[1] and its derivative is precisely the logistic function. However, the evaluation of the softplus function is considerably slower. The ReLU function is differentiable in $\mathbb{R}\setminus\{0\}$, and its derivative is easily calculable.

$$h'(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z < 0, \end{cases} \qquad z \in \mathbb{R}.$$

[1] Locally given by a convergent power series. In particular, infinitely differentiable.



(a) Rectified linear function (blue) and softplus function (green).

(b) Derivatives of both the rectified linear function (red) and the softplus function (purple).

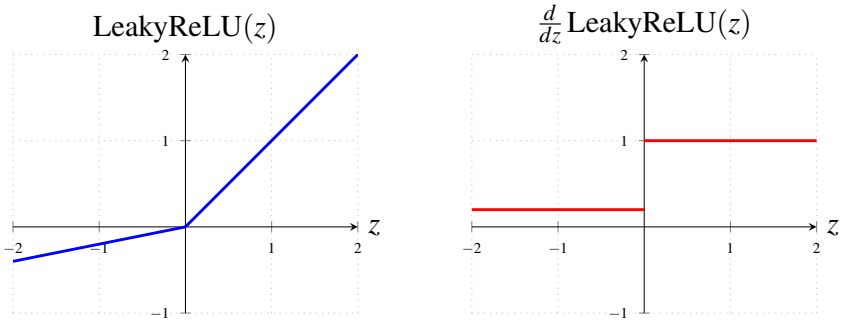
Figure 2.12: Graphs of rectified linear function and the softplus function, along with their derivatives.

This activation function completely deactivates the neuron when the input is negative. According to [23], this can cause some neurons to remain inactive no matter what input is supplied to the net, which is known as the *Dying ReLU problem*. A large number of dead neurons can affect the performance of the network. Our last example of activation function is a variation of the ReLU function that solves this problem, the **leaky ReLU**, where slope is changed for negative inputs.

$$h(z) = \text{LeakyReLU}(z) = \begin{cases} z & \text{if } z \geq 0 \\ \alpha z & \text{if } z < 0, \end{cases} \qquad z \in \mathbb{R}.$$

Where $\alpha \geq 0$, usually $\alpha \in]0,1[$. For $\alpha = 0$, we recover the traditional ReLU. This function has the same differentiation issues as the ReLU, but once again this is not a concern.

$$h'(z) = \begin{cases} 1 & \text{if } z > 0 \\ \alpha & \text{if } z < 0, \end{cases} \qquad z \in \mathbb{R}.$$



(a) Leaky ReLU for $\alpha = 0.2$.

(b) Its derivative.

Figure 2.13: Graphs of leaky ReLU (with $\alpha = 0.2$) and its derivative.

2.2.2 Universal approximation theorem

As we stated before, we intend the network to replicate a function f, that maps a dark photography to what the image would look if it had been taken with suitable illumination. We shall show some properties of neural networks that suggest that this approximation can be achieved.

Consider the function

$$G(\mathbf{x}) = \sum_{i=1}^{k} \alpha_i \sigma\left(\sum_{j=1}^{n} w_{ij} x_j + b_i\right) = \sum_{i=1}^{k} \alpha_i \sigma\left(\mathbf{w}_i^T \mathbf{x} + b_i\right), \tag{2.2.1}$$

where $k \in \mathbb{N}$, $\mathbf{x}^T = (x_1, \ldots, x_n) \in \mathbb{R}^n$ and $\alpha_i, b_i \in \mathbb{R}$, $\mathbf{w}_i^T = (w_{i1}, \ldots, w_{in}) \in \mathbb{R}^n$ for $i = 1, \ldots, k$.

The function G is a neural network with n inputs, one hidden layer with k neurons and a single output neuron. The ith neuron has bias b_i and multiplies the jth input by the weight w_{ij}. The weight a_i models the connection between the ith neuron and the output neuron. All neurons in the hidden layer apply the logistic activation function, while the output neuron applies the identity and has no bias.

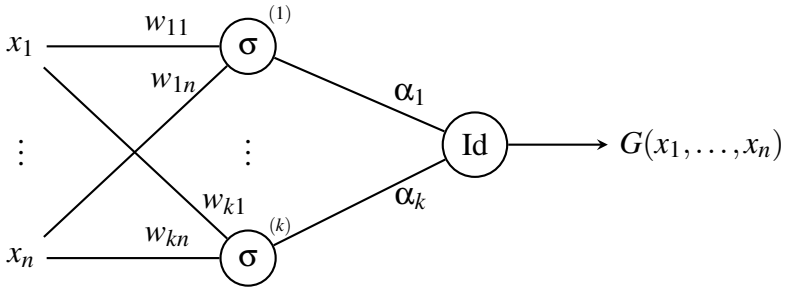


Figure 2.14: Diagram of the function G as a neural network. Biases are omitted.

Now, consider any continuous function $f : \mathbb{R}^n \to \mathbb{R}^m$. We may assume $m = 1$, since approximating f for any norm in $\mathbb{R}^m$ is equivalent to approximating each of its components. The following theorem states that there f can be approximated by neural networks of the form described above.

The symbol $C(I_n)$ stands for the space of real-valued continuous functions on the n-dimensional unit cube, with the supremum norm.

Theorem 2.2.1 (Universal Approximation Theorem) ([3, Theorem 2]) *Let σ be any continuous sigmoidal function. Then, the finite sums of the form described in (2.2.1) are dense in $C(I_n)$.*

In other words, given $f \in C(I_n)$ and $\varepsilon > 0$, there exist $k \in \mathbb{N}$, $\alpha_1, \ldots, \alpha_k, b_1, \ldots, b_k \in \mathbb{R}$ and $\boldsymbol{w}_i \in \mathbb{R}^n$ for $i = 1, \ldots, k$ such that the function G described in (2.2.1) satisfies

$$|G(\boldsymbol{x}) - f(\boldsymbol{x})| < \varepsilon$$

for each $\boldsymbol{x} \in I_n$.

Moreover, the theorem remains valid when replacing I_n with any compact subset of $\mathbb{R}^n$.

The definition of *sigmoidal function* used by G. Cybenko in [3] differs from the definition we provided previously in 2.2.1, that is the reason why continuity of the sigmoidal function is explicitly mentioned. However, all the results of this section are valid for the sigmoidal functions that we presented earlier.

We shall provide an outline of the proof ([24]). The proof requires several tools. The first one is a straightforward consequence of Hahn-Banach Theorem ([25, Theorem 3.2]).

Lemma 2.2.2 ([26, Proposition 7.8]) *Let X be a normed space, and $M \subset X$ a closed subspace. Given a point $x \in X \backslash M$, there exists a linear functional $f \in M^{\perp}$ ($f(M) = \{0\}$) with $\|f\| = 1$, such that $f(x) = d(x, M) > 0$.*

The next result we will need is a particular case of the Representation Theorem for the continuous dual of C_0 ([27, Theorem 6.19]), also known as Riesz-Markov Theorem. The set of finite, signed regular[2] Borel measures on I_n will be denoted by $M(I_n)$.

Lemma 2.2.3 *Let L be a bounded linear functional on $C(I_n)$. Then, there exists a unique $\mu \in M(I_n)$ satisfying*

$$L(h) = \int_{I_n} h(x) d\mu(x) \quad \forall h \in C(I_n).$$

We are one more assumption away from being able to prove the theorem.

Definition 2.2.4 ([3]) *A function σ is **discriminatory** if given a measure $\mu \in M(I_n)$, the identity*

$$\int_{I_n} \sigma(w^T x + b) d\mu(x) = 0$$

for all $w \in \mathbb{R}^n$ and $b \in \mathbb{R}$ implies that $\mu = 0$.

Lemma 2.2.5 ([3, Lemma 1]) *Any bounded, measurable sigmoidal function is discriminatory.*

Proof of Theorem 2.2.1. Let $S \subset C(I_n)$ be the set of all functions of the form

$$x \mapsto \sigma(w^T x + b), \qquad x \in I_n$$

with $w \in \mathbb{R}^n$ and $b \in \mathbb{R}$. We shall denote by Σ the linear span of S, the set of all functions which are the restriction to I_n of functions of the form described in 2.2.1. Our goal is to show that $\overline{\Sigma} = C(I_n)$. By contradiction, assume $\overline{\Sigma} \subsetneq C(I_n)$, that is, $\exists g \in C(I_n) \backslash \overline{\Sigma}$. The application of Lemma 2.2.2 guarantees the existence of a non-zero linear functional L satisfying $S \subset \overline{\Sigma} \subset \ker L$ $\big(L(S) \subset L(\overline{\Sigma}) = \{0\}\big)$.

By Lemma 2.2.3, there exists $\mu \in M(I_n)$ satifying

$$L(h) = \int_{I_n} h(x) d\mu(x) \quad \forall h \in C(I_n).$$

In particular, for each $w \in \mathbb{R}^n$ and $b \in \mathbb{R}$, define $h(x) = \sigma(w^T x + b)$, we have $h \in S$ and hence

$$0 = L(h) = \int_{I_n} \sigma(w^T x + b) d\mu(x).$$

Finally, Lemma 2.2.5 implies that $\mu = 0$, which is impossible, since $L \neq 0$. □

So far, we have shown that any continuous function in a compact space can be approximated by a neural network. A similar approximation can be achieved in the case of measurable functions, which is obtained as a consequence of Theorem 2.2.1 and the following particular case of Lusin's Theorem ([27, Theorem 2.24]). The symbol λ will denote the Lebesgue measure.

[2] A measure is said to be regular if every measurable set can be approximated from above by open measurable sets and from below by compact measurable sets.

Lemma 2.2.6 *Let $f : I_n \to \mathbb{R}$ a Lebesgue-measurable function. Then, for any $\varepsilon > 0$ there exist a set $D \subset I_n$ with $\lambda(I_n \backslash D) < \varepsilon$ and $h \in C(I_n)$ such that*

$$h(x) = f(x), \qquad \forall x \in D.$$

Theorem 2.2.7 ([3, Theorem 2]) *Let σ be any continuous sigmoidal function, and let $f : I_n \to \mathbb{R}$ be a Lebesgue measurable function. Then, for any $\varepsilon > 0$, there exist a function G of the form described in (2.2.1) and a set $D \subset I_n$ with $\lambda(I_n \backslash D) < \varepsilon$ such that*

$$|G(\boldsymbol{x}) - f(\boldsymbol{x})| < \varepsilon$$

for each $\boldsymbol{x} \in D$.

This theorem is also valid when replacing I_n with any compact subset of $\mathbb{R}^n$.

Proof. By Lemma 2.2.6, there exist a set $D \subset I_n$ with $\lambda(I_n \backslash D) < \varepsilon$, and a function $h \in C(I_n)$ such that

$$h(x) = f(x), \qquad \forall x \in D.$$

On the other hand, by Theorem 2.2.1, there exists a function G of the form described in (2.2.1) satisfying

$$|G(x) - h(x)| < \varepsilon$$

for each $x \in I_n$. Therefore,

$$|G(x) - f(x)| < \varepsilon, \qquad \forall x \in D.$$

□

It shall be noted that given a width W and a height H, the space of RGB images $W \times H$ is a compact space. The intensity of a color can be represented by a value in $[0,1]$. Hence, this space would correspond to $I_n = [0,1]^n$ with $n = W \cdot H \cdot 3$ (3 color channels).

Similar results have been stated for neural networks that use other activation functions instead of sigmoids, such as ReLU or LeakyReLU. See [28].

2.3 Training a neural network

We have shown that a wide class of functions can be approximated by neural networks. However, finding a net that provides a good approximation of the target function (which is in fact unknown) is a non-trivial task. The goal of training is to adjust the weights that model the connection between neurons so that the net learns to perform a certain job. For this work, some elements are needed.

First, we need a **dataset** with enough solved instances of the problem we intend the network to learn to solve.

2.3.1 Loss functions

Second, we need a way to measure the correctness of an output for an instance of the problem, this is called a **loss function**, also known as **objective function**, since the goal of the network is to minimize the expected value of the function for all possible inputs. There are objective

functions that are not loss functions. In other cases objective functions are maximized instead of minimized.

There exists a great variety of loss functions we could use. We present some of the most popular. We will denote by $\hat{y}$ the output of the model and y stands for the desired output. Assume that the output is a continuous vector with m components, the ith component of y will be denoted by $y(i)$, and the same for $\hat{y}$.

- The **Mean Squared Error** (MSE or L_2 penalty):

$$MSE(y,\hat{y}) = \frac{1}{m}\sum_{i=1}^{m}(y(i)-\hat{y}(i))^2.$$

  This loss function punishes the big errors severely, and is sensitive to outliers or noisy data.

- The **Mean Absolute Error** (MAE or L_1 penalty):

$$MAE(y,\hat{y}) = \frac{1}{m}\sum_{i=1}^{m}|y(i)-\hat{y}(i)|.$$

  This loss function solves the problems of MSE. Nevertheless, its partial derivative with respect to the variable $\hat{y}_i$ do not exists at the point y_i.

There exist discrete activation functions, like $h(z) = 1$ for $z \in \mathbb{R}^+$ and $h(z) = 0$ for $z \in \mathbb{R}_0^-$, so the output of a neural network can be discrete. In some of such cases (like classification problems), the **0-1 loss function** may be useful.

- $L_{0\text{-}1}(y,\hat{y}) = I(\hat{y} \neq y) = \begin{cases} 0 & \text{if } \hat{y} = y \\ 1 & \text{if } \hat{y} \neq y. \end{cases}$

In case the output is a probability, the **binary cross entropy** is a suitable loss function.

- $H(y,\hat{y}) = -y\ln\hat{y} - (1-y)\ln(1-\hat{y})$.

When the output is non-bounded (for instance, a ReLU or LeakyReLU), this loss function can be combined with the logistic function, obtaining $H(y,\sigma(\hat{y}))$.

2.3.2 Gradient descent

A method to adjust the weights of the network so the loss function is minimized is also needed. **Gradient descent** (GD) is an iterative method for such purpose. We shall follow [4, Section 3.3.2]. Assume that the network depends on q weights $w_1,\ldots,w_q$, which have already been initialized (to a constant value or randomly). Let $L(y,\hat{y})$ be a differentiable (at least almost everywhere) lost function. The goal of the training is to minimize the function

$$E_{\text{in}}(w_1,\ldots,w_q) = \frac{1}{N}\sum_{i=1}^{N}L(y_i,\hat{y}_i), \tag{2.3.1}$$

where $(\mathbf{x}_i, y_i)$ is an example of the dataset and $\hat{y}_i$ the output of the net for the input $\mathbf{x}_i$. Since each $\hat{y}_i$ depends on the weights, E_{in} is a function of them.

It is known that ∇E_{in} is the direction in which the function has the greatest slope. Hence, $-\nabla E_{\text{in}}$ is the direction in which E_{in} decreases faster. If we update the weights by the following rule for

a suitable value of $\eta > 0$ for $j = 1, \ldots, q$:

$$w'_j := w_j - \eta \frac{\partial E_{\text{in}}}{\partial w_j}(w_1, \ldots, w_q)$$

we should get

$$E_{\text{in}}(w'_1, \ldots, w'_q) \leq E_{\text{in}}(w_1, \ldots, w_q).$$

This method, applied iteratively, converges (under certain conditions) to a local minimum of the function E_{in}, which would be a fixed point for the method because the gradient would be zero. In the case L (and hence E_{in}) is a convex function, such as *MSE* or *MAE*, the local minimum would be a global one.

The convergence of the gradient descent method depends on a good choice for the value of η, known as *learning rate*, which could remain constant throughout the process or change with each iteration.

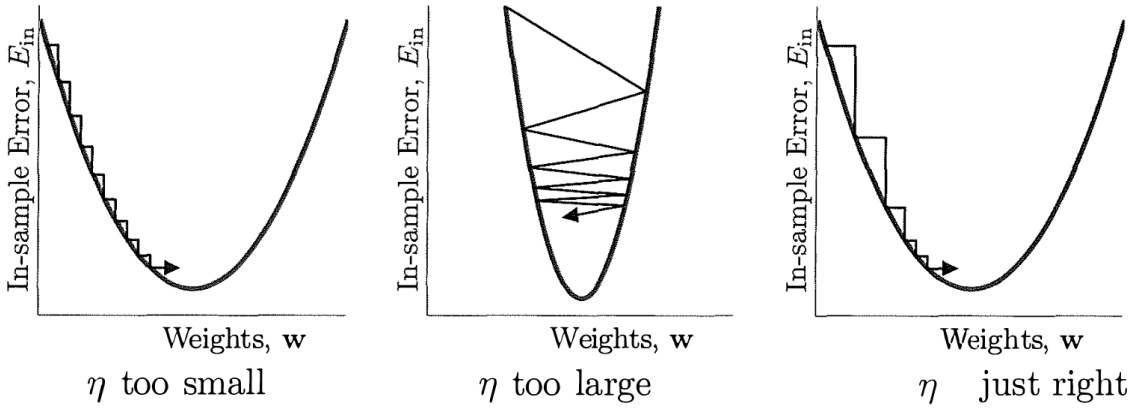


Figure 2.15: Influence of learning rate on gradient descent.

- A small learning rate requires many updates to converge.
- A large learning rate leads to drastic updates that may cause the algorithm to bounce around the minimum or even diverge.
- A suitable learning rate swiftly reaches the minimum point.

Source: [4, Section 3.3.2] (image), https://www.jeremyjordan.me/nn-learning-rate (caption).

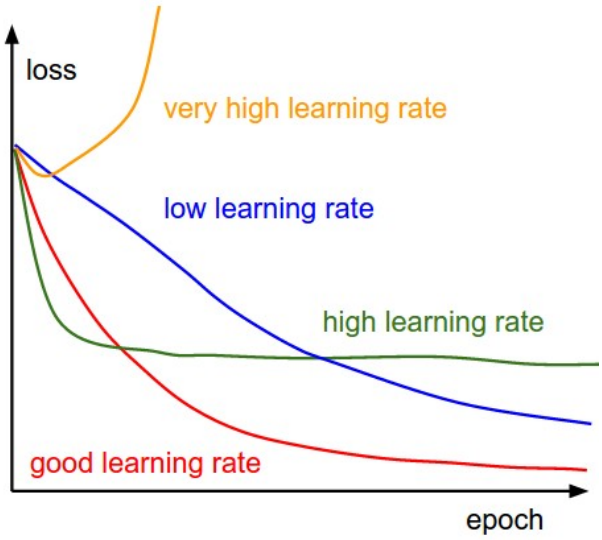


Figure 2.16: Evolution of the value of the in-sample error (loss function) throughout the iterations (epochs) for different learning rates.

- A very high learning rate may cause the algorithm to diverge.
- A low learning rate takes many epochs to converge.
- A high learning rate quickly gets close to the minimum at first, but then oscillates around it.
- A good learning rate rapidly converges to the minimum.

Source: https://cs231n.github.io/neural-networks-3.

There exist several criteria to decide when to stop the training:

- The in-sample error drops below a certain threshold.
- The gradient (in norm) is almost zero.
- A certain number of epochs have been completed.
- The in-sample error does not change (significantly) within a certain number of epochs.

The method we just described computes the gradient over the whole dataset for each iteration, this method is known as **batch gradient descent**, and is prone to getting stuck in the nearest local minimum it finds. Instead of the whole dataset, we could select one example at random, $(\mathbf{x}_i, y_i)$, for each iteration, compute the output $\hat{y}_i$ for the input $\mathbf{x}_i$ and adjust the weights of the network according to

$$\nabla L(y_i, \hat{y}_i) = \left(\frac{\partial L}{\partial w_1}(y_i, \hat{y}_i), \ldots, \frac{\partial L}{\partial w_q}(y_i, \hat{y}_i) \right),$$

this is called **online gradient descent**. Another possibility would be to select a random small subset (with $K << N$ elements) of the dataset for each iteration, and compute the gradient of a function identical to (2.3.1) but considering only the K selected examples instead of all N instances. This variation is called **mini-batch gradient descent**, and the parameter K is known as the *batch size*. Both online gradient descent and mini-batch gradient descent use a random sample to estimate the value of the gradient instead of the whole dataset, this is known as **stochastic gradient descent** (SGD), and usually performs better than batch gradient descent.

There are several techniques (*optimization algorithms*) to decide a suitable way to modify the

weights in each iteration based on the value of the gradient. Among them, we will rely on **Adam** (*adaptive moment estimation*) for the training of our model.

The Adam optimization algorithm

Adam is one of the most popular and effective algorithms to adapt the weights of the network based on the value of the gradient. The main reference for this section is [29], but we will also follow [17, Chapter 11], [30] and [31]. Adam combines various techniques from other optimization algorithms, which we shall describe first.

Let $E(\mathbf{w})$, with $\mathbf{w} \in \mathbb{R}^q$, be the function to minimize. The iteration (epoch) will be denoted by t, and $\mathbf{w}_t$ stands for the weight vector in the epoch t.

The gradient descent with **momentum** uses the information of past iterations instead of only the value of the gradient in the current epoch. This produces smooth updates, making the method more stable and faster. Its update rule is:

$$\begin{aligned} v_t &= \mu v_{t-1} - \eta \nabla E(\mathbf{w}_{t-1}) \\ \mathbf{w}_t &= \mathbf{w}_{t-1} + v_t, \end{aligned}$$

where $v_0 = 0$. The parameter $\eta > 0$ is the learning rate, $\mu \in [0,1[$ is the momentum, and v_t could be considered as the velocity at which the weights are updated. The lower the momentum, the faster the values of the gradient in past epochs are forgotten, for $\mu = 0$, we recover classic gradient descent.

The *adaptive gradient* strategy (**AdaGrad**) adjusts the learning rate individually for each component of the gradient. For each weight, a learning rate inversely proportional to the square root of the sum of the squares of the partial derivatives with respect to that weight over the epochs is used. This way, the weights that are updated frequently (the partial derivative is far from zero) do so smoothly, while the weights that are hardly updated (the partial derivative is usually close to zero) take bigger steps. The update rule of AdaGrad is:

$$\begin{aligned} r_t &= r_{t-1} + \nabla E(\mathbf{w}_{t-1})^2 = r_{t-1} + \nabla E(\mathbf{w}_{t-1}) \odot \nabla E(\mathbf{w}_{t-1}) \\ \mathbf{w}_t &= \mathbf{w}_{t-1} - \frac{\eta}{\sqrt{r_t} + \varepsilon} \odot \nabla E(\mathbf{w}_{t-1}), \end{aligned}$$

where $r_0 = 0 \in \mathbb{R}^q$. The square and the square root are element-wise operations, and the operator $\odot$ refers to element-wise vector product. The parameter $\varepsilon > 0$ is used to avoid division by zero.

AdaGrad causes an aggressive reduction in the learning rates. There is a variation, **AdaDelta**, that solves this problem by considering only the squared gradients of a fixed number of previous iterations. Instead of storing several squared gradients, a decaying average is applied. Its update rule is:

$$\begin{aligned} r_t &= \gamma r_{t-1} + (1-\gamma) \nabla E(\mathbf{w}_{t-1})^2 = \gamma r_{t-1} + (1-\gamma) \nabla E(\mathbf{w}_{t-1}) \odot \nabla E(\mathbf{w}_{t-1}) \\ \mathbf{w}_t &= \mathbf{w}_{t-1} - \frac{\eta}{\sqrt{r_t} + \varepsilon} \odot \nabla E(\mathbf{w}_{t-1}), \end{aligned}$$

where $r_0 = 0 \in \mathbb{R}^q$. The parameter $\gamma \in [0,1[$ modulates the importance of recent gradients. The lower its value, the faster the squared gradients in past epochs are forgotten.

Adam keeps exponentially decaying averages of past gradients, similar to momentums, and

past squared (element-wise) gradients, similar to AdaDelta.

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla E(\mathbf{w}_{t-1})$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\nabla E(\mathbf{w}_{t-1})$$

where $m_0 = v_0 = 0 \in \mathbb{R}^q$. The values m_t and v_t are estimations of the first (average) and second (uncentered variance) moments of $\nabla E(\mathbf{w}_{t-1})$ respectively. The parameters $\beta_1, \beta_2 \in [0,1[$ are usually set close to 1.

The estimations m_t and v_t are both biased towards zero (see [29, Section 3] and (2.3.3)) due to the initialization. That is, (denoting expectation by $\mathbb{E}$)

$$\mathbb{E}[m_t] \approx (1-\beta_1^t)\mathbb{E}[\nabla E(\mathbf{w}_{t-1})] \quad \text{and} \quad \mathbb{E}[v_t] \approx (1-\beta_2^t)\mathbb{E}[\nabla E(\mathbf{w}_{t-1})^2], \tag{2.3.2}$$

so a bias-correction is applied to obtain unbiased estimations:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \qquad \hat{v}_t = \frac{v_t}{1-\beta_2^t}.$$

Finally, the weights are updated similarly as in AdaDelta:

$$\mathbf{w}_t = \mathbf{w}_{t-1} - \frac{\eta}{\sqrt{\hat{v}_t} + \varepsilon} \odot \hat{m}_t.$$

The Adam algorithm has two important properties. Assuming the ideal cases $\varepsilon = 0$, let

$$\Delta_t = \mathbf{w}_t - \mathbf{w}_{t-1} = -\frac{\eta}{\sqrt{\hat{v}_t}} \odot \hat{m}_t.$$

- **Bounded step size:** For all t,

$$\|\Delta_t\|_\infty \leq \max\left\{1, \frac{1-\beta_1}{\sqrt{1-\beta_2}}\right\}\alpha.$$

- **Scale-invariance:** Let $\lambda > 0$, if we apply Adam to minimize λE instead of E, we get λm_t and $\lambda^2 v_t$ instead of m_t and v_t respectively. Therefore, Δ_t does not change.

Finally, we shall show why the estimations m_t and v_t are unbiased, see (2.3.2). For $t > 0$, we have

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla E(\mathbf{w}_{t-1}) = \sum_{i=1}^{t}(1-\beta_1)\beta_1^{t-i}\nabla E(\mathbf{w}_{i-1}). \tag{2.3.3}$$

On the other hand,

$$\sum_{i=1}^{t}(1-\beta_1)\beta_1^{t-i} = \sum_{i=1}^{t}\beta_1^{t-i} - \beta_1^{t-i+1} = 1-\beta_1^t.$$

To obtain an unbiased average, the weights (of the average, not $\mathbf{w}$) must add up to 1, so we need to divide by $1-\beta_1^t$. The same reasoning is valid for v_t, β_2 and $\nabla E(\mathbf{w}_{t-1})^2$.

2.3.3 The backpropagation algorithm

An optimization algorithm based on the value of the gradient requires a method that computes the partial derivatives of the function to be optimized. The *backpropagation* algorithm allows

us to calculate the partial derivatives of the in-sample error with respect to each of the weights. The first application of this technique to neural networks can be found in [32], although we will follow the references in [4, Section 7.2.3]. The method consist in several applications of the chain rule to compute the partial derivatives with respect to the weights, it proceeds in a recursive approach, starting from the weights of the neurons in the output layer and moving backwards throughout the hidden layers, hence the name of the algorithm. While our reference provides an algorithmic approach of this technique intended for implementation, we shall limit ourselves to show how the partial derivatives can be worked out via this method.

We shall use the same notation as before, take $K \leq N$ training examples $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_K, y_K)$, it could be just one example (SGD) or the whole dataset (batch GD). Let $\hat{y}_i$ denote the output of the network for the input $\mathbf{x}_i$. Our goal is to compute, for each weight w,

$$\frac{\partial E}{\partial w}(w_1, \ldots, w_q) = \frac{\partial}{\partial w} \sum_{i=1}^{K} L(y_i, \hat{y}_i) = \sum_{i=1}^{K} \frac{\partial L}{\partial w}(y_i, \hat{y}_i),$$

where L is a differentiable (at least almost everywhere) loss function. Clearly, it suffices to work out $\dfrac{\partial L}{\partial w}(y_i, \hat{y}_i)$ for each training example $(\mathbf{x}_i, y_i)$. Take a training example $(\mathbf{x}, y)$, for which the net outputs $\hat{y}$. We also need the output of every neuron in the network. Since only $\hat{y} = \big(\hat{y}(1), \ldots, \hat{y}(m)\big)$ depends on the weights, we have by the chain rule

$$\frac{\partial L}{\partial w}(y, \hat{y}) = \sum_{i=1}^{m} \frac{\partial L}{\partial \hat{y}(i)}(y, \hat{y}) \frac{\partial \hat{y}(i)}{\partial w}(w_1, \ldots, w_q),$$

so it suffices to compute the partial derivative of each component $\hat{y}(i)$ with respect to the weights. To simplify the notation, let o be a component of $\hat{y}$, we need to find $\dfrac{\partial o}{\partial w}$ for each weight w. To shorten the explanation, we shall say that a weight w can be updated when the partial derivative of o with respect to w has been found, we shall also say that a weight belongs to a layer if it is a parameter of a neuron in that layer.

The neuron in the output layer corresponding to o (we will denote a neuron and its output value with the same symbol) must have an activation function h_o, that we will assume differentiable. Suppose that there are k neurons in the preceding layer (last hidden layer), whose outputs are $\alpha_1, \ldots, \alpha_k$. Then, $o = h_o(w_{o1}\alpha_1 + \cdots + w_{ok}\alpha_k + w_{o0})$, where $w_{o0}, \ldots, w_{ok}$ are the weights of the neuron o. Therefore,

$$\frac{\partial o}{\partial w_{oj}} = h_o'(w_{o1}\alpha_1 + \cdots + w_{ok}\alpha_k + w_{o0})\alpha_j, \tag{2.3.4}$$

where $\alpha_0 = 1$. Clearly, if w is a parameter of the output neuron o and u is a different output neuron, $\dfrac{\partial u}{\partial w} = 0$. It is also clear that the value an output neuron does not affect other output neurons. Hence, for two output neurons u, o we have

$$\frac{do}{du} = \begin{cases} 1 & \text{if } u = o \\ 0 & \text{if } u \neq o. \end{cases}$$

So far, we have shown how to compute the partial derivatives of each output (and therefore those of the in-sample error) with respect to the weights in the output layer, it remains to do the same for the weights in hidden layers. Let R be the number of layers of the network, $R-1$ hidden layers $l_1, \ldots, l_{R-1}$ and the output layer l_R, whose weights can already be updated.

Fix an output neuron o, assume that the weights in layers $l_{i+1},\dots,l_R$ can be updated, and that $\dfrac{do}{du}$ is known for each neuron u in layers $l_{i+1},\dots,l_R$. Let v be a neuron in layer l_i and let w be a weight of v. By the chain rule

$$\frac{\partial o}{\partial w} = \frac{do}{dv}\frac{\partial v}{\partial w}. \tag{2.3.5}$$

The value of $\dfrac{\partial v}{\partial w}$ can be calculated just like in (2.3.4). The neuron v must have an activation function h_v, that we will assume differentiable. Suppose that the layer l_{i-1} has $\hat{k}$ neurons, whose outputs are $\beta_1,\dots,\beta_{\hat{k}}$. Then, $v = h_v(w_{v1}\beta_1 + \cdots + w_{v\hat{k}}\beta_{\hat{k}} + w_{v0})$, where $w_{v0},\dots,w_{v\hat{k}}$ are the weights of the neuron v. Assume that $w = w_{vj}$ for some $j = 0,\dots,\hat{k}$, then

$$\frac{\partial v}{\partial w} = \frac{\partial v}{\partial w_{vj}} = h_v'(w_{v1}\beta_1 + \cdots + w_{v\hat{k}}\beta_{\hat{k}} + w_{v0})\beta_j,$$

where $\beta_0 = 1$. We have found the second factor of (2.3.5).

Now, let $\tilde{k}$ be the number of neurons in the layer l_{i+1} (may be different as before), we shall denote the neurons in such layer by $u_1,\dots,u_{\tilde{k}}$. Assume that there are p neurons in the layer l_i, $v_1,\dots,v,\dots,v_p$. Let w_{u_jv} be the weight that models the connection between v and u_j for $j = 1,\dots,\tilde{k}$, that is,

$$u_j = h_{u_j}(w_{u_j0} + w_{u_jv_1}v_1 + \cdots + w_{u_jv}v + \cdots + w_{u_jv_p}v_p).$$

Therefore, we have

$$\frac{du_j}{dv} = h_{u_j}'(w_{u_j0} + w_{u_jv_1}v_1 + \cdots + w_{u_jv}v + \cdots + w_{u_jv_p}v_p)w_{u_jv}.$$

Finally, since v only affects o through the neurons in the layer l_{i+1} ($u_1,\dots,u_{\tilde{k}}$), we have

$$\frac{do}{dv} = \sum_{j=1}^{\tilde{k}} \frac{do}{du_j}\frac{du_j}{dv}. \tag{2.3.6}$$

The value of $\dfrac{do}{du_j}$ is known since u_j belongs to the layer l_{i+1}, the first factor of (2.3.5) is calculated.

There are cases where a neuron v in layer l_i may not be connected with all the neurons in layer l_{i+1}, or it may be connected with some neurons in subsequent layers $l_{i+2},\dots,l_R$ (*skip connections*). In such cases, all neurons directly affected by v (neurons that take the output of v as input) must appear in (2.3.6). The backpropagation method works as long as the network is a *feedforward neural network*, that is, the connections between its neurons do not form a cycle.

2.3.4 Overfitting

We shall describe a common problem that may occur during training, the *overfitting* problem. This problem causes the model to perform extremely well over the training samples (the in-sample error is almost zero) but poorly over examples that the model has not seen during training. The main reference followed for this section are [17, Chapter 10] and [4, Chapter 4].

A deep neural network can rely on millions of parameters, which makes the hypothesis set $\mathcal{H}$ huge. If the number of training samples is not enough to discard all hypothesis that do not solve our machine learning problem, we probably will end up with a model capable of solving all the training examples, but which fails when it finds new examples. Intuitively, we could say that the network has not detected features and patterns in the training data in order to solve the examples, it has just memorized how to solve all training examples. The model has been fitted to the training data more than necessary.

An approach to avoid this problem could be limiting the number of parameters of the network and, therefore, the hypothesis set. If the hypothesis set is simple enough that it is not possible to "memorize" all the training samples, the only chance to obtain a model capable of solving them would be to successfully detect features and patterns in training data. Which could be translated into a better performance outside the training examples.

On the other hand, if we select a hypothesis set that is too small, we will never be able to find a model that approximates the target function satisfactorily, even within the training data. This is called the *bias-variance tradeoff* (see [4, Section 2.3]).

One way to reduce the number of parameters of the network is to suppress some hidden layers or neurons in hidden layers. Anther approach would be suppressing some of the connections between neurons, so each neuron in the net does not receive an input for each neuron in the previous layer, and does not send its output to every neuron in the next layer. Neural networks where each neuron is connected with all the neurons in the previous and next layers are known as *fully connected neural networks*.

Alternatively, we could impose a constraint to the parameters instead of reducing its number. Generally, a restriction ensuring that the weights do not get too large. To achieve this, we can add a new term to the function to minimize, called the *penalty term* or *regularization term*:

$$E_{in}(\mathbf{w}) + \lambda \|\mathbf{w}\|_2,$$

where $\lambda > 0$ determines the importance given to the restriction. The larger its value, the smaller weights we will get.

There exist other techniques to prevent a neural network from being overfitted without limiting the number of parameters, which are applied during training. We shall highlight two of them among the most used.

The first one is **dropout**, where the outputs of hidden neurons are set to zero with a certain probability each training epoch. *Dropout layers* set to zero a random set of outputs of the preceding layer during training.

The other technique is **early-stopping**, where a small fraction (about 10%) of the training examples are set aside to calculate a *validation error* after each epoch. The training stops where the validation error stops decreasing, even if the in-sample error keeps doing so.

Any effort to combat overfitting that constrains the hypothesis set or the learning algorithm is called a *regularization*.

2.3.5 Batch normalization

Another concern during training of a deep neural network is ensuring that the weights of the network remain within a reasonable range of values. If weights become to large, the network may suffer from the *exploding gradient problem*. As errors are propagated backward through

the network, the gradient in the earlier layers can sometimes grow exponentially large, causing wild fluctuations in the weight values and preventing them from converging.

More information about this problem and the solution we will provide next can be found in [5, Chapter 2].

To solve this problem, we can normalize the inputs of the neural network. For instance, if the inputs are images, the pixel values could be scaled between -1 and 1. However, this only solves the problem during the first few epochs of training. **Batch normalization** is an effective solution for this problem, we only have to normalize the input of each layer by adding a *batch normalization layer* after every hidden layer.

Assume that we have a layer with m outputs and a minibatch-size of K. Then, for all the examples of a mini-batch together, the layer outputs mK values. The batch normalization layer that goes after applies the following transformation to the values. Let $x_1, \ldots, x_{mK}$ be the outputs of the layer.

First, the values are standardized so they have zero mean and unit variance.

$$\mu = \frac{1}{mK}\sum_{i=1}^{mK} x_i, \qquad \sigma^2 = \frac{1}{mK}\sum_{i=1}^{mK}(x_i-\mu)^2,$$

$$\hat{x}_i = \frac{x_i-\mu}{\sqrt{\sigma^2+\varepsilon}} \qquad \forall i = 1,\ldots,mK,$$

where $\varepsilon > 0$ avoids division by 0. Then, they are scaled and shifted by two trainable parameters of the layer, γ and β.

$$y_i = \gamma\hat{x}_i + \beta \qquad \forall i = 1,\ldots,mK.$$

The values $y_1, \ldots, y_{mK}$ are the inputs for the next layer.

When making predictions, there is no batch over which to take averages. Therefore, the batch normalization layer calculates a moving average of the mean and standard deviation of the values during training, and those values are used to normalize the data at test time.

2.4 Convolutional neural networks

2.4.1 Convolutional layers

Imagine that a fully connected neural network receives a 256×256 RGB image, the input layer would have $256 \cdot 256 \cdot 3 = 196608$ neurons, and each neuron in the first hidden layer would have that many weights. A network with a high number of weights may be really slow to train. Besides, it has the risk of overfitting. Therefore, if we used a fully connected NN for out purpose, the net would have to be extremely simple to have a not too high number of parameters.

This is where *convolutional neural networks* (CNN) come into play. We shall follow the references in [17, Chapter 13]. CNN are regularized neural networks where all neurons in a layer are not necessarily connected with all neurons in the next layer. Instead, there are layers (*convolutional layers*) where only a certain group of neurons that are "close to each other" affects a neuron in the next layer. Moreover, the same weights are applied for each group of neurons. This may be difficult to describe, but is easily understood with an example.

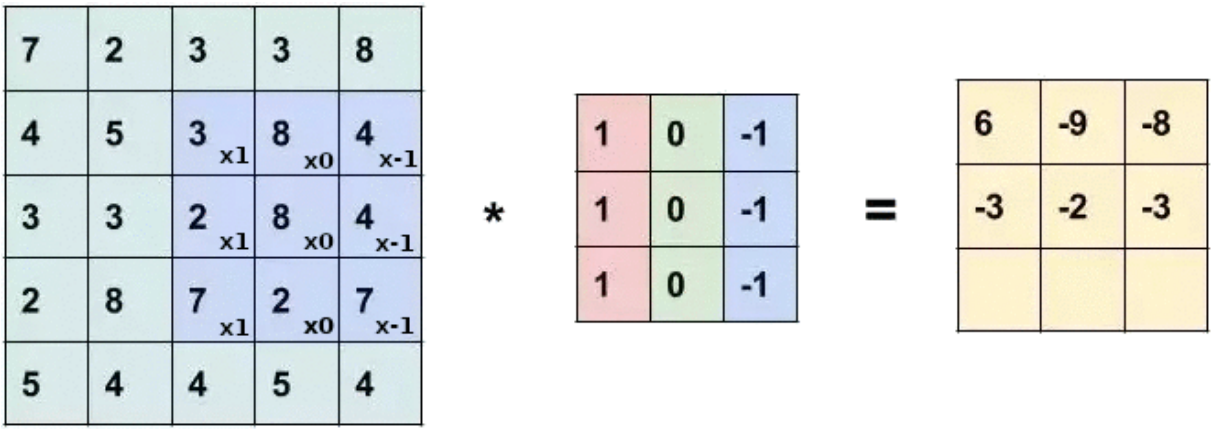


Figure 2.17: We can see the layer l_i in green at left, the layer l_{i+1} in yellow at right and the weights in the middle. The middle-right neuron of the layer l_{i+1} receives inputs from only 9 neurons (those in the 3×3 blue square) in the layer l_{i+1}, the inputs are multiplied by the weights and added up obtaining the value $-3 = 1 \cdot 3 + 0 \cdot 8 - 1 \cdot 4 + 1 \cdot 2 + 0 \cdot 8 - 1 \cdot 4 + 1 \cdot 7 + 0 \cdot 2 - 1 \cdot 7$. The rest of the values in the yellow square can be calculated by shifting the blue square and repeating the process. After that, an activation function is applied to each one. Although they do not appear in this example, biases can also be used.
Source: https://learnopencv.com/image-classification-using-convolutional-neural-networks-in-keras.

In a fully connected network, there would be needed $25 \cdot 9 = 225$ weights ($26 \cdot 9 = 234$ including biases) to model all the connections between the neurons within a layer with 25 neurons and the neurons in a layer with 9 neurons. The connection in Figure 2.17 depends on 9 weights (10 with the bias), since each neuron in layer l_{i+1} only depends on 9 neuron of the layer l_i and of all the 9 neurons in the layer l_{i+1} share the same weights, which is called *weight sharing*.

In order to train the model via backpropagation when there are restriction on the weights, the following considerations must be made. Assume we want to satisfy the restriction $w_1 = \cdots = w_r$. Then,

- The weights must be initialized to the same value.
- The weights must be updated in the same way. We shall use $\sum_{j=1}^{r} \frac{\partial E}{\partial w_j}$ to update each of the weights.

This strategy allows us to create neural networks with the same number of weights, but that depend on much less parameters. Therefore, we have to learn fewer parameters during training, and we work with a much smaller hypothesis set.

In Figure 2.17, the operation that is applied to the values in the green square and the weights resulting in the values in the yellow square is called *convolution*, and denoted by the symbol $*$.

The convolution is applied between matrices. The first matrix is the input and the second one (weights) is called a *filter* or *kernel*. The filter is shifted along the input matrix, and for each shift the multiplications and the sum described in Figure 2.17 are carried out, obtaining a value of the output matrix. The *stride* determines the number of positions the filter moves in each shift, it may be different for width and height.

Clearly, the shape of the output matrix depend on the dimensions of the input matrix and the

hyperparameters[3] stride and kernel shape. If we want to modify the shape of the output, we can add a border to the input so its dimensions are modified.

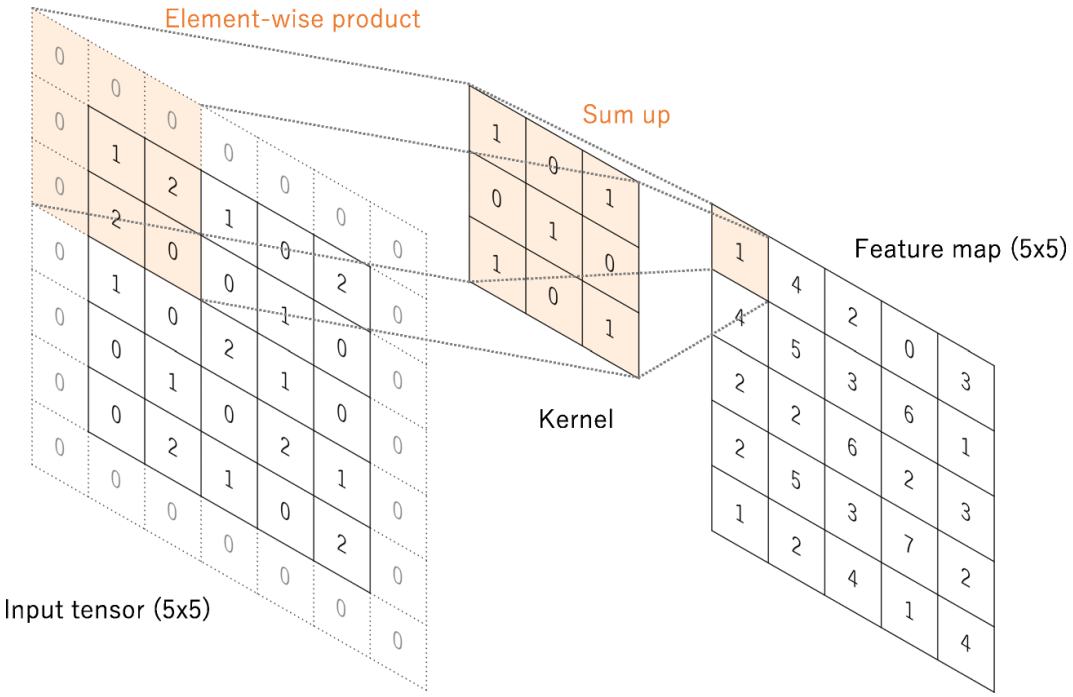


Figure 2.18: ([33, Figure 4]) In this example, a border of zeros is added to the input matrix (sometimes called tensor) so its shape (5×5) is kept in the output matrix (also called feature map).

This technique is known as *zero padding*. Another option would be to extend the border by replicating the closest row or column until the image reaches the desired shape.

Assume that the shape of the input matrix are $m_h \times m_w$ and the kernel size is $k_h \times k_w$, with $k_h \leq k_h$ and $q_w \leq m_w$. If the stride is (s_h, s_w) and the amount of zero padding on the border is (p_h, p_w), the dimensions of the output matrix are:

$$\text{height} = \frac{m_h - k_h + 2p_h}{s_h} + 1 \quad \text{and} \quad \text{width} = \frac{m_w - k_w + 2p_w}{s_w} + 1, \tag{2.4.1}$$

where the padding must be adjusted so those values are integers.

In case the input has several *channels* (a depth dimension, like RGB images), a kernel with more dimensions can be used. The number of channels is a hyperparameter of the layer.

[3] Parameters that are not derived via training, like the learning rate or minibatch size.

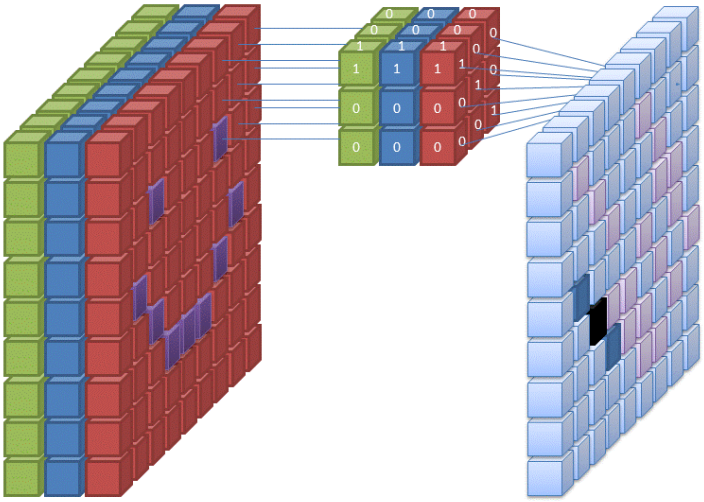


Figure 2.19: In this example, the input has 3 channels, so a kernel with depth 3 is used. The kernel has $3 \cdot 3 \cdot 3 + 1 = 28$ weights including bias. The convolution is performed in the same way as before, the weights in green (respectively blue and red) multiply the values of the input in green (respectively blue and red), then they are all summed up plus the bias to obtain a value of the output. Finally, the activation function is applied to each value.
Source: https://commons.wikimedia.org/wiki/File:Convolutional_Neural_Network_with_Color_Image_Filter.gif.

We have shown a technique to reduce the number of parameters of a neural network drastically. Since we are also capable of processing inputs with several channels, there is the possibility of incorporating several filters in the same layer. Each filter acts independently and the outputs are concatenated, the number of filters of a layer will be the number of channels of its output. Ideally, each filter detects a feature of the input, which is why output channels are also called *feature maps*.

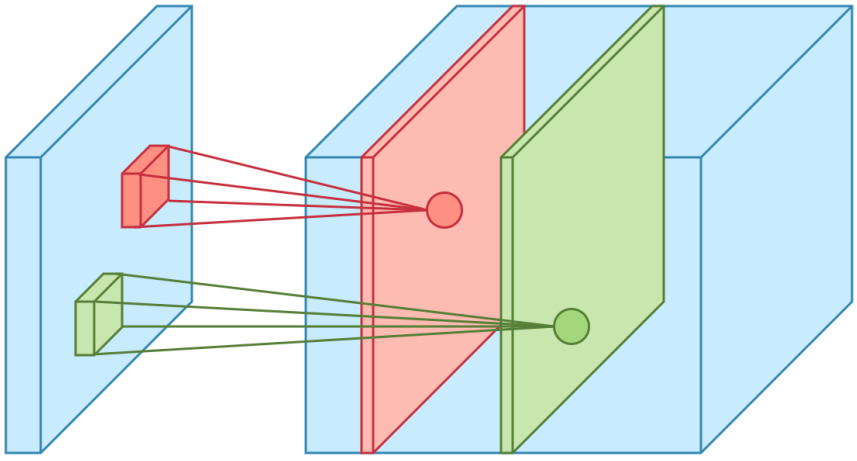


Figure 2.20: Two different filters output two different feature maps.
Source: https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2.

So far, we have introduced convolutional layers, that allow us to reduce the number of parameters of the net significantly, and we have shown that multiple filters can be used in each layer. However, we have provided no reason why convolutional networks should perform better than fully connected ones. The key is **local correlation**. In an image, two pixel that are close to each other are more likely to be correlated than two pixels that are far apart in the image, which could be totally independent. When the inputs have this property, convolutional networks become way more effective than fully connected networks. Since the suppressed connections are those between unlikely correlated neurons, the network capacity to learn how to solve the problem should not be affected. In other words, we preserve the connections that are likely to transfer useful information between neurons. On the other hand, thanks to this reduction in the number of parameters, we prevent overfitting and we may be able to afford to add more filters and layers in order to detect patterns in data more effectively. In addition, convolutional neural networks detect features locally, so they are robust against transformations in data such as translations.

We have presented two-dimensional convolution with the example of images, but there are other types of data that have the property of local correlation. For instance, text is a one-dimensional input with local correlation, so is video, with three dimensions if it has no sound (height, width and time). Audio tracks are another example of data with local correlation.

3D convolution should not be confused with multi-channel 2D convolution (Figure 2.19). In multi-channel 2D convolution the filter must have the same depth as the number of channels of the input, while in 3D convolution, the filter (generally) shifts in the third dimension as in the other two. This occurs analogously for higher dimensions.

In addition to feature extraction, convolutions can be used to apply different effects to an image by setting a fixed kernel (instead of learning a kernel). We shall show some examples of this type of convolutions. For each example, we will apply the same convolution filter to the three channels of an RGB image. Each filter achieves a different effect on the image. The examples are produced with the Convolutional Matrix filter of GIMP ([34, Chapter 16, Section 8.2]). Border replication is applied to preserve the width and height of the images. All matrices (filters) are normalized so the weights add up to 1, except when the weights add up to 0, in which case an offset is applied to the pixel values.

0	0	0
0	1	0
0	0	0

Figure 2.21: Identity filter.
Source: https://commons.wikimedia.org/wiki/File:800_Houston_St_Manhattan_KS.jpg.

1	1	1
1	1	1
1	1	1

Figure 2.22: Blur.

1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

Figure 2.23: Blur.

0	0	0
1	1	1
0	0	0

Figure 2.24: Horizontal blur.

0	1	0
0	1	0
0	1	0

Figure 2.25: Vertical blur.

0	1	0
1	-4	1
0	1	0

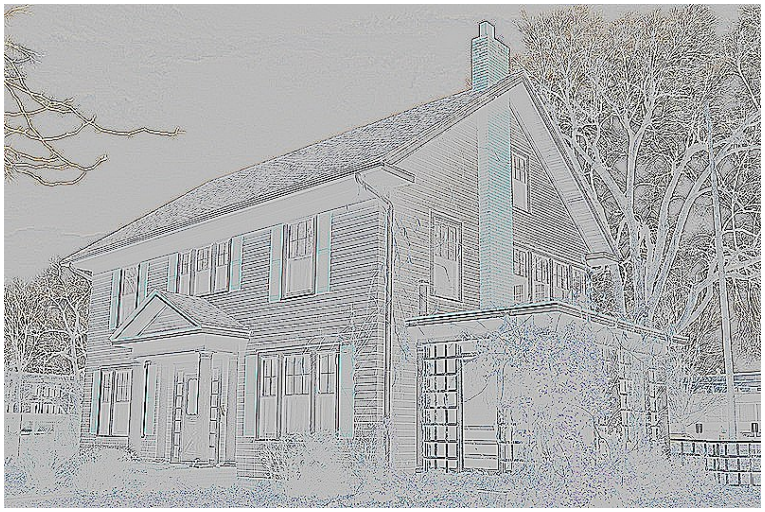


Figure 2.26: Edge detection.

-1	-1	-1
-1	8	-1
-1	-1	-1

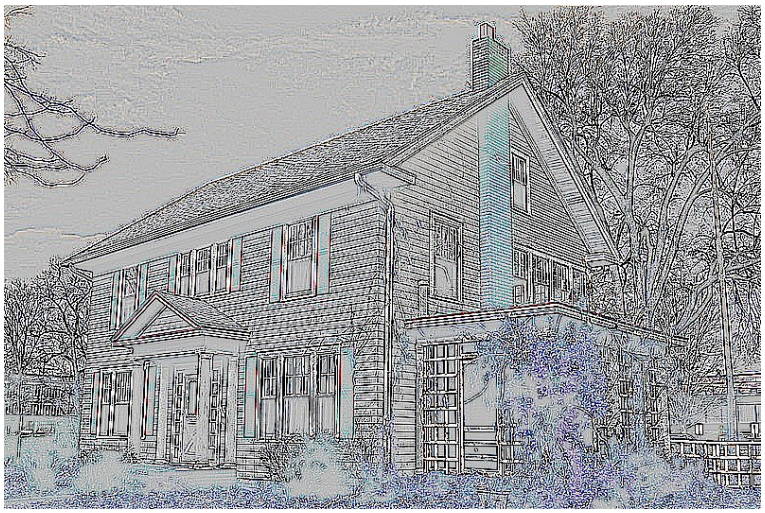


Figure 2.27: Edge detection.

-2	-1	0
-1	1	1
0	1	2

Figure 2.28: Embossing.

-2	-1	0
-1	0	1
0	1	2

Figure 2.29: Embossing.

2.4.2 Pooling layers

Another operation commonly used in convolutional neural networks is *pooling*. Pooling layers provide a robust approach to reduce the dimensionality of the inputs of the following layers, which is known as *downsampling*. Generally, pooling layers lack trainable parameters. Instead, they apply an operation (such as maximum or average) to each rectangular block (in the two-dimensional case) of the input. Then, the kernel shifts and the same operation is applied to another rectangular block, which may overlap with the previous one.

Just as convolution, pooling also depends on hyperparameters like the stride and the kernel size. Furthermore, pooling can be applied to multidimensional inputs, and works well when there exists local correlation in the data.

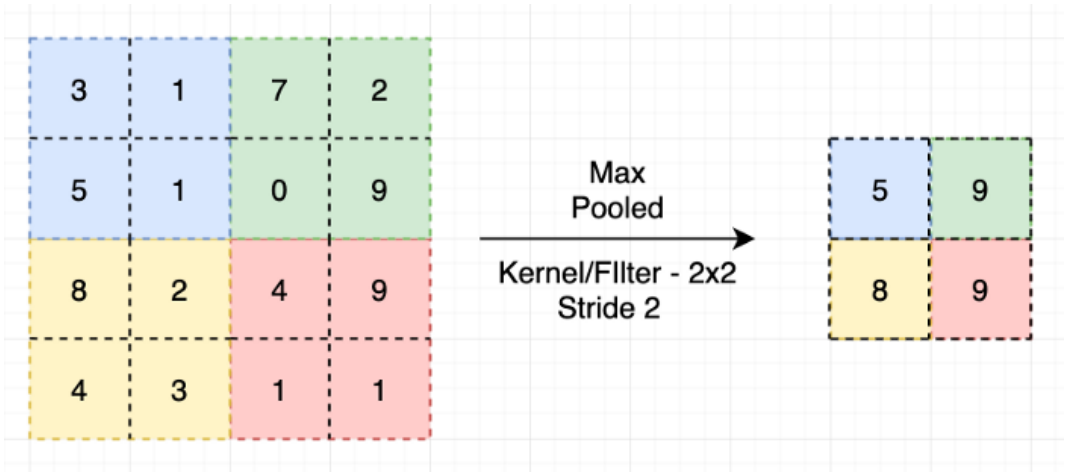


Figure 2.30: Example of 2D max-pooling layer with kernel size 2×2 and stride $(2,2)$. Each number in the right square is obtained by selecting the maximum value among the numbers in the left square that share the same color with it.
Source: https://ai.plainenglish.io/pooling-layer-beginner-to-intermediate-fa0dbdce80eb.

A convolutional neural network can combine convolutional layers, pooling layers and fully connected layers. First, convolutional and pooling layers extract features of the data, that is then used to fed fully connected layers to make a prediction. The following image shows an example of the architecture of a CNN to solve handwritten digits recognition.

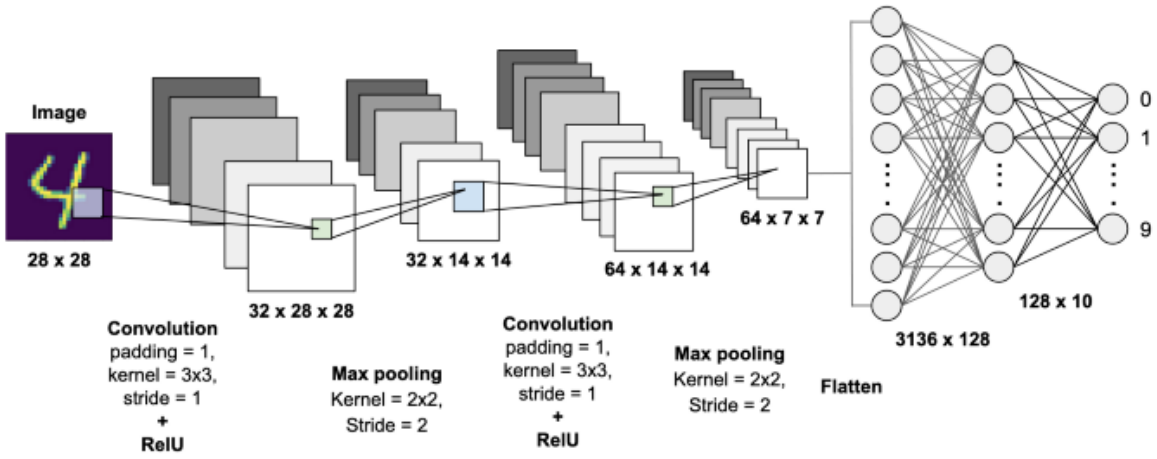


Figure 2.31: The input is a 28×28 1 channel image, a convolutional layer with 32 filters is applied, the padding and kernel size do not change the width and height. The ReLU activation function is applied to each pixel. Then, each 2×2 block is replaced by its maximum value with a max-pooling layer like the one shown in Figure 2.30. Another convolution with 64 filters and ReLU is applied, and then another max-pooling. Finally, the $64 \cdot 7 \cdot 7 = 3136$ are rearranged into a single feature vector used as the input of a fully connected neural network with a single hidden layer (with 128 neurons) and ten output neurons, each one returning the probability of a digit, outputs should be normalized to be interpreted probabilities.
Source: https://jackdani.medium.com/handwritten-digit-recognition-by-using-cnn-99d3ca51ed4e.

2.5 Generative deep learning

So far, every example of machine learning problem or model we have shown had a discriminative nature. The model was trained with a labeled dataset, and it learned to label new examples that were not in the trained dataset. However, the training dataset would not be so different if some of those examples were present, we could say that they were "likely" to be in the dataset. Those are the inputs that the model learns how to label, inputs that resemble other examples in the dataset.

Now, let us think about another problem. What if we wanted our model to generate samples that are likely to be in the dataset? *Generative deep learning* aims to solve this problem, creating models capable of generating new data that do not clash with the training examples. We will follow the references in [5, Part I].

For instance, we could train a discriminative model with paintings, some painted by Van Gogh and some by other artists. The model would predict if a given painting was painted by Van Gogh or not. However, we want a model capable of generating images that look like Van Gogh's paintings, a generative model.

Given an input x, discriminative modeling estimates $p(y|x)$ for each possible label y, the probability of y being the correct label for the input x. Generative modeling cares about the probability of seeing the observation at all, $p(x)$.

Note that we do not want our generative model to generate the same output over and over again. Therefore, it must be *probabilistic* rather than *deterministic*. It cannot be a fixed calculation, the model must include a *stochastic* (random) element that influences the individual samples generated by the model. For example, the model could receive a vector as input, that we would set to random values (*random noise*) to modify the output of the model when we generate a sample. Besides, during training, we have to make sure that the model does not ignore the random noise and generate the same output every time.

2.5.1 Transposed convolution

The problem now is how to create a sample of the dataset from a random noise vector, **transposed convolutional layers** allow us to increase the size of the input. Transposed convolutions are also known as *deconvolutions*.

Considering the case of images, assume that we want to generate a 4×4 image from a 2×2 one. If it was the other way around, we could use a convolution with a 3×3 kernel and unit stride.

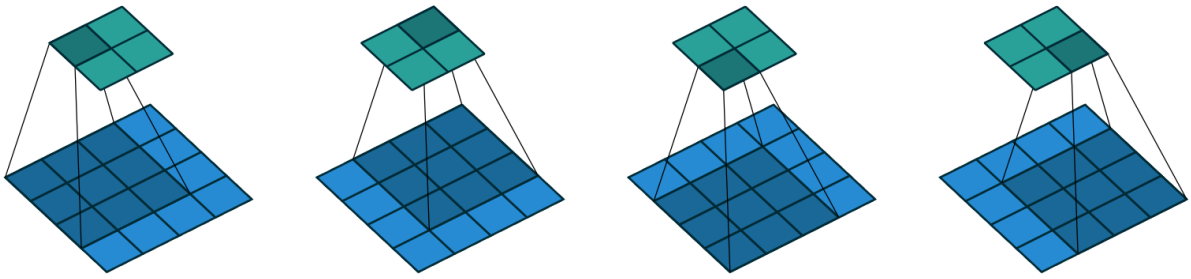


Figure 2.32: ([35, Figure 2.1]) Convolving a 3×3 kernel (grey) over a 4×4 input (blue) using unit strides and no padding results in a 2×2 output (green).

The transposed of this convolution will allow us to generate the 4×4 image from the 2×2 one.

In general, consider a convolution described by m (size of the input), k (kernel), p (padding) and s (stride), where $m+2p-k$ is a multiple of s. The output would have size $n = \frac{m-k+2p}{s} + 1$, as we can see in (2.4.1). Then, it has an associated transposed convolution that is described by: $m' = n$, $\tilde{m}'$, $k' = k$, $s' = 1$ and $p' = k-p-1$, where $\tilde{m}'$ is the size of the stretched input obtained by adding $s-1$ zeros between each input unit, and its output size is $n' = s(n-1)+k-2p$, obtained by solving for the input m in (2.4.1). See [35, Relationship 13]. This applies to both width and height of an image.

Therefore, transposed convolutions are nothing more than convolutions where "zero padding is added between the neurons of the input" when the stride of the original convolution was non-unit.

Let us show some examples of how transposed convolutions work, we will use the same size for width and height. The transpose of the convolution in Figure 2.32 is a convolution with $m' = 2$, $k' = 3$, $s' = 1$ and $p' = k-p-1 = 3-0-1 = 2$.

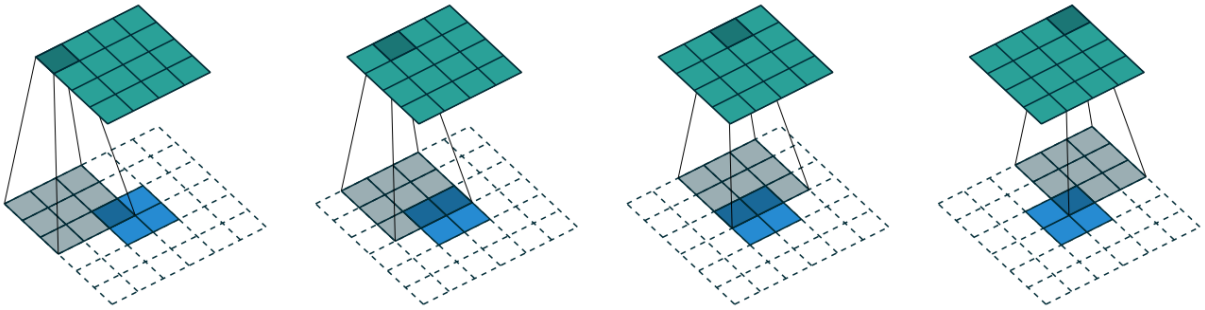


Figure 2.33: ([35, Figure 4.1]) The transpose of the convolution in Figure 2.32 is equivalent to convolving a 3×3 kernel (grey) over a 2×2 input (blue) padded with a 2×2 border of zeros and using unit strides, resulting in a 4×4 output (green).

It is a bit more complicated when the stride is non-unit. The following convolution has $m = 5$, $p = 1$, $k = 3$ and $s = 2$.

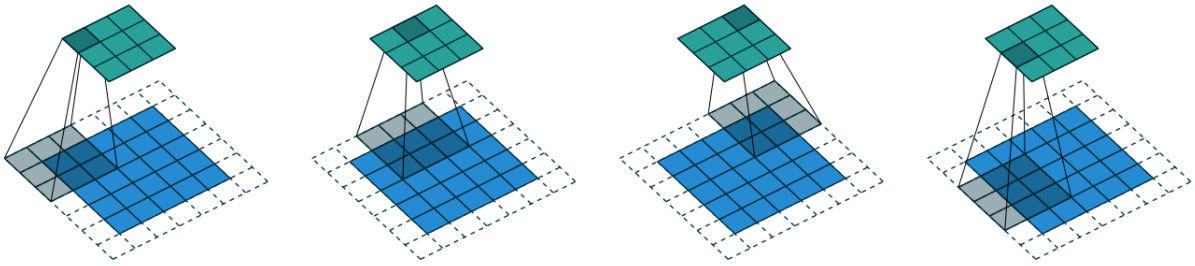


Figure 2.34: ([35, Figure 2.6]) Convolving a 3×3 kernel (grey) over a 5×5 input (blue) using stride 2 and padding 1 results in a 3×3 output (green).

Its transposed has $m' = 3$, $\tilde{m}' = 5$, $k' = 3$, $s' = 1$ and $p' = 1$.

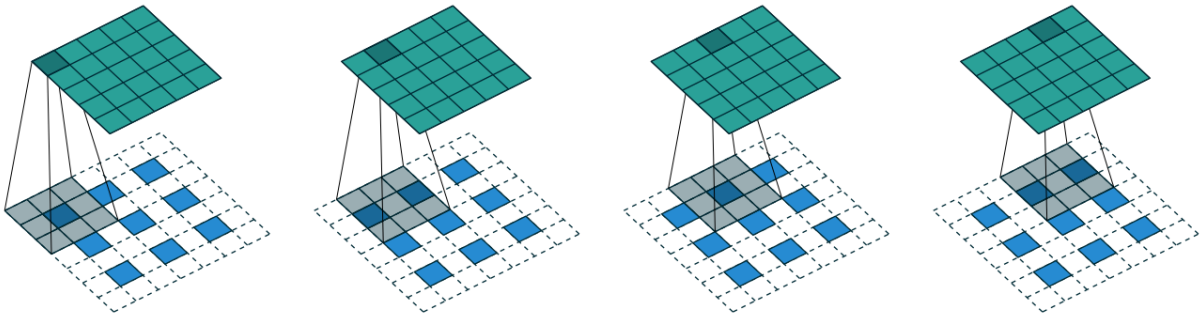


Figure 2.35: ([35, Figure 4.6]) First, $s-1=1$ zeros are inserted between each two neurons of the 3×3 input (blue), obtaining a 5×5 input ($\hat{m}'=5$). That input is used in a convolution with padding 1, unit stride and a 3×3 kernel (grey), which outputs a 5×5 ($n'=5$) image (green).

Using several transposed convolution layers, we can generate a high-dimensional image from a lower-dimensional random noise vector.

2.5.2 Variational autoencoders

An **autoencoder** is a type of artificial network made up of two parts:

- An *encoder* network that compresses high-dimensional input data into a lower dimensional representation vector, via convolutions.
- A *decoder* network that decompresses a given representation vector back to the original domain, via transposed convolutions.

The space of all possible representation vectors is known as latent space.

Consider the following example.

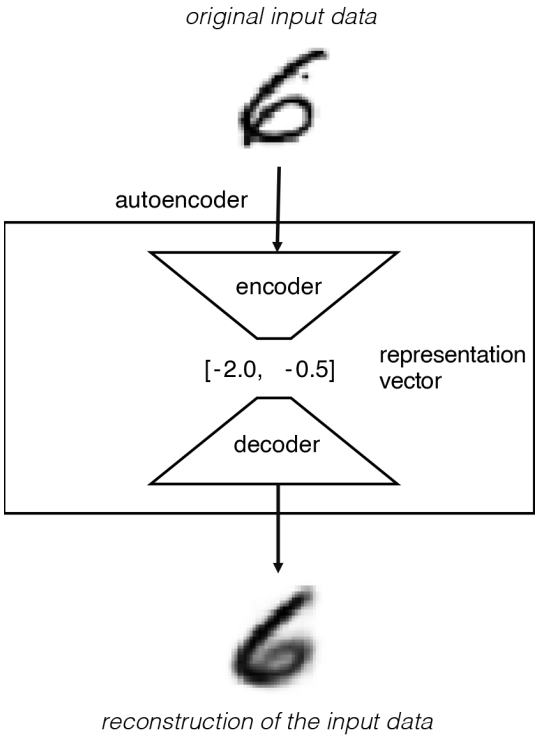


Figure 2.36: ([5, Figure 3-4]) An example of autoencoder that works with images of handwritten digits. The encoder is given an image of a handwritten digits and maps it to a two-dimensional vector. Then, the decoder takes that vector as input and attempts to re-create the original image.

The idea behind this type of architecture is the following. Assume that we manage to train the autoencoder to accomplish the task we have discussed. Then, if we take only the decoder part and feed it with random representation vectors, it should output images similar to the ones used as input for the encoder. However, this idea has three problems.

First, every vector in the latent space may not represent a valid image, so the decoder could transform that vector in an image that does not resemble a handwritten digit.

Second, we would like our decoder to generate images of each handwritten digits with the same probability, and some regions could be wider than others in the latent space.

Third, we would like our model to generate a *continuous* latent space. For example, if the point $(2, -2)$ is decoded to give a satisfactory image of the digit 4, there is no mechanism in place to ensure that the point $(2.1, -2.1)$ also produces a 4.

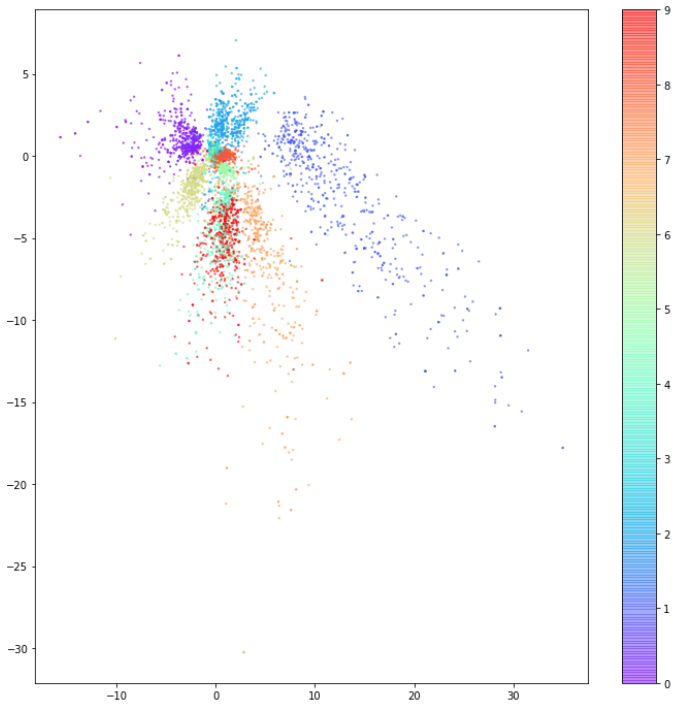


Figure 2.37: ([35, Figure 3-8]) Plot of the latent space of an example autoencoder, colored by digit. There are regions in the space where no digits are mapped, some digits map to larger regions than others and the digit regions do not have smooth frontiers.

Some modifications may be applied to the autoencoder (specifically, to the encoder part) to partially or totally solve this problems, giving place to the **variational autoencoder** (VAE), one of the most fundamental and well-known deep learning architectures for generative modeling. It was introduced in [36].

In a variational autoencoder, the encoder maps each image to a multivariate normal distribution in the latent space, given by a mean vector μ and a covariance matrix Σ. VAE assume that there is no correlation between any of the dimensions in the latent space, so Σ will be a diagonal matrix. We also choose to map to the component-wise logarithm of the variance, that can take any real value, whereas variance values are always positive. Therefore, our encoder now returns two vectors in the latent space: μ and log_var.

Then, the image is mapped to the point

$$z = \mu + \sigma \odot \varepsilon,$$

where

$$\sigma = \sqrt{e^{\text{log\_var}}}$$

and ε is a random point of the latent space sampled from the standard normal distribution. All operations are carried out component-wise.

Finally, the point z is used as the input of the decoder.

A training sample will be randomly mapped anywhere in an area around μ. Hence, the decoder must ensure that all points in the same neighborhood produce similar images when decoded so the reconstruction resembles the original input.

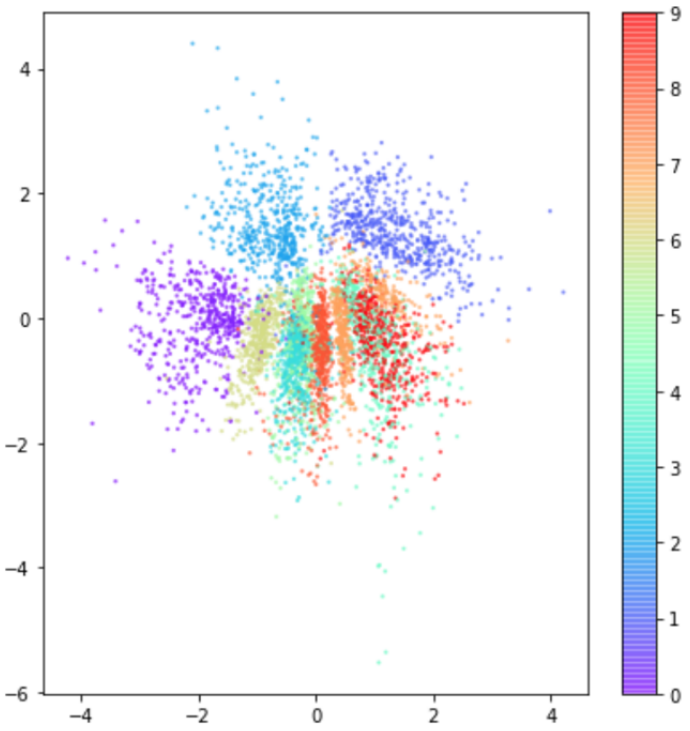


Figure 2.38: ([35, Figure 3-14]) Plot of the latent space of an example of VAE, colored by digit. There is a symmetric area centered at $(0,0)$ where almost each point represent a valid image, the regions corresponding to the digits are more balanced in amplitude than in Figure 2.37, and the frontiers are smoother.

Therefore, when we select a random point around $(0,0)$, the chances of that vector being satisfactorily decoded into a realistic handwritten digit image are greater.

2.5.3 Training a generative model

So far, we have assumed that we somehow managed to train the generative model, but it is non-trivial in practice. In a simple case like the generation of handwritten digits, maybe a pixel-wise L_1 or L_2 loss function comparing the original image and the generated image could suffice. However, this kind of loss functions may not be appropriate for more complex problems.

When it is uncertain where exactly to locate an edge, L_2 loss will encourage a blurry transition in the image, since L_2 is minimized by the expectancy of the conditional probability density function over possible colors. Something similar happens with L_1 loss, minimized by the median. The L_1 loss produces sharper edges than L_2, but the problem persists.

When it comes to the colors, we have the same problems. The L_2 penalty will encourage a grayish color, the average color between all the plausible ones. And the L_1 does not completely solve this problem.

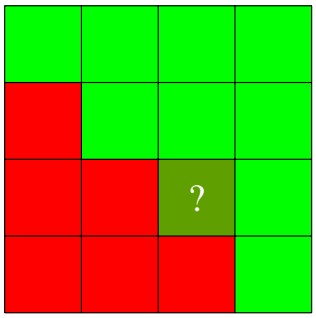


Figure 2.39: In this case, L_2 would fill the uncertain pixel with the average color of its neighborhood, a brownish green. Whereas L_1 penalty would use the median color, that could be red or green, green in this case.

Clearly, the L_1 loss seems better than the L_2 loss, but we will provide a better approach later. We shall discuss another problem that affects all pixel-wise loss functions.

In the case of digits generation, consider the following image of the digit 1.

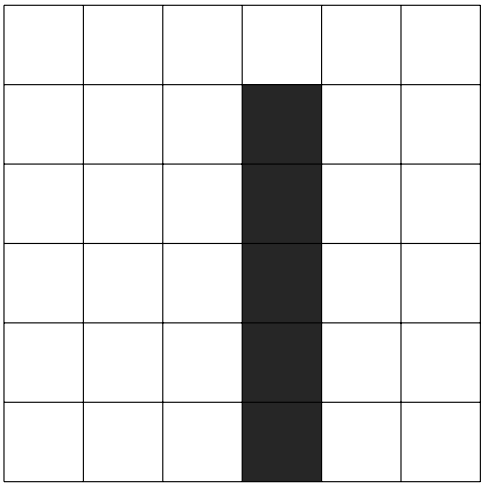
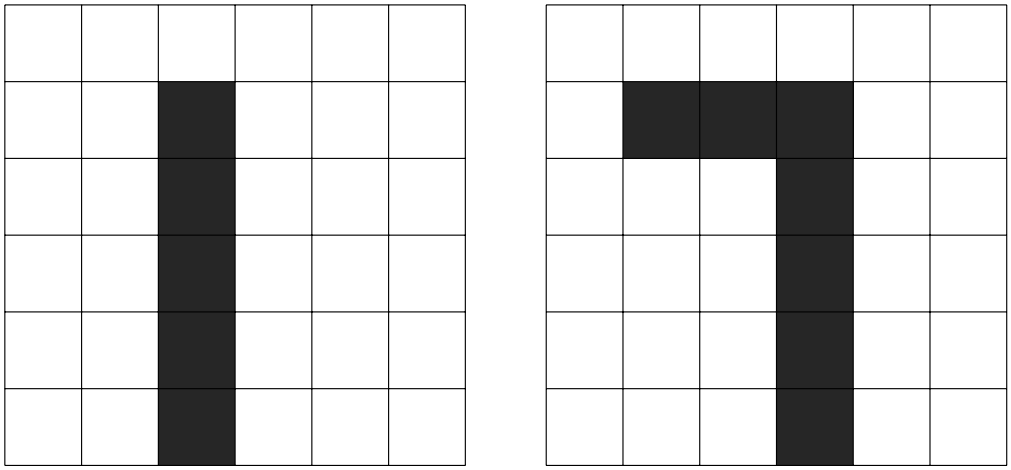


Figure 2.40: Image of the digit 1.

Where the dark pixels are set to 1 and the white pixels are set to 0. Now consider these two possible outputs of the generator.



(a) Possible output, resembles the number 1. (b) Possible output, resembles the number 7.

Figure 2.41: Two possible outputs of the generator.

The images 2.40 and 2.41(a) differ by 10 pixels, so the L_1 and L_2 losses between them are both $\frac{10}{36} \approx 0.278$. The images 2.40 and 2.41(b) differ by only two pixels, so the L_1 and L_2 losses between them are both $\frac{2}{36} \approx 0.056$. Therefore, these two pixel-wise losses consider that the second image is much closer to the original image than the first one. However, images 2.40 and 2.41(a) are identical except for a translation, while the image 2.41(b) looks more like the digit 7 than the digit 1.

We have already seen that convolutional neural networks are not affected by translations, so we will use one to decide if the images generated by the generative model are appropriate.

Adversarial training

Adversarial training has proved to be one of the best approaches to train a generative model.

Consider any generative model, it can be an autoencoder with images as input, or simply a decoder that receives random noise as input, we shall call it the *generator*. We will use another convolutional network as loss function, called *discriminator*.

The discriminator will receive images a input and it will try to guess whether they are real training samples or samples generated by the generator. At the same time, the generator will try to generate images similar enough to the training examples to fool the discriminator. This whole system is called a **Generative Adversarial Network** (GAN), GANs where introduced in [37].

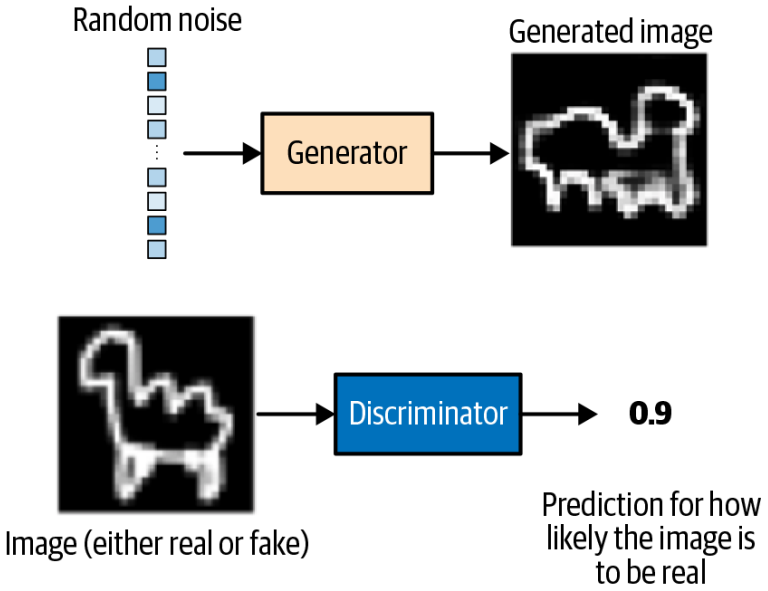


Figure 2.42: ([35, Figure 4-3]) How a GAN works. The generator creates images from random noise. The discriminator receives an image and try to guess whether it is a real image or it was generated by the generator, it can output the probability of it being real.

The training of both networks is carried out at the same time. First, the generator generates a batch (could be one or more) of images from a random noise vector. Then, those images, and some training samples, are given to the discriminator, who tries to tell the real images from the generated ones. The loss function of the discriminator is easy to compute, and we construct a loss function for the generator using the discriminator.

For example, consider the following function ([37, Section 3]).

$$\mathcal{L}_{\text{GAN}}(G,D) = \mathbb{E}_y[\log D(y)] + \mathbb{E}_z[\log(1 - D(G(z)))], \tag{2.5.1}$$

where y represent a real image and z a random noise vector; and G and D are the generator and discriminator respectively.

In practice, y will be an image from the training dataset. When considering only the images in the dataset and the generated images (let them be M in total), the objective function in (2.5.1) can be written as

$$\mathcal{L}_{\text{GAN}}(G,D) = \frac{1}{M}\sum_{i=1}^{M}\big(y_i \log p_i + (1-y_i)\log(1-p_i)\big), \tag{2.5.2}$$

where for each image, the value y_i indicates whether the image is real (1) or generated (0), and p_i is the probability of the image being real according to the discriminator.

Therefore, the objective function in (2.5.1) or (2.5.2) is not new, it is just the binary cross entropy loss that we introduced in Section 2.3.1.

If the generator works well, the discriminator will think that the generated images, $G(z)$, are real, so $D(G(z))$ will be close to 1 and $\mathcal{L}_{\text{GAN}}(G,D)$ will be low. On the other hand, if the discriminator works well, it will recognize that the images from the dataset are real ($D(y)$ will be close to 1), and that the generated images are not real ($D(G(z))$ will be close to 0). Therefore, $\mathcal{L}_{\text{GAN}}(G,D)$ will be high (negative but close to 0).

In conclusion, G tries to minimize $\mathcal{L}_{\text{GAN}}$ and D tries to maximize it. Hence, the generative model we pursuit is $G^* = \arg\min_G \max_D \mathcal{L}_{\text{GAN}}(G,D)$.

The function $\mathcal{L}_{\text{GAN}}$ is differentiable (at least almost everywhere) with respect to the parameters of both the generator and discriminator long as all the activation functions applied are differentiable. Therefore, we can use an optimization algorithm (such as gradient descent or Adam) to maximize $\mathcal{L}_{\text{GAN}}$ in D and minimize it in G.

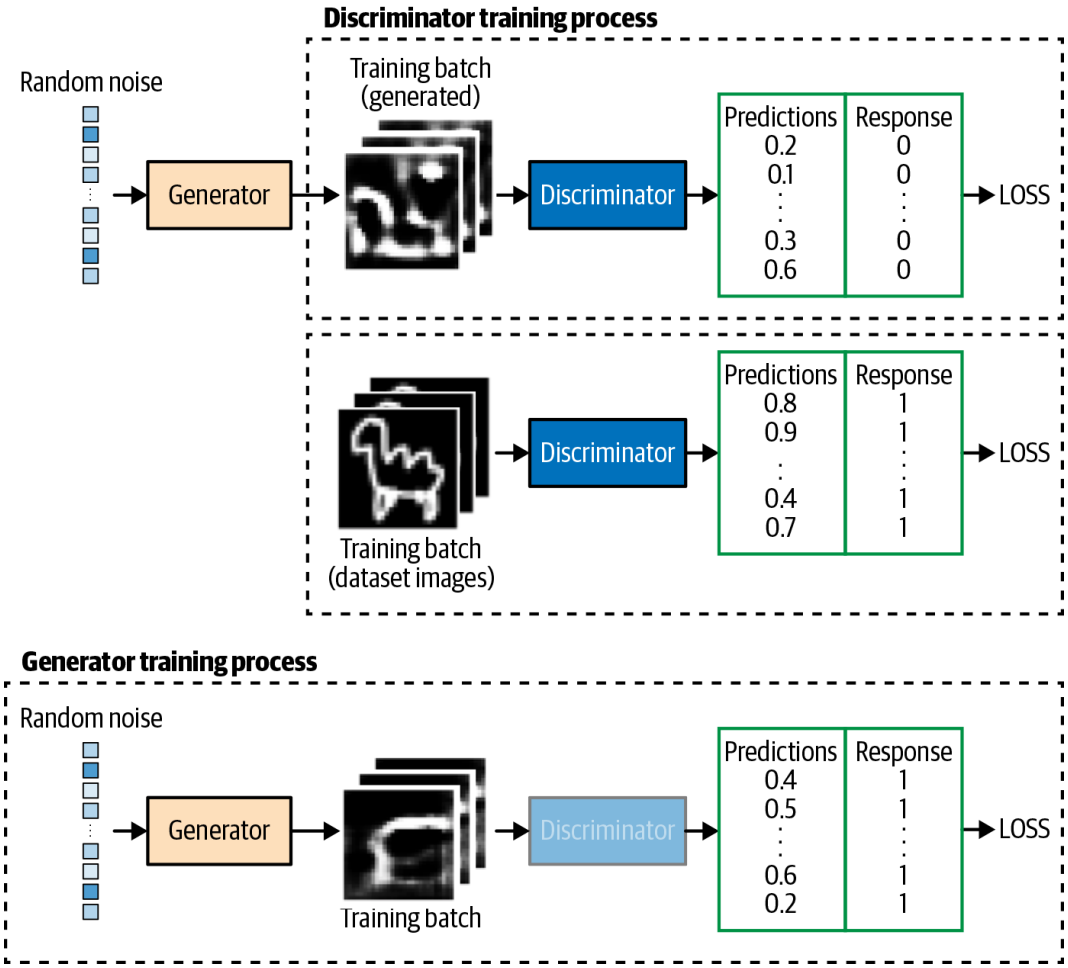


Figure 2.43: ([35, Figure 4-7]) GAN training process. The loss of the generator only depends on the discriminator cataloging his creations as real, the loss will be low when the discriminator maps the generated images to values close to 1. In the case of the generator, its loss depends on both cataloging the generated images as fake (mapping them to values close to 0) and the training images as real (mapping them to values close to 1).

To sum up, the idea is using the binary cross entropy loss (or another loss function) for the discriminator, and constructing a loss function for the generator using the discriminator.

The generator could also be an autoencoder with images as inputs, which will be our case.

During the training of a GAN, we may encounter various problems.

- **Oscillating Loss:** The loss function oscillates wildly and fails to converge. The generator makes it lower and the discriminator makes it higher, but the loss does not exhibit long-term stability.

- **Mode Collapse:** The generator finds a single sample (*mode*) or a limited set of samples that fools the discriminator, as it maps every input to one of those observations. When the discriminator learns that those images are fake, the generator finds another set. When the training stops, the generator will not diversify its output.
- **Uninformative Loss:** The loss function does not provide real information about the quality of the generated images. If the loss is low, either the generator is good or the discriminator is bad.

We will show how to solve some of these problems for the GAN we will use for our purpose. Our generator will be an autoencoder, so we will avoid the mode collapse using a *conditional GAN* ([38]), in which the generator can compare the generated output with the original image. We will manually check the quality of a test set of images after a certain number of epochs to decide when to stop the training.

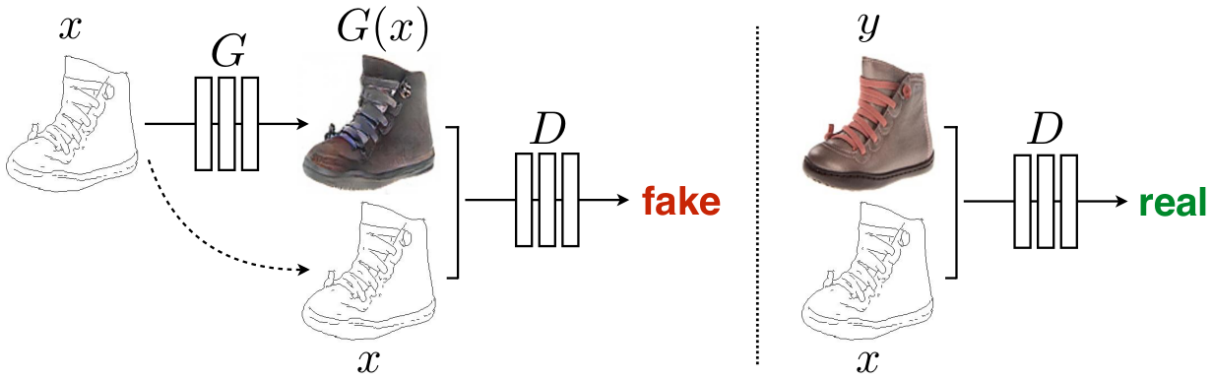


Figure 2.44: ([7, Figure 2]) Example of conditional GAN where the generator creates images of shoes from sketches. It is a conditional GAN because the discriminator uses both the image and the sketch to tell whether the image is real or not. This forces the image not only to be realistic, but also to correspond to the sketch.

2.5.4 Checkerboard artifacts in generated images

This section is motivated by the appearance of checkerboard patterns in the images generated, see Figures 3.38 and 3.14. In [39], an explanation for this problem is provided, along with some advice on how to avoid it. Although we lack the time to put a solution into practice, we present possible causes and solutions for this problem below.

We call *artifacts* to noticeable distortions in the images that should not appear, they can be unnatural patterns (like checkerboard artifacts in Figure 3.38) or elements (see Figures 3.31 and 3.32).

As stated in [39], checkerboard patterns may appear in mages generated by convolutional neural networks that use deconvolutions (transposed convolutions). These artifacts can be caused by several reasons.

There is a high chance for this patterns to appear when convolutions have **"uneven overlap"**, that is, the kernel size is not divisible by the stride. As a consequence, not all pixels of the output image receive signal from the same number of pixels in the input image, in other words, some pixels receive more zeros than others. For instance, in Figure 2.35, the first pixel of the output (green) receives signal from only one pixel of the output (blue) and 8 pixels set to zero (white), whereas the second pixel of the output receives signal from two pixels of the input. There are

other pixels in the second row of the output that receives signal from four pixels of the input and only 5 zeros. This cause some pixels to have a different color or intensity, and occurs with a fixed frequency. We can also see it the other way around, purely as deconvolutions instead of convolutions with inner zeros.

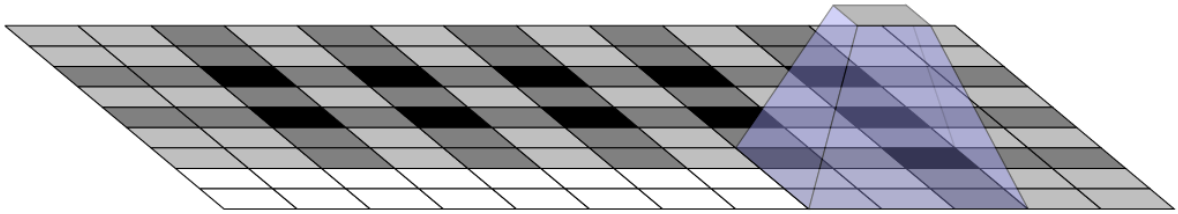
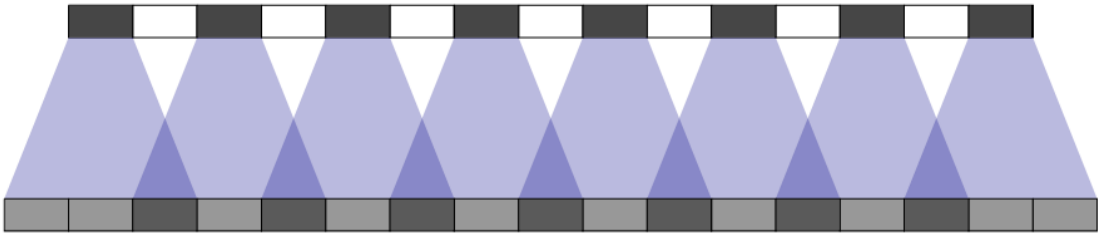
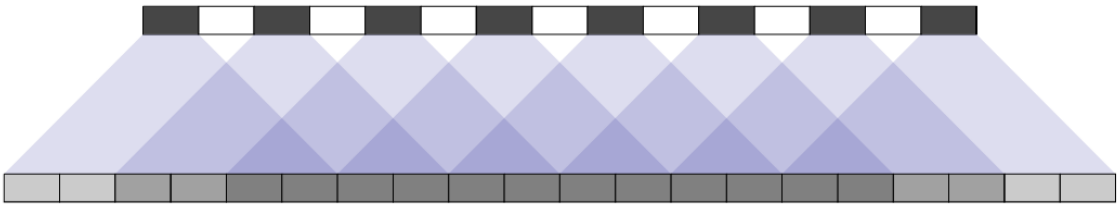


Figure 2.45: ([39]) Example of two-dimensional deconvolution with stride 2 and kernel size 3. Some pixels receive signal "once", while other pixels receive signal twice or even four times. This produces a checkerboard pattern in the output image.

Although these patterns should be recognized as unreal by the generator and the net could learn kernels (weights) that avoid them, this is really difficult in practice. Therefore, deconvolutions with uneven overlap should be avoided when possible.



(a) A one-dimensional deconvolution with uneven overlap ($s = 2$, $k = 3$).



(b) A one-dimensional deconvolution with even overlap ($s = 2$, $k = 6$).

Figure 2.46: ([39]) When using convolutions with even overlap, only border pixels receive a different number of non-zero inputs, which fights the apparition of checkerboard patterns.

Even if we avoid deconvolutions with uneven overlap, the network may learn kernels that cause checkerboard artifacts. This is, in fact, our case. For instance, in a two-dimensional deconvolution with $k = 4$ and $s = 2$, if the weights in the first or last row of the kernel end up close to zero and the same happens to the weights in the first or last column, the kernel produces a similar effect to that of a 3×3 kernel.

According to [39], checkerboard artifacts seem to be strongly linked to deconvolutions with non-unit stride. This artifacts are present even from before the training, so they are not GAN specific. Furthermore, the authors claim to have seen this problem in other kinds of generative models that make use of deconvolutions.

Therefore, other upsampling approaches are advised in order to achieve a specific shape. However, these methods require a complete redesign of the architecture. The main recommendation is to perform upsampling and feature extraction separately. For example, we might resize the image using an interpolation method (like nearest neighbor) and then apply a convolutional layer.

Artifacts in Gradients

As stated in [39], whenever we compute the gradients of a convolutional layer, we do a deconvolution on the backward pass. Thus, checkerboard patterns may appear in the gradient, affecting the convergence of the training and maybe producing checkerboard patterns in the generated outputs. We will not go into how to detect and solve this problem, which is in fact unclear. Nonetheless, the authors make two recommendations to deal with this problem. Avoiding convolutions with non-unit stride in the discriminator (we have three stride 2 convolutions) and not applying the neural network to a fixed position of the image every time, random *jitter* has this covered, and so does our data augmentation method.

CHAPTER 3

Generative model for image lightening

The main goal of this chapter is the production of a fully functional trained generative model capable of transforming an input image into what the image would look like with a suitable illumination. To accomplish this objective, we do not only need to design and code this network, but also the corresponding discriminative model to perform adversarial training for the generative model. First, we shall describe the architecture and parameters of the network in detail, and we will comment the implementation in TensorFlow-Keras.

Besides, we will need a dataset made up of dark images and their corresponding ground truths, what each image would look like if it had been captured with suitable illumination.

Finally, we shall carry out tests to compare the results of the generative model with those of the other alternative, histogram equalization.

The exact scripts and notebooks used throughout this process, among other files that we will mention along the section, can be found in this GitHub repository:

https://github.com/dcabezas98/lux-model

The resolution of test images found along the document was originally really high. The test images shown as results are 3072×4608 ($Height \times Width$) when processed by the model. The Python script `toLow.py` was used to reduce the resolution of the images (generally to 512×768) so the document does not get too large. Is applies interpolation using pixel area relation from the OpenCV Python package.

3.1 Dataset

We generate our own dataset from the images found in See-in-the-Dark (SID) dataset (see [6, Section 3]). Which contains 5094 short-exposure images, each with a reference long-exposure (ground truth) image. The images correspond to photographs captured with two cameras: Fujifilm X-T2 and Sony α7S II. The same long-exposure image may serve as ground truth for several short exposure images.

The exposure time is the length of time that the camera sensor is exposed to light. The higher the exposure time, the greater amount of light that reaches the sensor and the higher the brightness of the image. The exposure time for the input images was set to 1/30, 1/25 or 1/10 seconds. The corresponding reference (ground truth) images were captured with an exposure time of 10 or 30 seconds. Due to the long exposure time, all images correspond to static scenes.

These images are in raw format, `.RAF` for the photographs taken with the Fujifilm camera and `.ARW` for the ones taken with the Sony camera. According to [6], the resolution of the images is 4240×2832 ($W \times H$) pixels for Sony and 6000×4000 pixels for the Fuji images, although we found it to be 4256×2848 and 6032×4032 respectively.

We want our model to work with a standard image format (the input of the generator must be a RGB image), so we must postprocess[1] the images first. For that purpose we make use of rawpy, a Python wrapper for the LibRaw library. It is important to select the options `use_camera_wb=True` (use camera white balance) for the postprocessed images colors to look as similar as possible to the raw images colors; and `no_auto_bright=True` for the short-exposure postprocessed images to actually be dark, since it is enough information in the raw sensor data to generate an image with suitable illumination, very similar to the ground truth image except for some color noise typical of photographs taken in low-light environments (see Figure 1.16 and comments above).

The SID dataset has its photos divided in separate folders to distinguish photographs taken with the Fuji camera from the ones taken with the Sony camera, and also short-exposure images from long-exposure ones. A short-exposure image captured with the Fuji camera would be in the `SID/Fuji/short` folder, and its name would look as follows:

`S0XXX_YY_Zs.RAF`,

where:

- `S=0,1,2` indicates if the image was intended to be used in training, test or validation respectively.
- `XXX` denotes the number of the corresponding long-exposure image among images captured with the same camera.
- `YY` denotes the image number among short exposure images associated with the long-exposure image `XXX`.
- `Z=0.1,0.04,0.033` indicates the exposure time used to capture the image.

It's corresponding long-exposure image would be located in `SID/Fuji/long`, and its name would be `S0XXX_00_Ws.RAF`, where `W` could be `10` or `30` seconds, depending on the exposure time used to capture the image.

In case the image had been captured with the Sony camera, the folder would have been `Sony` instead of `Fuji`, and the extension would have been `.ARW` instead of `.RAF`.

The `S` part of the name is determined by the `XXX` number, this means that if a image was intended for a purpose (training, test or validation), no images corresponding to the same ground truth are used for a different purpose.

[1] Processing the raw sensor data to store it in a standard image format with a good quality-size tradeoff. Several steps are performed, usually demosaicing, tone mapping, white balancing, denoising, sharpening and compression. See [16, Post-processing pipeline].

```
SID
├── Fuji
│   ├── long
│   │   └── S0XXX_00_Ws.RAF
│   └── short
│       └── S0XXX_YY_Zs.RAF
└── Sony
    ├── long
    │   └── S0XXX_00_Ws.ARW
    └── short
        └── S0XXX_YY_Zs.ARW
```

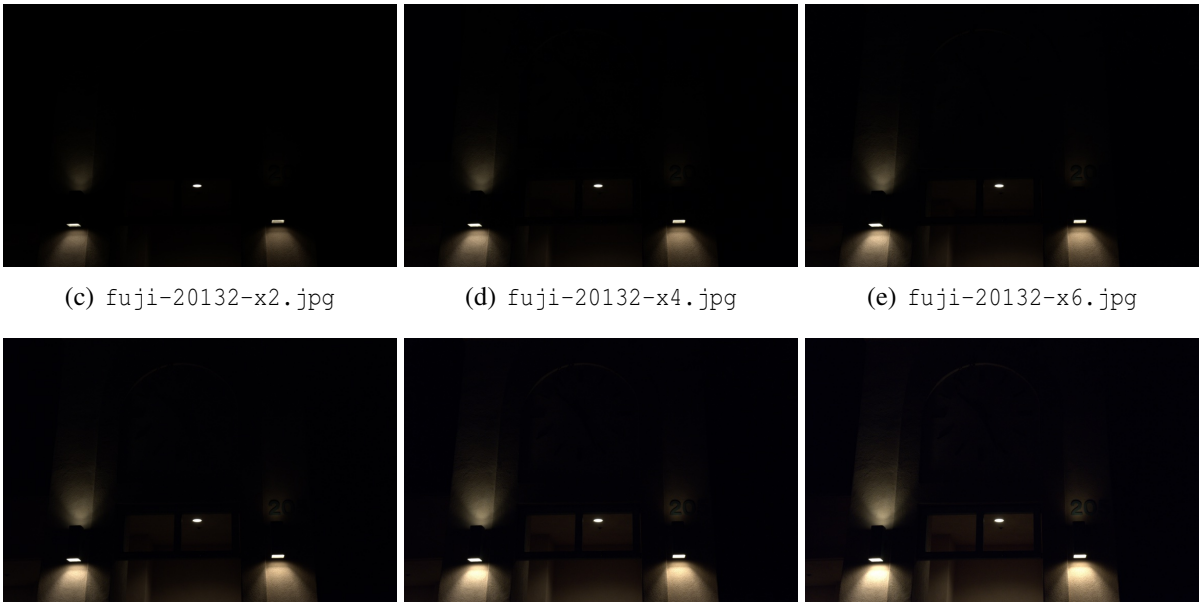
Figure 3.1: The images of the SID dataset are stored in separate folders by camera (Fuji or Sony) and exposure time (short or long). We can also observe the name format of the images.

Due to the large size of the dataset (slightly greater than 200GB) the we will only select one short-exposure image for each long-exposure one, in order to obtain a dataset much smaller but with almost the same diversity of images. Altogether, there are 424 (231 from the Sony camera and 193 from the Fuji one) long-exposure images, for each of them, the short-exposure image with the number `YY=00` happens to been captured using an exposure time of 1/10 seconds, we shall select this image. The images of the SID dataset are extremely dark, so this images are already dark enough for our purpose, and present higher Signal-to-noise ratio than the ones take with exposure times of 1/25 or 1/30 seconds. Therefore, we shall remove the `YY` and `Zs` parts of the names for the postprocessed images.

Once the images are postprocessed, we are no longer concerned about the camera that took the photos, so we shall store all of them in the same folder. However, a Fuji image can share the same number (`XXX`) with a Sony image, so we add `fuji` or `sony` as a prefix of the name for the postprocessed images.

As we said, the resulting images are extremely dark, and we want our model to improve the illumination of images with different levels of darkness. To obtain input images with a greater variety of illumination, we adjust the parameter `bright` of the function `postprocess` to scale the brightness of the postprocessed images. The values we will use are 1 (default), 2, 4, 6, 8, 12, 16. We shall add the suffix `xb` to the name of the postprocessed images, where `b=1,2,4,6,8,12,16` indicates the brightness scale factor. As a consequence, the resulting dataset has 424 ground truth images and $424 \cdot 7 = 2968$ input images, with each long-exposure image corresponding to 7 short-exposure images.

(a) `fuji-20132.jpg` (Ground truth) (b) `fuji-20132-x1.jpg`



(c) `fuji-20132-x2.jpg` (d) `fuji-20132-x4.jpg` (e) `fuji-20132-x6.jpg`

(f) `fuji-20132-x8.jpg` (g) `fuji-20132-x12.jpg` (h) `fuji-20132-x16.jpg`

Figure 3.1: Ground truth and all the inputs with different brightness scale factor for the image 132 among the images captured with the Fuji camera. The image used to belong to the validation set (`S=2`). The image has been postprocessed into JPEG format.

For example, if we use the format JPEG, the postprocessed images we shall train our model with would now be stored with the following structure and name format:

```
jpeg
├── long
│   ├── fuji-S0XXX.jpg
│   └── sony-S0XXX.jpg
└── short
    ├── fuji-S0XXX-xb.jpg
    └── sony-S0XXX-xb.jpg
```

Figure 3.2: Structure and name format of the images postprocessed into JPEG. We merged the Sony and Fuji images into the same folder and added a prefix to distinguish images with the same number. The suffix `xb` indicates the brightness scale factor for the input images.

For the experimentation developed in [6], the train set (`S=0`) included 296 ground truth images the test set (`S=1`), 91, and the validation set (`S=2`), 37 ground truth images. The validation set was meant to be examined by the authors in order to monitor the behavior of the model while it was being designed and trained. The test set was used to evaluate the model by comparison with other techniques in A/B tests[2] deployed on the Amazon Mechanical Turk platform. Given that we lack such kind of means, we will leave the previous validation set (`S=2`) to monitor the performance of the model during and after the training, whereas all the images with `S=0,1` will be used for training. Altogether, we are left with 37 ground truth images for testing and $296 + 91 = 387$ for training, each one with its seven corresponding inputs with different brightness scales.

[2]Tests where an individual is shown two images and decides which one has higher quality.

There are several standard image formats we can use. Since we intend to upload the dataset to Google Drive so we can read it and carry out the training in Google Colab, we do not want our dataset to be excessively large. Keeping this in mind, JPEG seems like the most appropriate format to save the postprocessed images, given that formats that use lossless compression such as PNG or TIFF lead to 100GB or greater datasets. On the other hand, storing the postprocessed images in JPEG format could undermine their quality, but due to the high resolution the images already present, we can afford to sacrifice some quality for the sake of size. Therefore, we shall use the JPEG format to store our dataset, that will take up over 15GB. Nonetheless, we shall store the test images also in PNG and TIFF formats to evaluate the performance of our final model also on lossless image formats.

The JPEG images were saved using the Pillow library, setting the quality to 95%, the highest quality for which none of the compression algorithms get disabled. For the PNG and TIFF images, the imageio library was used.

Neither the SID raw data nor the postprocessed data can be found in the GitHub repository, but the original SID dataset can be downloaded from the repository that hosts the implementation of [6]:

https://github.com/cchen156/Learning-to-See-in-the-Dark,

and the script `raw_to_postprocessed.py` (found in our repository) can be used to postprocess and save the images we will train and test our model with, it employs `joblib.Parallel` to speed up the process.

3.2 Architecture of the GAN

Designing a loss function to measure how well lit a photograph is can be a really difficult task, so we will use a GAN. The architecture of the neural network we will use can be mostly found in [7], such architecture has been shown to work for different purposes, depending on the dataset used to train the model. We shall apply some minor modifications, since we are in possession of images with larger resolution instead of 256×256 pixels, which is the case of [7].

As for the implementation, [40] holds the code for [7]. We will rely on this implementation for ours. Our code for both models can be found here:

https://github.com/dcabezas98/lux-model/blob/main/p2pGAN_models-train.ipynb.

Generator

The generator is an autoencoder that receives RGB images as inputs and outputs RGB images of the same dimension. The value range of each pixel is $[-1, 1]$. The generator has an encoder part made up of 3-layer blocks that we shall name *downsample blocks*, and a decoder part made up of 4-layer blocks that we shall name *upsample blocks*.

Each downsample block has a Convolutional 2D layer, a batch normalization layer and a leaky ReLU layer.

The convolution always use 4×4 kernels with stride 2, and the zero padding is adjusted so both the height and width of the output image are exactly half of those of the input, that is, $p_h = p_w = 1$, obtained from (2.4.1). The number of filters is different each time. Bias is not used unless we say otherwise.

The slope for the negative part of the leaky ReLU is $\alpha = 0.3$, Keras default parameter.

Each upsample block has a transposed convolutional 2D layer, a batch normalization layer, a dropout layer and a (non-leaky) ReLU layer.

The transposed convolution is the transpose of the one in the downsample block. See section 2.5.1. The number of filters is also different each time.

The dropout layer randomly drops input units with a probability of 0.5.

The encoder has 8 downsample blocks, the first upsample block omits batch normalization and uses bias in the convolution. The other 7 blocks are identical to the one described. The number of filters of the convolution in each block is: 64, 128, 256, 512, 512, 512, 512 and 512 respectively. Each block halves both the height and the width input, so the input image its reduced by a factor of 256 to 1 in both width and height. Hence, we shall assume that the input image has shape $256h \times 256w \times 3$ (where h and w are integers), we will show how to transform arbitrary images later. The output of the encoder (and input of the decoder) will be $h \times w \times 512$.

Then, the decoder transforms its input back into a $256h \times 256w \times 3$ image. It has 7 downsample blocks, where only the first three apply dropout. The number of filters of the transposed convolution in each block is: 512, 512, 512, 512, 256, 128 and 64 respectively. The transposed convolution in each block doubles the width and height of the data until it reaches the shape $128h \times 128w \times 64$, finally a transposed convolution layer with identical parameters and 3 filters is applied to reach the original shape $256h \times 256w \times 3$, and a tanh layer sets the pixel values into the valid range.

The encoder and the decoder are mirror images. Note that the dimension of the representation array is not fixed, and neither is the shape of the images.

We intend the generator to transform a dark image into a representation vector and then recreate a version of it with suitable illumination, but the information in the representation vector may not be enough to recreate the same image. To solve this problem, we add *skip connections*, that allow each upsample block (and the final layer) in the decoder to access the output of its mirror layer in the encoder, the outputs of the previous layer and the mirror layer (both outputs have the same shape) are concatenated. This doubles the number of channels in the input of each block in the decoder, and would not be possible if the encoder and decoder were not mirror images, since the shapes (width and height) of the images would not match. In the case of the first upsample block, the mirror layer and the previous layer are the same, so we do not add a skip connection. This architecture is called a U-Net ([41]).

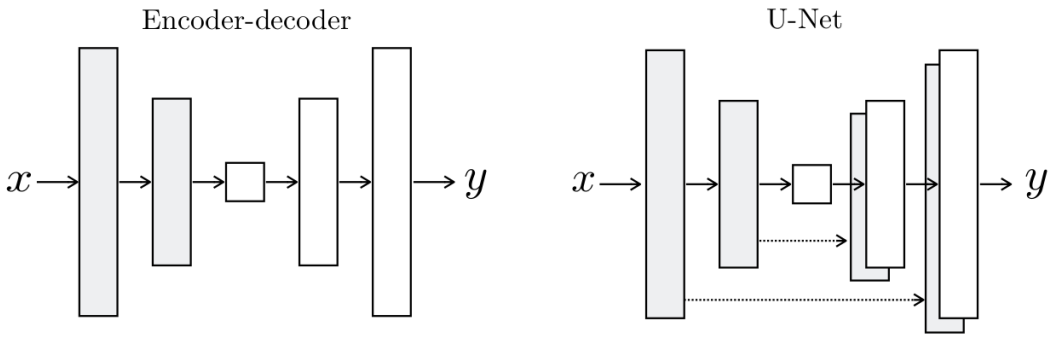


Figure 3.3: ([7, Figure 3]) Examples of generators. The left one is a classic symmetric encoder-decoder; the right one is an U-Net like our generator. Each layer of the decoder receives as input the concatenation of the outputs of the previous layer and the mirror layer in the encoder.

The Keras API provides the utility `plot_model`, that allow us to plot the architecture of the model along with the shape of the input data in each block (sequential layer in TensorFlow).

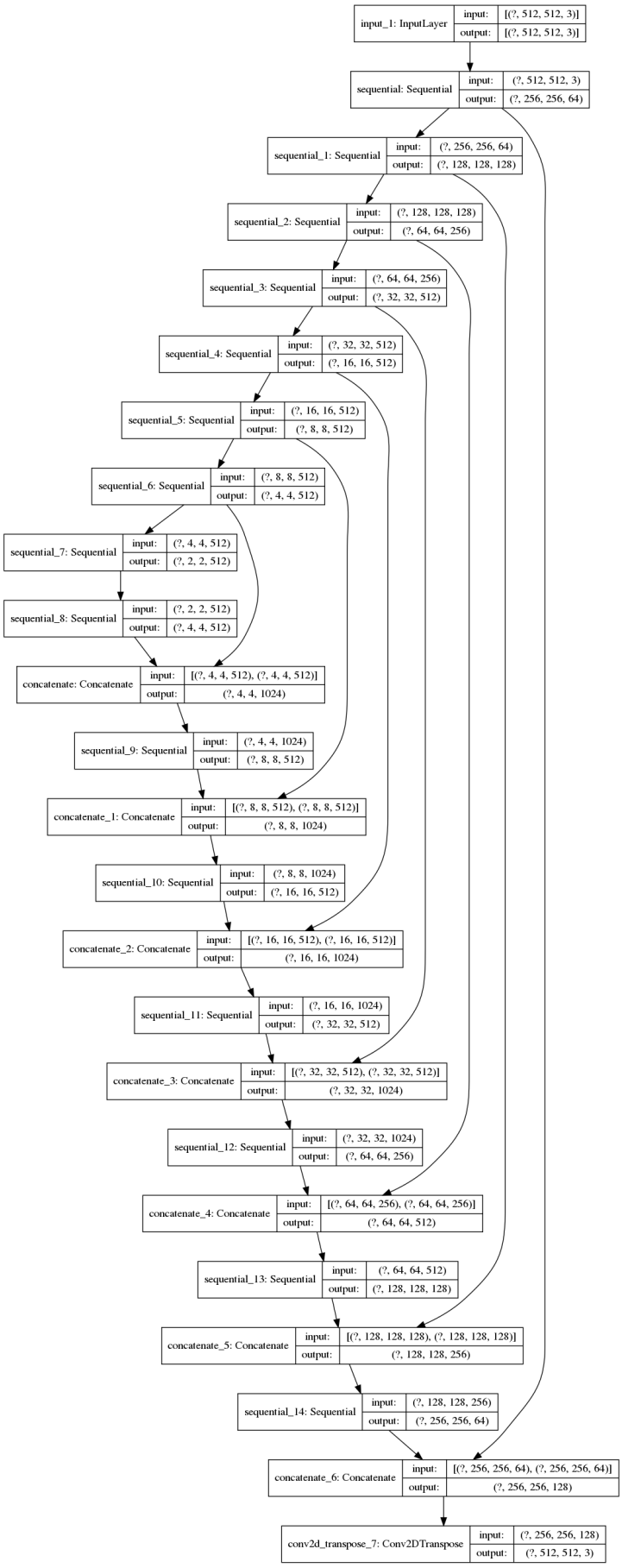


Figure 3.4: Architecture of the generator, along with the shape of the input and output of each layer (height $\times$ width $\times$ channels) when the input image has a width and height of 512 pixels. The symbol ? indicates that the batch size could be any value, but we will use bs $= 1$.

TensorFlow models also have the utility `summary`, that reports on the number of trainable parameters in each layer.

```
Layer (type)                    Output Shape         Param #   Connected to
==================================================================================================
input_1 (InputLayer)            [(None, 512, 512, 3)]  0
__________________________________________________________________________________________________
sequential (Sequential)          (None, 256, 256, 64)  3136     input_1[0][0]
__________________________________________________________________________________________________
sequential_1 (Sequential)        (None, 128, 128, 128) 131584   sequential[0][0]
__________________________________________________________________________________________________
sequential_2 (Sequential)        (None, 64, 64, 256)   525312   sequential_1[0][0]
__________________________________________________________________________________________________
sequential_3 (Sequential)        (None, 32, 32, 512)   2099200  sequential_2[0][0]
__________________________________________________________________________________________________
sequential_4 (Sequential)        (None, 16, 16, 512)   4196352  sequential_3[0][0]
__________________________________________________________________________________________________
sequential_5 (Sequential)        (None, 8, 8, 512)     4196352  sequential_4[0][0]
__________________________________________________________________________________________________
sequential_6 (Sequential)        (None, 4, 4, 512)     4196352  sequential_5[0][0]
__________________________________________________________________________________________________
sequential_7 (Sequential)        (None, 2, 2, 512)     4196352  sequential_6[0][0]
__________________________________________________________________________________________________
sequential_8 (Sequential)        (None, 4, 4, 512)     4196352  sequential_7[0][0]
__________________________________________________________________________________________________
concatenate (Concatenate)        (None, 4, 4, 1024)    0        sequential_8[0][0]
                                                                  sequential_6[0][0]
__________________________________________________________________________________________________
sequential_9 (Sequential)        (None, 8, 8, 512)     8390656  concatenate[0][0]
__________________________________________________________________________________________________
concatenate_1 (Concatenate)      (None, 8, 8, 1024)    0        sequential_9[0][0]
                                                                  sequential_5[0][0]
__________________________________________________________________________________________________
sequential_10 (Sequential)       (None, 16, 16, 512)   8390656  concatenate_1[0][0]
__________________________________________________________________________________________________
concatenate_2 (Concatenate)      (None, 16, 16, 1024)  0        sequential_10[0][0]
                                                                  sequential_4[0][0]
__________________________________________________________________________________________________
sequential_11 (Sequential)       (None, 32, 32, 512)   8390656  concatenate_2[0][0]
__________________________________________________________________________________________________
concatenate_3 (Concatenate)      (None, 32, 32, 1024)  0        sequential_11[0][0]
                                                                  sequential_3[0][0]
__________________________________________________________________________________________________
sequential_12 (Sequential)       (None, 64, 64, 256)   4195328  concatenate_3[0][0]
__________________________________________________________________________________________________
concatenate_4 (Concatenate)      (None, 64, 64, 512)   0        sequential_12[0][0]
                                                                  sequential_2[0][0]
__________________________________________________________________________________________________
sequential_13 (Sequential)       (None, 128, 128, 128) 1049088  concatenate_4[0][0]
__________________________________________________________________________________________________
concatenate_5 (Concatenate)      (None, 128, 128, 256) 0        sequential_13[0][0]
                                                                  sequential_1[0][0]
__________________________________________________________________________________________________
sequential_14 (Sequential)       (None, 256, 256, 64)  262400   concatenate_5[0][0]
__________________________________________________________________________________________________
concatenate_6 (Concatenate)      (None, 256, 256, 128) 0        sequential_14[0][0]
                                                                  sequential[0][0]
__________________________________________________________________________________________________
conv2d_transpose_7 (Conv2DTrans) (None, 512, 512, 3)   6147     concatenate_6[0][0]
==================================================================================================
Total params: 54,425,923
Trainable params: 54,415,043
Non-trainable params: 10,880
```

Now, we can appreciate the magnitude of the number of trainable parameters of the generator.

Discriminator

The discriminator is a classic CNN that receives RGB images as inputs. The discriminator also has access to the original input (the dark image), the channels of both images are concatenated, so the input has 6 channels. Therefore, we are using a *conditional GAN*, that prevents *mode collapse*.

First, the discriminator has 3 downsample blocks with 64, 128 and 256 filters respectively. The first block omits batch normalization and uses bias. Then, a convolution with zero padding $p_h = p_w = 1$, a 4×4 filter and stride 1 is applied, also without bias, followed by a batch normalization layer and a leaky ReLU with $\alpha = 0.3$. Finally, another convolution is applied, with padding $p_h = p_w = 1$, a 4×4 filter and stride 1, but this time with just one filter.

The output is a 1 channel image when each pixel indicates if the discriminator considers real a 70×70 region of the original image, that region contains the pixels that affect the pixel of the output, and it is called its receptive field. Receptive fields of the pixels overlap. This is called a convolutional *PatchGAN*, because it penalizes structure at the scale of image patches. Pixels separated by more than the patch diameter (70) are assume independent, in other words, the image is treated as a *Markov random field*.

The value 70 comes from simple convolution arithmetic, a patch of size (both height and width) n in the output becomes affected by a patch of size $s(n-1)+k$ in the input.

Therefore, a pixel of the output is affected by a patch of size $1(1-1)+4 = 4$ of the input of the last layer, which is affected by a patch of size $1(4-1)+4 = 7$ of the input of the penultimate layer. Then, the convolutions with stride two come in play, and a patch of size $2(7-1)+4 = 16$ of the input of the third-to-last layer affects the pixel of the output. Two more convolutions lead to $2(16-1)+4 = 34$ and finally $2(34-1)+4 = 70$. Note that the zero padding has nothing to do with the receptive field.

Just like the generator, the shape of the discriminator also depends on the input.

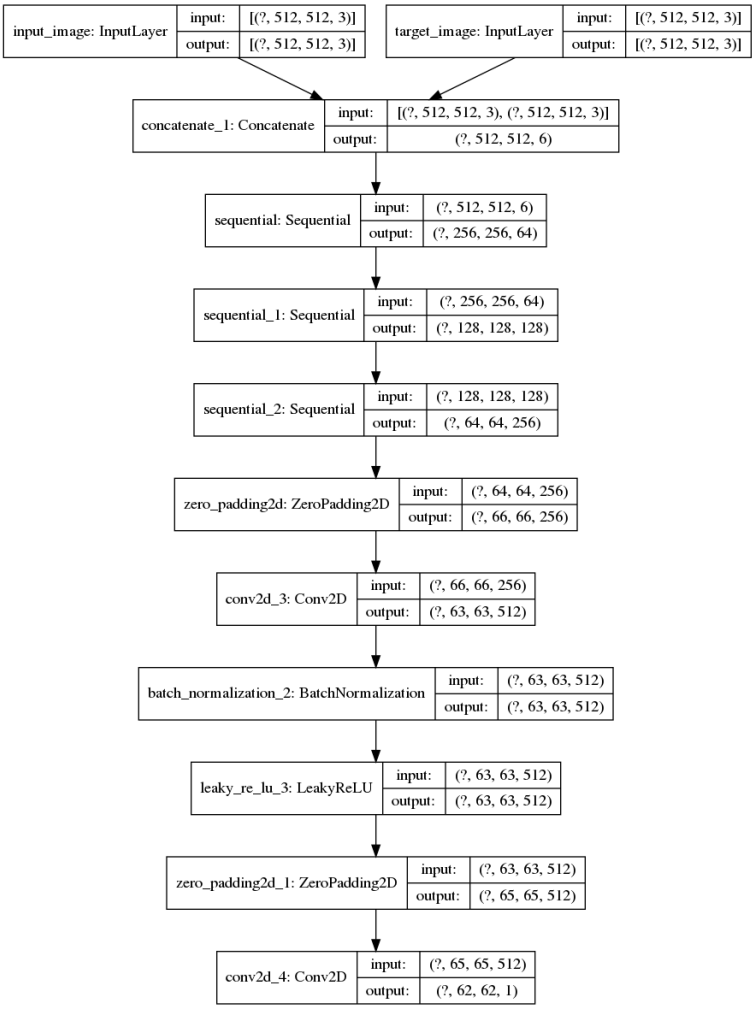


Figure 3.5: Architecture of the discriminator, along with the shape of the input and output of each layer (height $\times$ width $\times$ channels) when the input image has a width and height of 512 pixels. The symbol ? indicates that the batch size could be any value, but we will use bs $= 1$.

We can also check the discriminator summary. Its number of trainable parameters is considerably lower, since its architecture is much simpler.

```
Layer (type)                 Output Shape              Param #   Connected to
==================================================================================================
input_image (InputLayer)     [(None, 512, 512, 3)]     0
__________________________________________________________________________________________________
target_image (InputLayer)    [(None, 512, 512, 3)]     0
__________________________________________________________________________________________________
concatenate (Concatenate)    (None, 512, 512, 6)       0         input_image[0][0]
                                                                 target_image[0][0]
__________________________________________________________________________________________________
sequential (Sequential)      (None, 256, 256, 64)      6208      concatenate[0][0]
__________________________________________________________________________________________________
sequential_1 (Sequential)    (None, 128, 128, 128)     131584    sequential[0][0]
__________________________________________________________________________________________________
sequential_2 (Sequential)    (None, 64, 64, 256)       525312    sequential_1[0][0]
__________________________________________________________________________________________________
zero_padding2d (ZeroPadding2D) (None, 66, 66, 256)     0         sequential_2[0][0]
__________________________________________________________________________________________________
conv2d_3 (Conv2D)            (None, 63, 63, 512)       2097152   zero_padding2d[0][0]
__________________________________________________________________________________________________
```

```
batch_normalization_2 (BatchNorm) (None, 63, 63, 512)   2048      conv2d_3[0][0]
leaky_re_lu_3 (LeakyReLU)         (None, 63, 63, 512)   0         batch_normalization_2[0][0]
zero_padding2d_1 (ZeroPadding2D)  (None, 65, 65, 512)   0         leaky_re_lu_3[0][0]
conv2d_4 (Conv2D)                 (None, 62, 62, 1)     8193      zero_padding2d_1[0][0]
==================================================================================================
Total params: 2,770,497
Trainable params: 2,768,705
Non-trainable params: 1,792
```

We shall show some examples of how the generator works. Once trained, the generator should be capable of detecting the patches of the input image that looks fake. We will mix some generator predictions from epoch 60 (should look less realistic than final results) with the ground truth images, and see what the discriminator outputs. This examples are generated with the notebook p2pGAN_discriminator.ipynb.

(a) Input 1: dark image. (b) Input 2: generated/real image. (c) Output.

Figure 3.6: Generator prediction for `fuji-20018-x8`. The upper left triangle is produced by the generator and the lower right triangle belongs to the ground truth image.

(a) Input 1: dark image. (b) Input 2: generated/real image. (c) Output.

Figure 3.7: Generator prediction for `fuji-20069-x6`. The upper half of the input is real and the lower half is generated. It can be seen how the discriminator outputs higher values for the real patches and lower values for the generated patches.

(a) Input 1: dark image. (b) Input 2: generated/real image. (c) Output.

Figure 3.8: Generator prediction for `fuji-20167-x2`. The upper left triangle is real and the lower right triangle is generated.

(a) Input 1: dark image. (b) Input 2: generated/real image. (c) Output.

Figure 3.9: Generator prediction for `fuji-20167-x16`. The upper left triangle is real and the lower right triangle is generated.

All the weights of both the generator and the discriminator are initialized randomly using a Gaussian normal with mean 0 and standard deviation 0.02.

When batch normalization is applied, it affects each channel independently. Since we will use batch size 1, 256×256 images arrive to the "bottleneck" of the generator (the output of the encoder) with only one value per channel (shape $1 \times 1 \times 1 \times 512$) and batch normalization would set it to zero. In [7], removing the batch normalization layer in the last block of the encoder is advised, since they use 256×256 images for training. In practice, images will be always larger than that, so we will use 512×512 images for training, which allows us to maintain the batch normalization layer in the last downsample block of the generator.

Versions of Figures 3.4 and 3.5 where downsample and upsample blocks (sequential layers) are expanded can be found here:

https://github.com/dcabezas98/lux-model/tree/main/images,

but the shape of the images does not allow us to display them in this document.

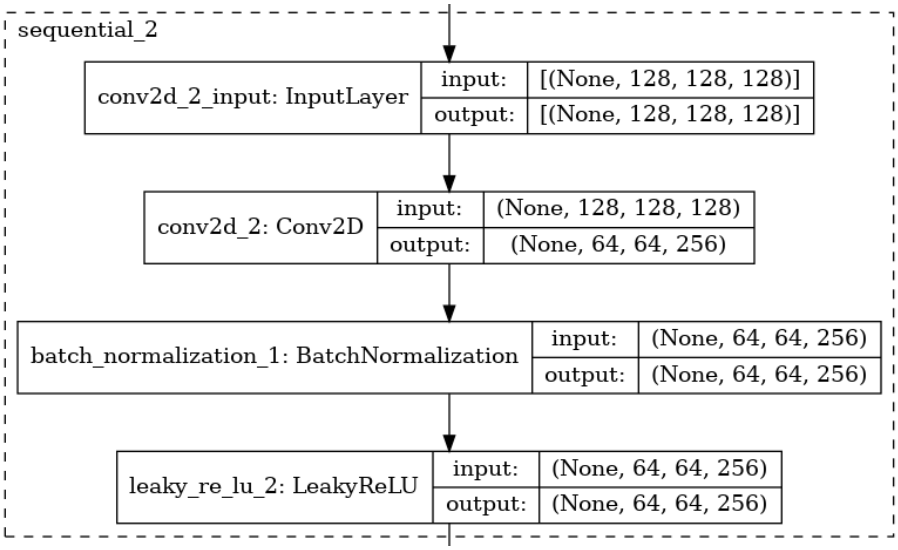


Figure 3.10: Expansion of an upsample block. None also indicates that the batch size is arbitrary.

3.3 Training

The file

https://github.com/dcabezas98/lux-model/blob/main/p2pGAN_models-train.ipynb

also holds the coding of the reading and processing of the dataset, as well as the training process.

Data reading and processing

First, we match every short-exposure image (input) with its corresponding long-exposure image (target), each long-exposure image appears 7 times. Then, the pairs are split into two lists, training and testing.

When the images are read, some transformations are applied.

Training images are resized to 1024×1536 (height $\times$ width) using nearest neighbor interpolation. Then, a random 512×512 square is cropped. The same square must be cropped in both input and target images. Then, the square is flipped horizontally with a 50% chance. This transformation is applied each epoch, so every time we iterate over a training image, the model sees a different square of that image. This is a *data augmentation* method, because the model "has the impression" that we have more training images than we really have. In [7], a similar technique called random *jittering* and mirroring is applied, the 256×256 images are resized to 286×286 and randomly cropped to 256×256, then, the same random flip is applied. Our data augmentation technique takes more advantage of the higher resolution our images have.

Test images are simply resized to 3072×4608. Then, both training and test images are normalized so their pixel values lie in $[-1, 1]$ instead of $[0, 255] \cap \mathbb{Z}$.

Generator loss

The loss function of the generator depends on the output of the discriminator when fed with the generated image. We will compare the discriminator output pixel-wise with an all-ones one-channel image of the same shape using the binary cross entropy, see (2.5.2). We will compute

the generator loss one image at a time (batch size 1). Since the output of the discriminator is non-bounded, the logistic function is applied to each pixel so that it can be treated as a probability. Finally, the loss shape is reduced to a scalar using the average of the pixel values.

Furthermore, we add a pixel-wise penalty term to preserve the relative position of the items in the image, since convolutional neural networks only attend to features locally. Even so, this problem is already partially solved by using a conditional GAN. We use the L_1 loss (between the generated image and the target image) rather than L_2 as L_1 encourages less blurring. See Figure 2.39 and comments above the figure in Section 2.5.3. The L_1 penalty term is multiplied by a factor $\lambda = 100$ and added to the previous one (the binary cross entropy loss).

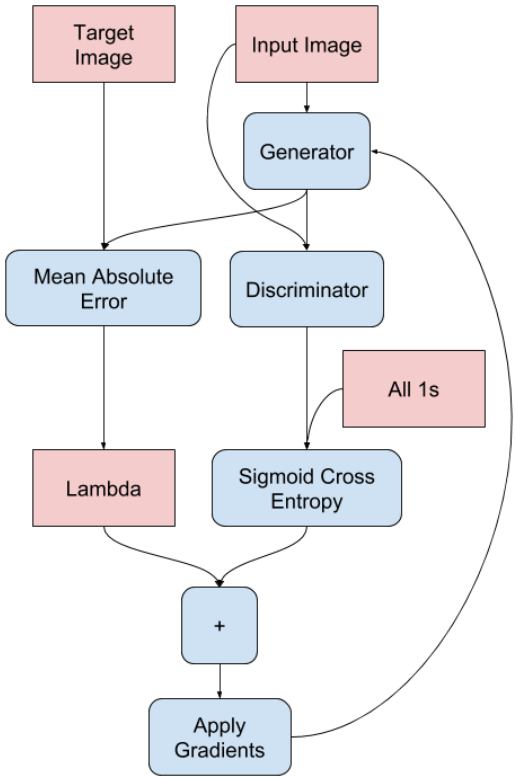


Figure 3.11: ([40]) Diagram showing the loss function of the generator. First, the generator receives the input image and generates a prediction. Next, the discriminator uses the input to tell if the prediction looks real, and outputs a matrix to which a logistic function is applied and then compared with an all-ones matrix using the binary cross entropy. On the other hand, the mean absolute error (L_1 loss) between generated image and target image is calculated, this value is multiplied by λ and both terms are added up to obtain the total loss of the generator, to which we later calculate the gradient to update the weights of the generator.

Discriminator loss

The loss function of the discriminator depends on cataloging the real image as real and the generated image as fake, it always has access to the input (dark) image. We will compare the discriminator output when fed with a generated image pixel-wise with an all-zeros one-channel image of the same shape using the binary cross entropy, and the discriminator output for the target (ground truth) image with an all-ones image. Since the discriminator output is non-bounded, logistic function is applied before the binary cross entropy. These two loses are reduced to a scalar using the average of the pixels values. The final loss of the discriminator is the sum of these two losses.

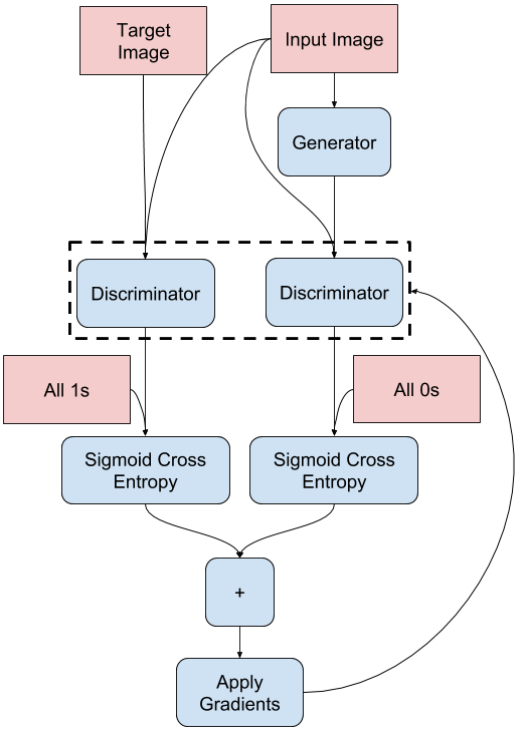


Figure 3.12: ([40]) Diagram showing the loss function of the discriminator. The generator outputs for the real and the generated images are compared pixel-wise with all-ones and all-zeros images respectively using binary cross entropy. Both losses are reduced using the average value of the pixels and added up to obtain the total loss of the discriminator, to which we later calculate the gradient to update the weights of the discriminator.

Training process

As we have commented previously, we use a batch size of 1. For each epoch, we iterate through all the images in the training dataset, one image at a time. For each image, the loss of the generator and discriminator are calculated like we showed in Figures 3.11 and 3.12, and the weights of the generator and discriminator are updated. This is done in the function `train_step`.

As an optimization algorithm, we use Adam for both the generator and the discriminator. In both cases, we set the following parameters: $\eta = 0.0002$, $\beta_1 = 0.5$, $\beta_2 = 0.999$ and $\varepsilon = 10^{-7}$. TensorFlow also provides utilities for *automatic differentiation* by representing operations as nodes of a graph, which greatly simplifies weight updating since the calculation of partial derivatives is transparent to us.

Furthermore, we place the decorator `@tf.function` over the function `train_step`, this builds a graph with that function and the functions invoked from functions in the graph (recursively), and optimizes the code. As a result, execution is accelerated.

Finally, TensorFlow provides us with utilities to save and load checkpoints. This allow us to resume the training from the last checkpoint in case an error occurs, and also to split the training process into several sessions due to the long time it takes. At each checkpoint, we need to save the parameters of both nets (generator and discriminator) as well as the parameters of both optimizers, their moving averages for the first and second moments of the gradient (v_t and m_t).

All the training process is carried through in Google Colaboratory, where we have free access to a GPU, which is crucial for the training to be accomplished in a reasonable time. However, we are not allowed to select a specific GPU model or even know what type of GPU is assigned to us.

Since the loss function depends on the discriminator and is uninformative, we lack a stopping criterion for training. Therefore, the training is carried out in sessions of 5 epochs and the quality of the generated images for the test dataset is manually checked on the test images after

each session. Besides, the training images are shuffled at the start of each session.

We decide to stop the training after 75 epochs, since we do not appreciate a significant increase in the quality of the images generated compared to the epoch 70 as we show in the examples below.

(a) Input.

(b) Ground truth.

(c) Prediction at epoch 70.

(d) Prediction at epoch 75.

Figure 3.13: Predictions for `fuji-20018-x8.jpg`, epoch 70 vs epoch 75.

(a) Input.

(b) Ground truth.

(c) Prediction at epoch 70.

(d) Prediction at epoch 75.

Figure 3.13: Predictions for `sony-20005-x6.jpg`, epoch 70 vs epoch 75.

There are other test examples where quality is increased.

(e) Input.

(f) Ground truth.

(g) Prediction at epoch 70.

(h) Prediction at epoch 75.

Figure 3.14: Predictions for `fuji-20041-x2.jpg`, epoch 70 vs epoch 75. A slight improvement can be appreciated in the color and sharpness of the leftmost parasol.

However, in other examples it decreases.

(a) Input.

(b) Ground truth.

(c) Prediction at epoch 70.

(d) Prediction at epoch 75.

Figure 3.14: Predictions for `fuji-20069-x4.jpg`, epoch 70 vs epoch 75. There is a remarkable worsening for this example.

The training process takes a long time, about 1 hour per epoch for a total of 75 hours. Besides, Google Colab limits your access to GPU when you consume it for long periods of time. For this reasons, we settle for the current results and desist from seeking an improvement in further epochs.

As recommended in [40], we set `training=True` when making predictions. This affects batch normalization layers, that uses the batch statistics instead of the accumulated statistics for the standardization of the values. Furthermore, dropout layers keep randomly dropping half of their inputs. As we can see in the following examples, a greenish hue appears in the images generated with `training=False`.

(e) `fuji-20018-x8.jpg`

(f) `sony-20005-x6.jpg`

(g) `fuji-20041-x2.jpg`

(h) `fuji-20069-x4.jpg`

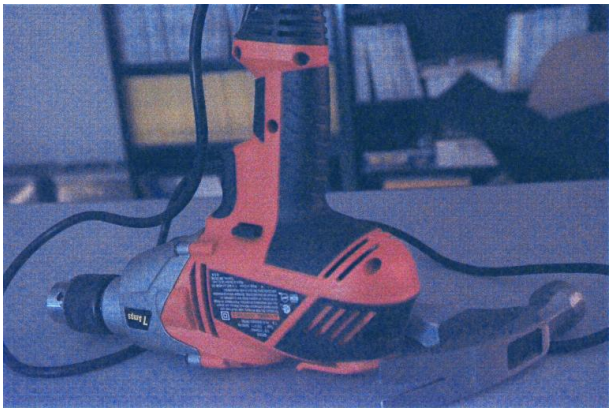
Figure 3.14: Predictions generated with `training=False`.

The tone turns blue when the input brightness is higher, maybe due to the azure color noise in the input (see Figure 1.16 and comments above).

(i) Input `fuji-20167-x2.jpg`

(j) Prediction for `fuji-20167-x2.jpg`

(k) Input `fuji-20167-x12.jpg`



(l) Prediction for `fuji-20167-x12.jpg`

Figure 3.15: Predictions generated with `training=False`.

After the training is completed, we save both models to disc.

3.4 Testing

The following notebook contains the code that generates the test predictions.

https://github.com/dcabezas98/lux-model/blob/main/p2pGAN_test-results.ipynb

Although moderate color noise appears for some inputs corresponding to photographs taken in really dark interiors (Figures 3.16, 3.17 and 3.18), the generator produces a satisfactory outputs for the majority of test inputs. We shall show some examples.

(a) Input. (b) Ground truth. (c) Generated image.

Figure 3.16: `fuji-20167-x8.jpg`

(a) Input. (b) Ground truth. (c) Generated image.

Figure 3.17: `fuji-20184-x1.jpg`

(a) Input. (b) Ground truth. (c) Generated image.

Figure 3.18: `sony-20201-x4.jpg`

(a) Input. (b) Ground truth. (c) Generated image.

Figure 3.19: `sony-20005-x6.jpg`

(a) Input. (b) Ground truth. (c) Generated image.

Figure 3.20: `sony-20177-x12.jpg`

(a) Input. (b) Ground truth. (c) Generated image.

Figure 3.21: `sony-20020-x6.jpg`

Most of the predictions for the test images are satisfactory. However, test images have a considerably high resolution, 3072×4608 ($H \times W$), and outputs lose a certain quality. This is not noticeable with a naked eye unless we zoom in on the images quite a bit. Nonetheless, the loss of quality is significant when the images have lower resolution. The following examples show the results for the images in Figures 3.16 to 3.21 when the input has lower resolution. Specifically, 1024×1536 and 2048×3072. Clearly, these outputs have poorer quality than those corresponding to 3072×4608 inputs. Line patterns appear in some of the outputs, but the quality of 2048×3072 predictions is still decent.

(a) Input. (b) Output for shape 1024×1536. (c) Output for shape 2048×3072.

Figure 3.22: `fuji-20167-x8.jpg`

(a) Input. (b) Output for shape 1024×1536. (c) Output for shape 2048×3072.

Figure 3.23: `fuji-20184-x1.jpg`

(a) Input. (b) Output for shape 1024×1536. (c) Output for shape 2048×3072.

Figure 3.24: `sony-20201-x4.jpg`

(a) Input. (b) Output for shape 1024×1536. (c) Output for shape 2048×3072.

Figure 3.25: `sony-20005-x6.jpg`

(a) Input. (b) Output for shape 1024×1536. (c) Output for shape 2048×3072.

Figure 3.26: `sony-20177-x12.jpg`

(a) Input. (b) Output for shape 1024×1536. (c) Output for shape 2048×3072.

Figure 3.27: `sony-20020-x6.jpg`

Failure cases

The generator fails to recreate images in some test examples, mainly when it comes to extremely dark inputs.

The `x1` and `x2` versions of some particularly dim images produce undesired outputs.

(a) Input. (b) Ground truth. (c) Generated image.

Figure 3.28: `fuji-20015-x2.jpg`

(a) Input. (b) Ground truth. (c) Generated image.

Figure 3.29: `fuji-20069-x2.jpg`

(a) Input. (b) Ground truth. (c) Generated image.

Figure 3.30: `fuji-20098-x1.jpg`

Sometimes, in inputs with a source of light, noisy artifacts appear below the light. This occurs mostly with very dim inputs. See Figure 3.30 and the following two examples.

(a) Input. (b) Ground truth. (c) Generated image.

Figure 3.31: `sony-20007-x2.jpg` (cropped)

(a) Input. (b) Ground truth. (c) Generated image.

Figure 3.32: `sony-20115-x4.jpg` (cropped)

In other occasions, `x1` and `x2` versions of extremely dark images, a very strong color noise appears out of nowhere.

(a) Input. (b) Ground truth. (c) Generated image.

Figure 3.33: `sony-20005-x1.jpg`

(a) Input. (b) Ground truth. (c) Generated image.

Figure 3.34: `fuji-20091-x1.jpg`

When the input presents a strong color noise (see Figure 1.16 and comments above), this is increased by the generator when recreating the image.

(a) Input. (b) Ground truth. (c) Generated image.

Figure 3.35: `fuji-20169-x8.jpg`

(a) Input. (b) Ground truth. (c) Generated image.

Figure 3.36: `sony-20210-x12.jpg`

Checkerboard artifacts (see Section 2.5.4) appear in some outputs, normally in dark areas of the images. The generator learns to avoid checkerboard patterns for some inputs.

(a) Epoch 60. (b) Epoch 75.

Figure 3.37: Predictions generated for `fuji-20018-x4.jpg`, the outputs have been cropped.

In other cases, checkerboard artifacts keep showing up during the whole training process.

(a) Epoch 60. (b) Epoch 75.

Figure 3.38: Predictions generated for `fuji-20091-x4.jpg`, the outputs have been cropped.

Finally, the loss of quality is severe when the input image has low resolution.

Figure 3.38: Predictions (right) generated for the images in Figures 3.16 to 3.21. This time the inputs (left) have resolution 512×768 pixels, and a severe blur is noticeable.

Performance on lossless formats

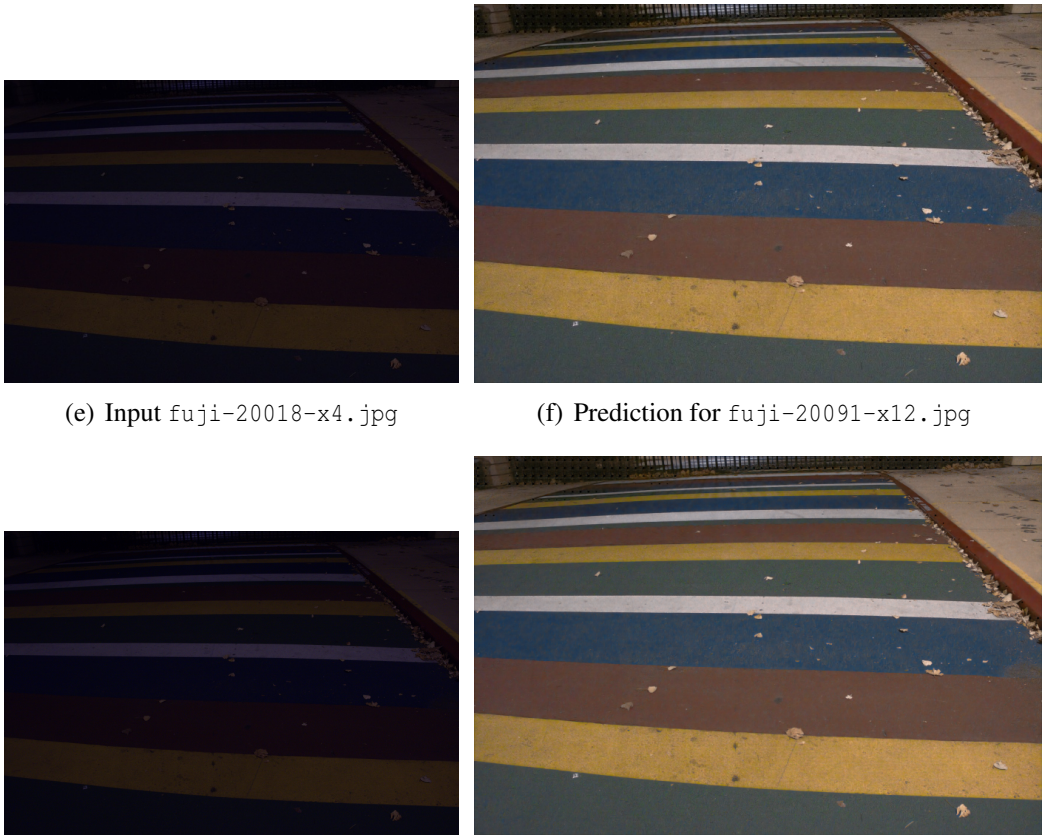
The function `decode_jpeg` from TensorFlow is capable of decoding PNG images ignoring the alpha channel, this allows our model to process PNG images without any extra effort. However, the output image is set to max opacity regardless of the initial transparency level.

The performance of our model on PNG images is similar to that already shown on JPEG images for most cases.

(a) Input `fuji-20018-x4.jpg` (b) Prediction for `fuji-20018-x4.jpg`

(c) Input `fuji-20018-x4.png` (d) Prediction for `fuji-20018-x4.png`

Figure 3.38: The PNG and the JPEG images produce outputs with similar quality.



(e) Input `fuji-20018-x4.jpg` (f) Prediction for `fuji-20091-x12.jpg`

(g) Input `fuji-20018-x4.png` (h) Prediction for `fuji-20091-x12.png`

Figure 3.39: The PNG and the JPEG images produce outputs with similar quality.

Furthermore, PNG test examples produce fewer failure cases than JPEG test examples, or at least they are milder. Probably because postprocessing raw images into PNG format leads to less loss of SNR.

(a) Input `fuji-20018-x4.jpg`

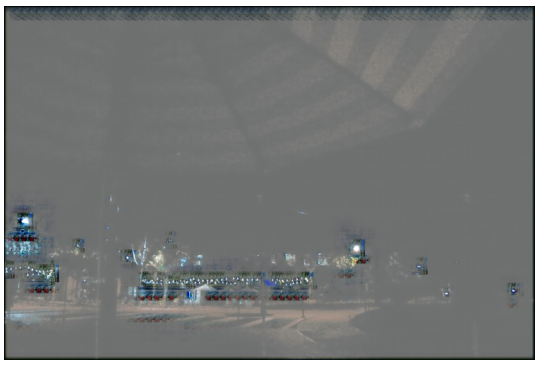
(b) Prediction for `fuji-20093-x1.jpg`

(c) Input `fuji-20018-x4.png`

(d) Prediction for `fuji-20093-x1.png`

Figure 3.40: The PNG input results in an output with more quality.

(a) Input `fuji-20018-x4.jpg`



(b) Prediction for `fuji-20098-x1.jpg`

(c) Input `fuji-20018-x4.png`

(d) Prediction for `fuji-20098-x1.png`

Figure 3.41: The PNG input results in an output with more quality.

On the other hand, TIFF images cannot be read with that function. The fastest and simplest approach to solve this is by converting the images into RGB formats before applying the model. Therefore, our model will not provide support for this image format.

3.4.1 Comparison with histogram equalization

The equalized images in this section and the color examples in Section 1.1 (Figures 1.10 and 1.11) are produced with the `createCLAHE` function provided by OpenCV. The default parameters are used: clip limit $= 40$ and the images are divided in $8 \times 8 = 64$ tiles. The notebook used to equalize the images can be found here:

https://github.com/dcabezas98/lux-model/blob/main/histEq.ipynb

Since Histogram Equalization is a much simple a faster approach to correct the luminosity of images, we cannot allow our model to be overtaken by it in quality of the generated images.

Fortunately, our generative model surpasses this technique for both representations (HSL and HSV). As we can see in the following examples, both histogram equalization techniques amplify the color noise present in most of the images, while the GAN generator intends to amend it.

(a) Input.

(b) Ground truth.

(c) GAN prediction.

(d) HSL equalization.

(e) HSV equalization.

Figure 3.42: `fuji-20167-x8.jpg`

(a) Input.

(b) Ground truth.

(c) GAN prediction.

(d) HSL equalization.

(e) HSV equalization.

Figure 3.42: `fuji-20184-x1.jpg`

(f) Input.

(g) Ground truth.

(h) GAN prediction.

(i) HSL equalization.

(j) HSV equalization.

Figure 3.43: `sony-20201-x4.jpg`

(a) Input.

(b) Ground truth.

(c) GAN prediction.

(d) HSL equalization.

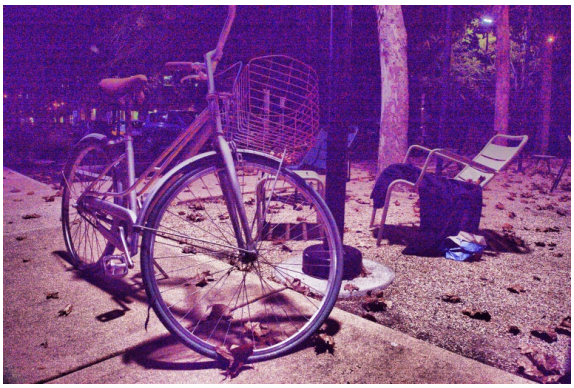
(e) HSV equalization.

Figure 3.43: `sony-20005-x6.jpg`

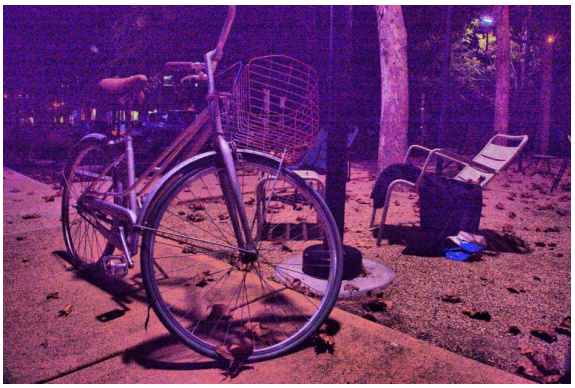
(f) Input.

(g) Ground truth.

(h) GAN prediction.



(i) HSL equalization.



(j) HSV equalization.

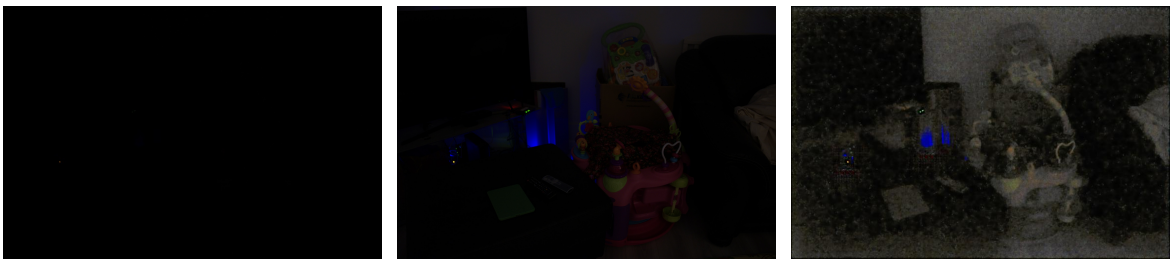
Figure 3.44: `sony-20177-x12.jpg`

(a) Input. (b) Ground truth. (c) GAN prediction.

(d) HSL equalization. (e) HSV equalization.

Figure 3.44: `sony-20020-x6.jpg`

Furthermore, both equalization techniques fail in most examples where GAN fails. Hence, we might think that the input images are so dark that they do not provide enough information to recreate the ground truths.



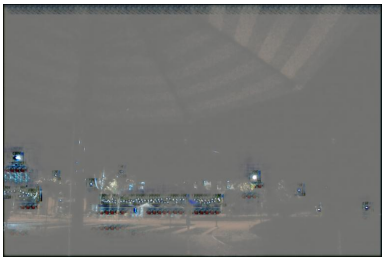
(f) Input. (g) Ground truth. (h) GAN prediction.

(i) HSL equalization. (j) HSV equalization.

Figure 3.45: `fuji-20015-x2.jpg`

(a) Input.

(b) Ground truth.



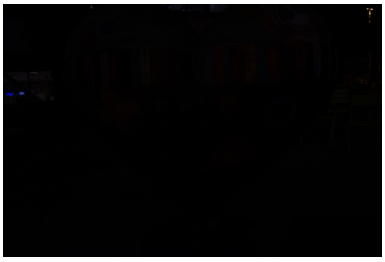
(c) GAN prediction.

(d) HSL equalization.

(e) HSV equalization.

Figure 3.46: `fuji-20098-x1.jpg`

All equalized images have strong color noise, which is greater when using the HSV representation. However, except for the cases described above where the input was too dark, histogram equalization generate images with sharp edges (even with low resolution inputs) and no artifacts. Generally, this does not compensate for the color noise, but the are few test examples where the GAN fails and histogram equalization achieves a minimally decent output.



(a) Input.

(b) Ground truth.

(c) GAN prediction.

(d) HSL equalization.

(e) HSV equalization.

Figure 3.47: `fuji-20069-x2.jpg`

All in all, there is no doubt that the images produced by our generative model are far superior than the equalized images, specially when the resolution of the images is high. Therefore, we shall use the GAN generator for our purpose.

3.5 Getting the model ready

Once we are satisfied with the performance of our model, we save it to a file. The model will be loaded and used in our application.

We still have to solve the shape problem. Right now, our generator only works correctly on images whose height and width are divisible by 256. Normally, images with different shapes would simply have their shapes reduced in the output. For instance, a 1150×1700 input would produce a $1024 \times 1536 = 4 \cdot 256 \times 6 \cdot 256$ output, since the decoder part creates a 256×256 squares for each pixel in its output (the "bottleneck" of the generator). However, the concatenations in skip connections require the shapes to match exactly, which produces an error. Therefore, we need to find a way to process images with arbitrary shapes.

The notebook used to test the two following strategies can be found here:

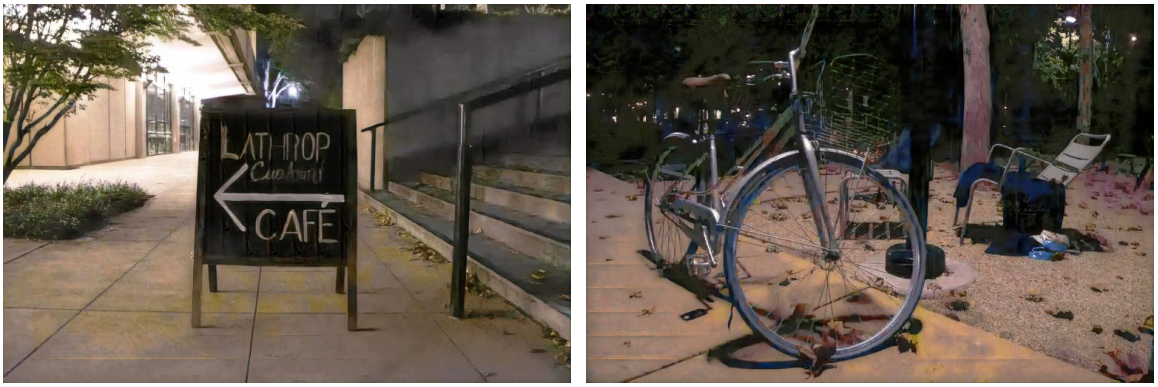
https://github.com/dcabezas98/lux-model/blob/main/border.ipynb

We try a first approach, applying zero padding to the right and bottom border until they meet a valid shape. Then, the generator creates a prediction that is cropped into the initial shape. However, this approach not only produces disturbances in the border, but in the whole image.

(a) `fuji-20018-x6.jpg` (b) `sony-20177-x12.jpg`

Figure 3.48: Predictions for inputs with shape 900×1400.

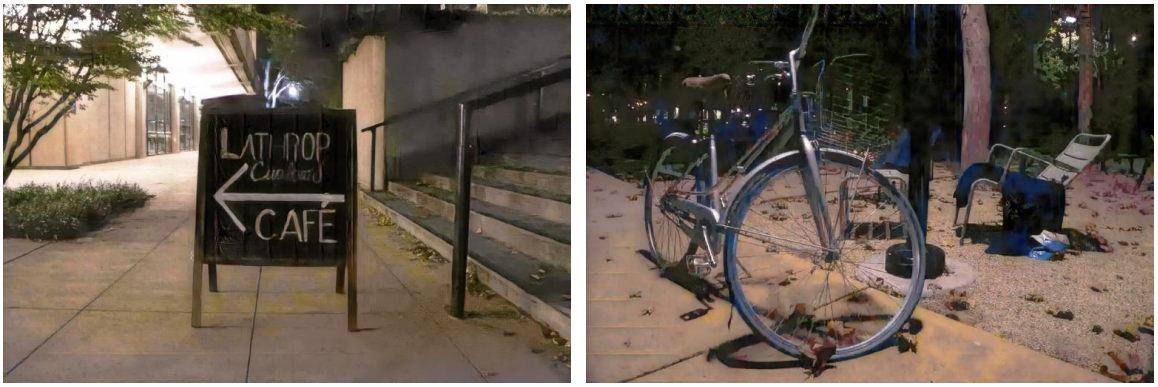
A huge difference can be appreciated with respect to the predictions for shape 1024×1536, to which zero padding does not need to be applied.



(a) `fuji-20018-x6.jpg` (b) `sony-20177-x12.jpg`

Figure 3.49: Predictions for inputs with shape 1024×1536.

We try a more complex strategy. This time, we will replicate the right and bottom borders (the right column and the bottom row) of the image to achieve the desired shape before applying the generator.



(a) `fuji-20018-x6.jpg` (b) `sony-20177-x12.jpg`

Figure 3.50: Predictions for inputs with shape 900×1400.

As we can see, the images produced by this approach compare to those in Figure 3.48. We shall apply this method to solve the problem of the shape. Our generative model is finally ready.

The following script contains a function that loads the generative model and defines a function that adjusts the shape of a given image, applies the model to it and removes the padding. The function receives the image path as input and the image is loaded from disk, the output image is returned directly.

https://github.com/dcabezas98/lux/blob/main/lux/model.py

CHAPTER 4

Web application development

We shall devote this chapter to the production of an easy-to-use interface that grants non-expert users access to the generative model that we created previously. We will build a web application, another other option would be to build a mobile app, but a web app can potentially work on a mobile device. We shall start with a basic requirements analysis and design, a discussion about what framework to employ for the development, and finally the coding, testing and deployment of the application.

Besides, we shall also describe our first alternative evaluated, whose failure made us opt for the Flask web app we present later.

4.1 Requirements

At the very least, we want our web app to provide the following functionality:

- A simple image slider that allows the user to navigate through some test examples (Figures 3.16 to 3.21), so the user can appreciate the effect of the model on dark images.
- A page with the main functionality, with a form that allows the user to upload an image. When an image is uploaded, it is fed to the model and the output is displayed to the user along with a download link.
- An "About" page that displays the name of the author and the GitHub projects containing the web application and the generative model.

The following features are also desirable:

- A basic "Home" page that welcomes the user and provides a little description of the app.
- A navigation bar that allows the user to easily switch from one page to another.
- A Dark Mode option, toggled by a button in the navigation bar.

Since the purposes of this project are purely academic, we do not face strict non-functional requirements. Nonetheless, we shall make the following considerations while developing our application.

- Usability. The app must be intuitive and the steps must be detailed enough so the user knows how to proceed at all times.
- Overload prevention. We must limit the number and size of the images that the model processes so that we do not exceed the resource limits. We also need to allocate enough resources at the time of deployment so that the application can work correctly.

RAM is the most consumed resource by the model, the use of skip connections (see Figure 3.3 and the paragraph above it) requires storing the image data at each layer of the encoder, which consumes a high amount of RAM.

4.2 Design

We shall try to accomplish all of the above requirements while keeping the application as simple as possible. We therefore propose the following design.

4.2.1 Logical viewpoint

We will start by exposing the logical view of our whole system. It represents the different modules our system is composed of and how they depend on (or interact with) each other.

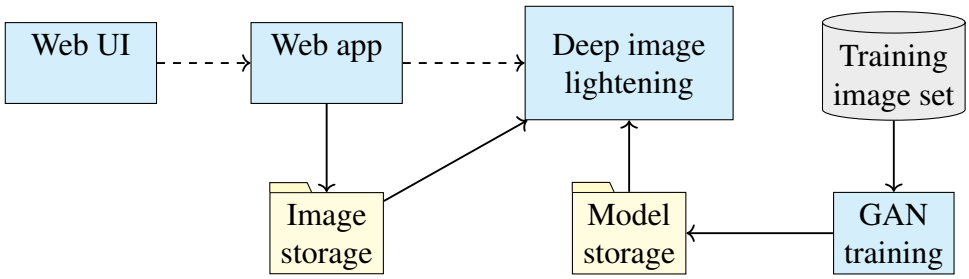


Figure 4.1: Logical viewpoint of the whole project.

The **user interface** is rendered by the **web app**, that stores the submitted images and invokes the **deep image lightening**, which loads the model from the **model storage** (not necessarily for every input submitted, ideally it should be loaded only once) and the image from **image storage** and applies the model to the image.

For the production of the trained model, the **GAN training** was necessary, which loaded the **training image set**. Then, the model was stored.

Let us remark that the **GAN training** and **training image set** are necessary modules for our system, but they will not be part of the deployed application, since they are not necessary once the generative model is trained and stored.

4.2.2 Image processing

Now, we shall describe how the main functionality will be carried out.

When the input image is uploaded, it will be temporarily stored, a random ID will be appended to the name to avoid collisions. Next, the path will be fed to the function described at the end of Section 3.5, which will read the image, add the padding, process the image using the generative model and remove the padding. Then, the image file is deleted from disk, and the output will be displayed in the page.

Moreover, the following precaution is taken in order to avoid RAM overflow. The maximum number of pixels in the input image (padding included) is limited to $2048 \cdot 3072$, the maximum size between the test images, which is quite large. If a larger image is submitted, it is resized using the following rule:

$$h \leftarrow \left\lfloor h \sqrt{\frac{2048 \cdot 3072}{(h+hpad)(w+wpad)}} \right\rfloor; \qquad w \leftarrow \left\lfloor w \sqrt{\frac{2048 \cdot 3072}{(h+hpad)(w+wpad)}} \right\rfloor,$$

where h and w are the height and width of the input image respectively, and $h+hpad$ and $w+wpad$ are the lowest multiples of 256 satisfying $h \leq h+hpad$ and $w \leq w+wpad$. This way, we prevent excessively large images from being used as input for our model, which could overload the RAM and cause the application to crash. The user is warned if its input image is resized.

4.2.3 UI design

For the user interface, we shall start from the following scheme.

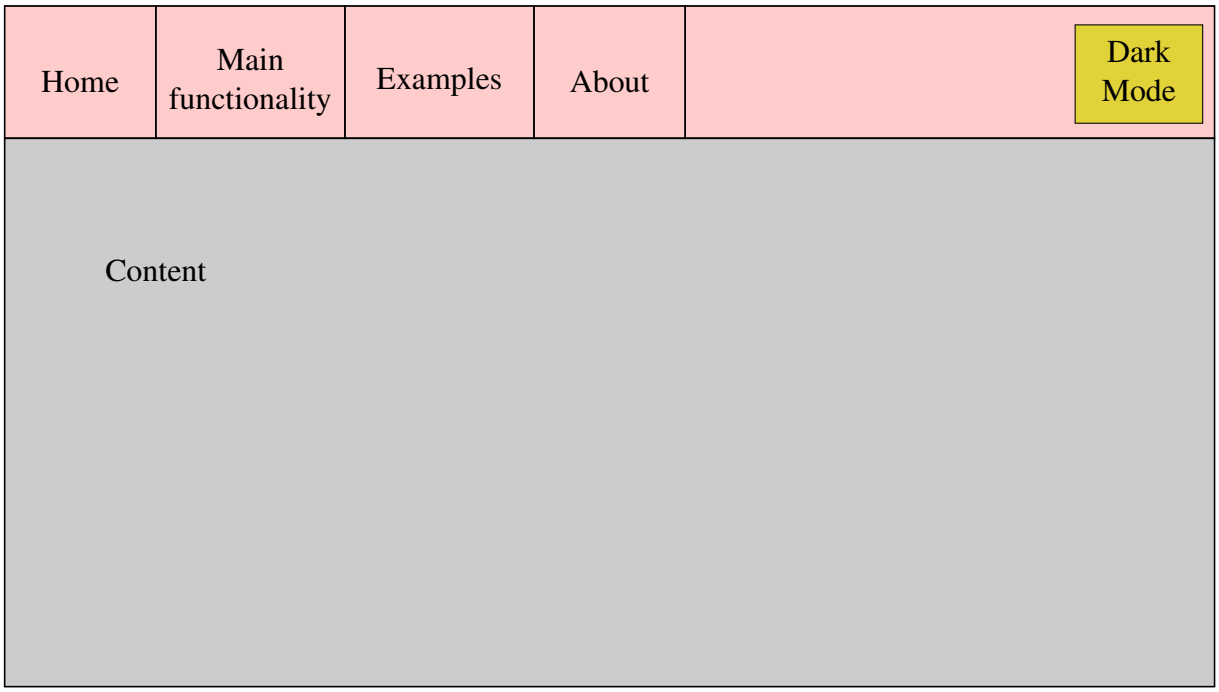
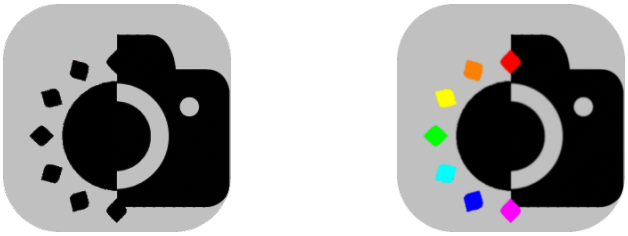


Figure 4.2: Scheme of the web page. The navigation bar will be displayed horizontally at the top, while the content of each page will be displayed in the gray area. At the leftmost part of the navigation bar, links to the different pages of the app are located. The button that toggles the dark mode can be found at the rightmost part of the navigation bar.

We shall name our application “LUX” (from the Latin word for “light”), and we will display this name along with a logo designed by ourselves in the link for the Home page.



(a) Black-and-white version.

(b) Color version.

Figure 4.3: Logo of our application, LUX.

The content displayed will vary depending on the page. The Home and About pages will display simple text. The main functionality page (we shall name it “Light Up!”) will contain a form to upload an input image. After the input is submitted, the page will show a scaled version of the output along with a download link.

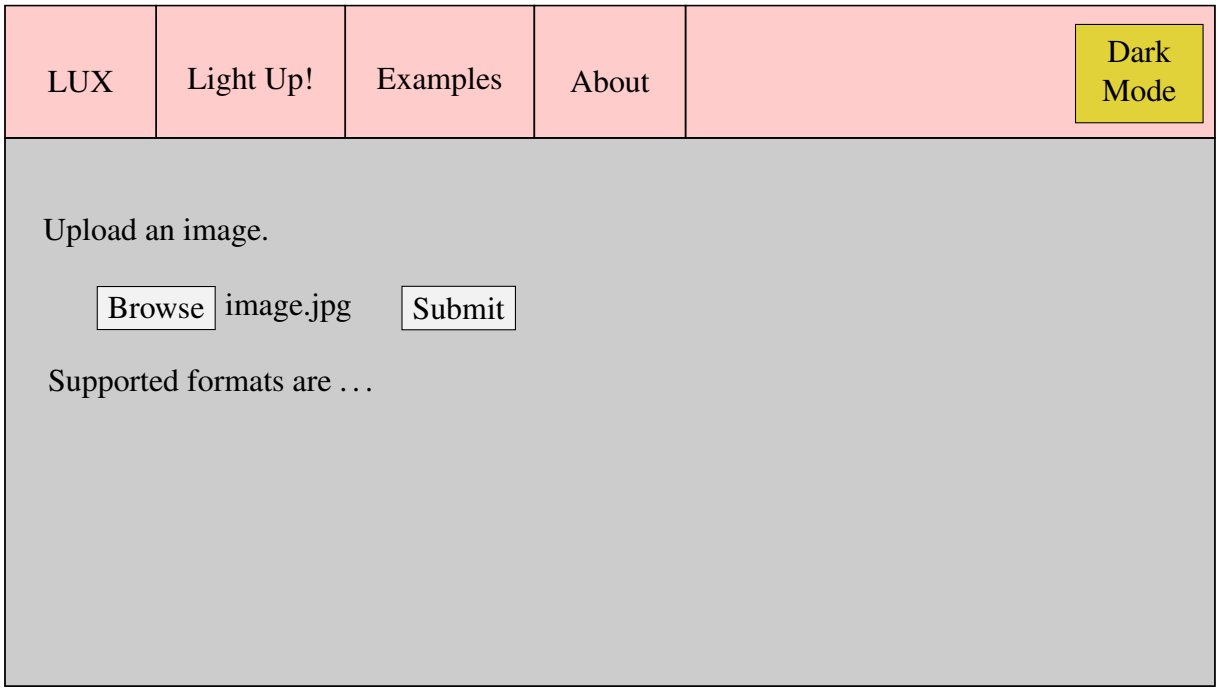


Figure 4.4: Scheme of the Light Up! page for submitting an input.

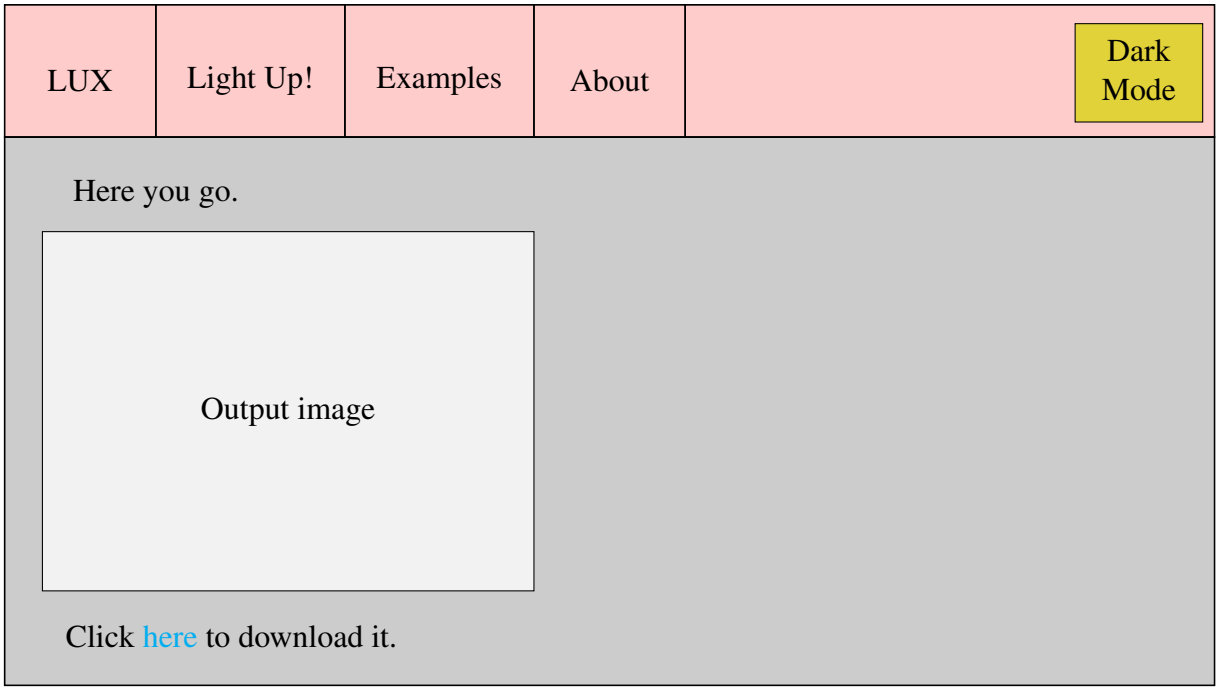


Figure 4.5: Scheme of the Light Up! page for displaying the output.

The Examples page will show the user some test examples (Figures 3.16 to 3.21). Two buttons will allow the user to switch to the next or previous example.

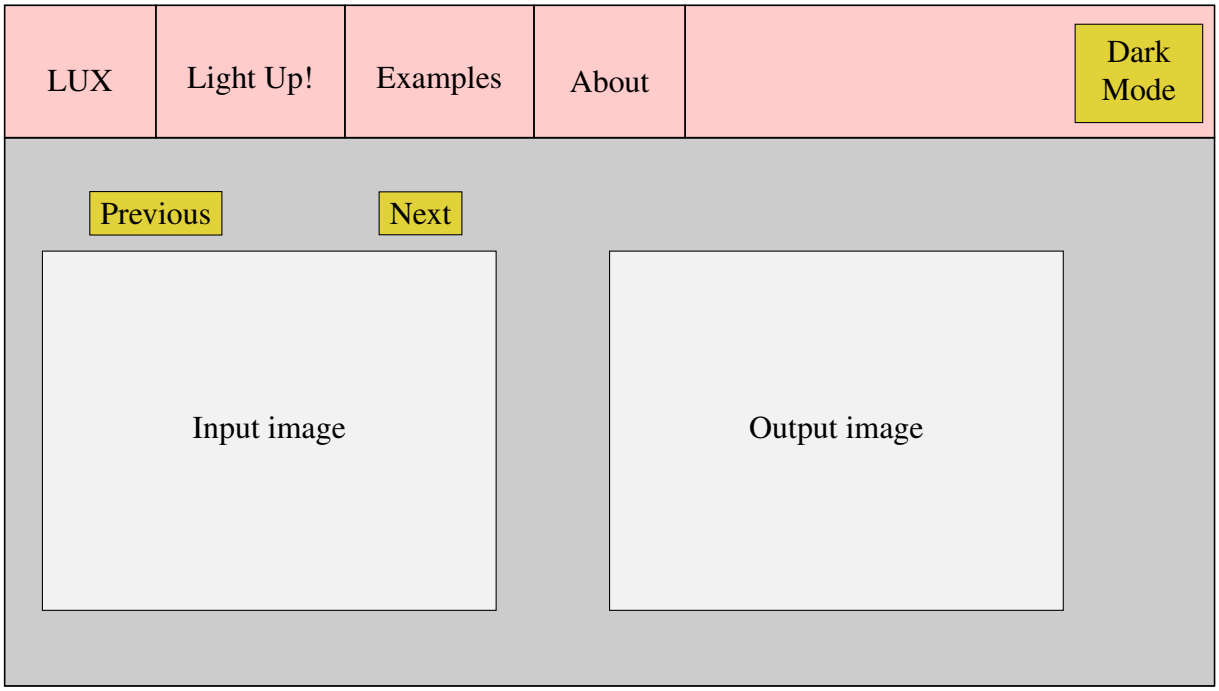


Figure 4.6: Scheme of the Examples page.

4.3 Implementation

We shall implement the application in Python3, which is a popular programming language for these types of tasks, and the one we are most comfortable with.

The employment of a framework is highly recommendable, the most popular alternatives are Flask and Django. We shall use Flask because it is simpler and more lightweight (it is, in fact, a micro-framework) than Django, which provides many more features that we will not need.

Flask also provides simple message flashing, which allows us to send the user warnings and errors.

We shall make use of a Bootstrap template with a navigation bar at the top. Specifically, we shall take the template *Sticky footer with fixed navbar* as starting point.

All the code of the application can be found in the following repository.

https://github.com/dcabezas98/lux

The file `main.py` renders the templates of all the pages mentioned above. The function `light_up` found in this file contains the code to handle submitted images so they can be processed by the function defined in `model.py`, and also manages the produced output. This way, the code in `model.py` does not change at all, with the exception of the resizing described above, that takes place when the input image is too large.

We shall use the function `uuid4` from the Python library UUID to generate the random IDs that we append to the names of the inputs in order to avoid collisions. The library Pillow allows us to obtain the resulting image from the array that the model outputs. Then, the image is encoded in base 64 and is rendered within a template.

Result

We shall show the final appearance of the produced application, as well as its functioning.

The Home page welcomes the user, it has links to the Examples and Light Up! pages.

Figure 4.7: Home page. Dark mode is activated.

The About page displays the name of the author. It also has links to the GitHub projects of the web application and the generative model.

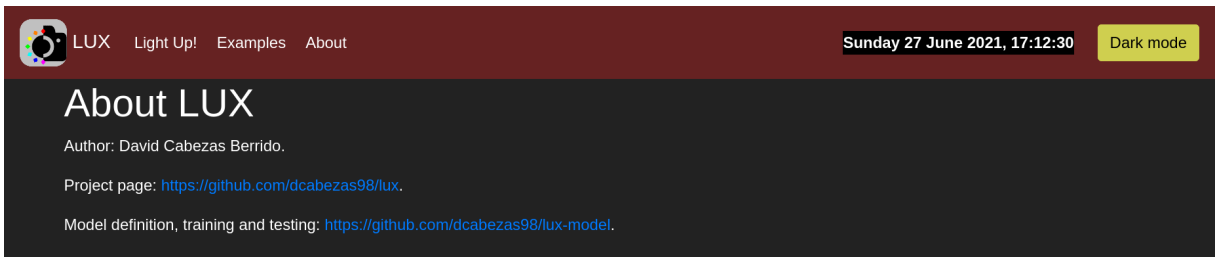


Figure 4.8: About page. Dark mode is activated.

The Examples page allows the user to navigate throughout a battery of test examples.

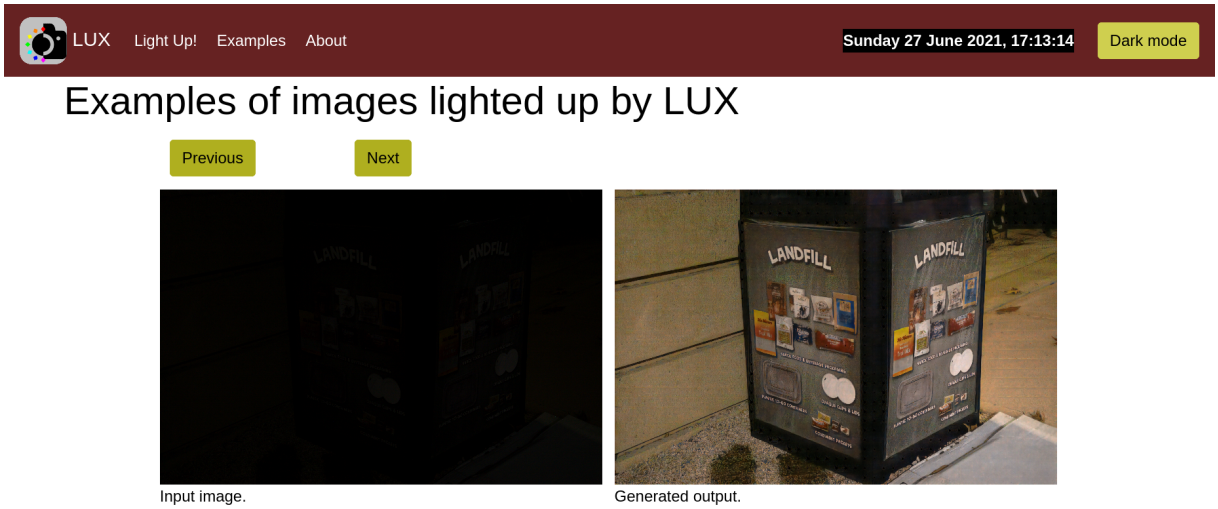


Figure 4.9: Examples page. Dark mode is deactivated.

Now we shall test the main functionality with the test image `fuji-20069-x8.jpg`. The resolution is 2000×3000 pixels (height $\times$ width), so padding and cropping will be needed.

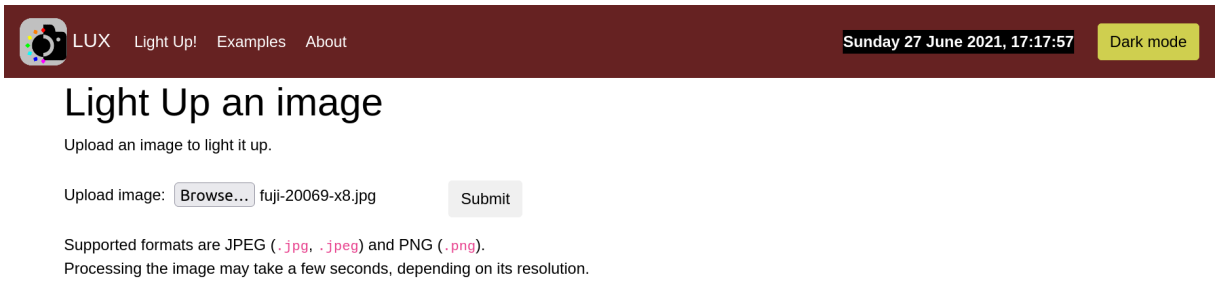


Figure 4.10: Light Up! page for submitting inputs. The input `fuji-20069-x8.jpg` has already been uploaded. Dark mode is deactivated.

When we click the Submit button, the generated output is displayed after a few seconds.

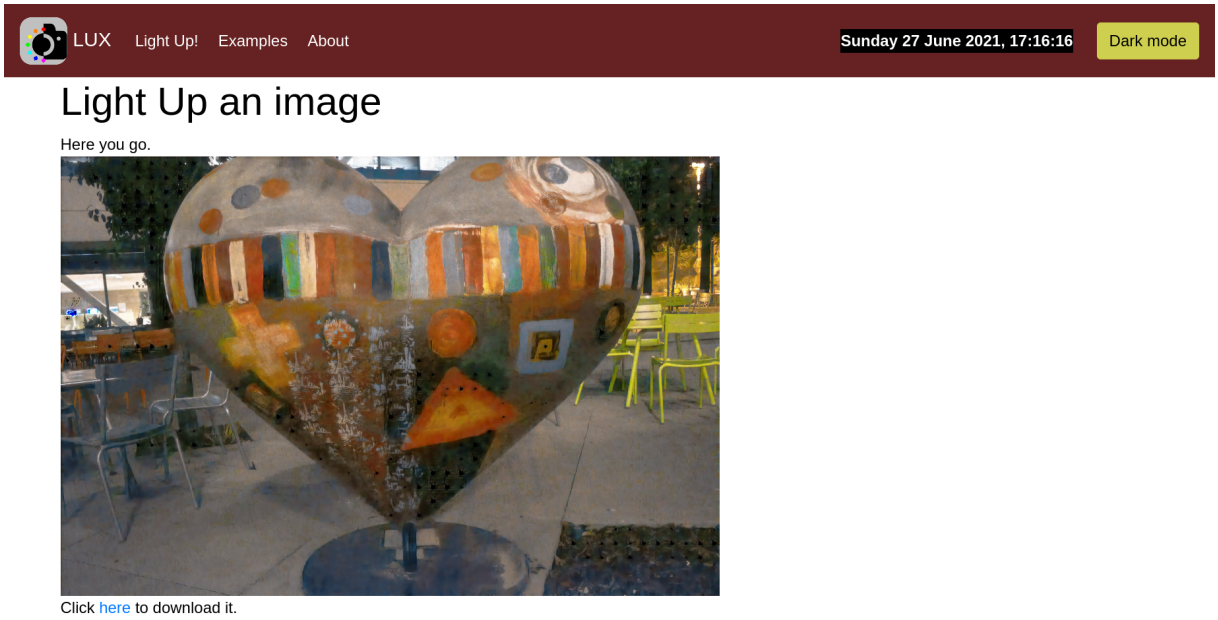


Figure 4.11: Light Up! page for displaying the generated output. Dark mode is deactivated.

(a) Input. (b) Output.

Figure 4.12: Input and generated images. Test example `fuji-20069-x8.jpg`.

As we mentioned before, we shall send errors and warnings to the user via message flashing.

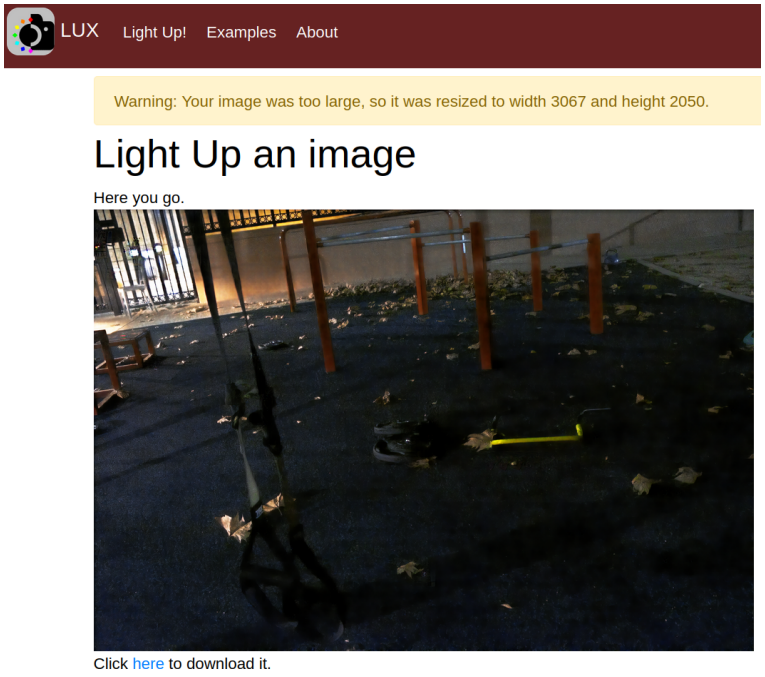


Figure 4.13: Output obtained from a 4032×6032 input. The user is notified that its image has been resized.

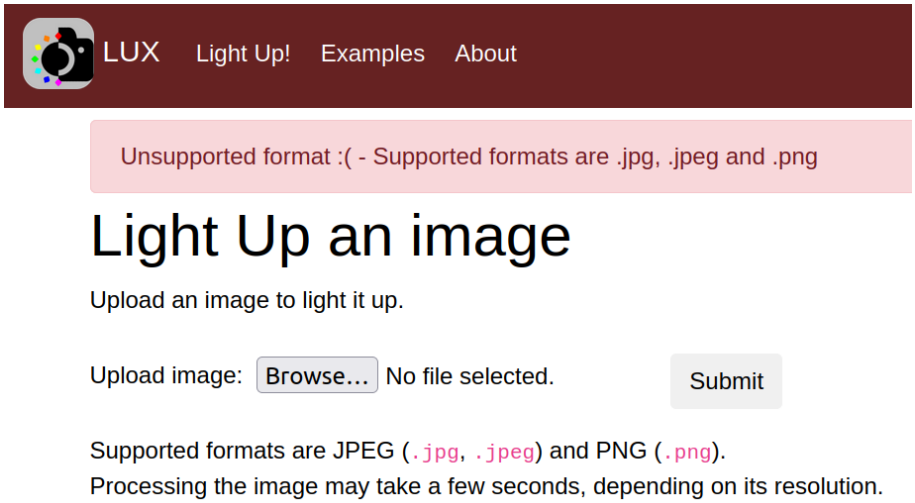


Figure 4.14: Error received when trying to submit an image with another extension. The user remains on the same page.

4.4 Deployment

We shall encapsulate our application in a Docker image so we can deploy on a cloud computing service. We would like our application to be run cost-free, so we could consider deploying it on Heroku. However, our app will consume a high amount of RAM, and Heroku's free container running only allocates 512MB. Therefore, we shall use a paid cloud computing suite with free trial such as Google Cloud. Google Cloud offers $300 in free credits to new customers, that can be spent within 3 months. We will make use of this offer and create an account on this platform.

The deployment view of our application would be the following.

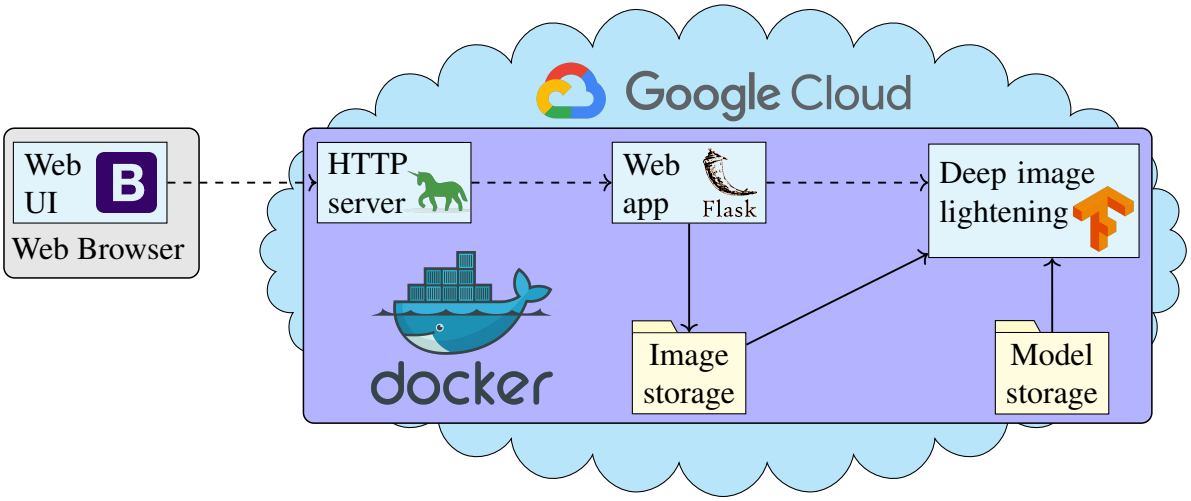


Figure 4.15: Deployment viewpoint of the whole project. Note that only the modules **GAN training** and **training image set** from Figure 4.1 are not present in this view, since they are not necessary once the model is trained and stored.

Apart from the application structure, we represent the tools employed in each module: Gunicorn for the HTTP server, Bootstrap for the user interface, Flask for the web app, TensorFlow to apply the model, Docker to encapsulate the whole application and Google Cloud to deploy it.

Let us start with the deployment. First, we sign up to Google Cloud and create a new project named “Lux Image Lightening”, with the project ID “lux-image-lightening” and “No organization” as location.

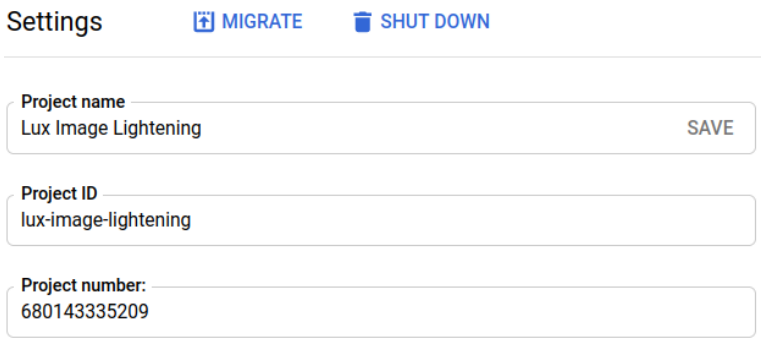


Figure 4.16: Project settings.

We follow the tutorial in [42] to containerize our application, the Dockerfile installs the required packages and starts a Gunicorn web service that listens on the port defined by the PORT environment variable. The number of current workers is set to 2 since we will allocate 2 CPUs per instance at the time of deployment. We also disable the timeouts of the workers to allow Cloud Run to handle instance scaling. The final line of this file reads as follows.

```
CMD exec gunicorn --bind :$PORT --workers 2 --threads 8 --timeout 0 main:app
```

We now use the following command to build and submit our container with Cloud Build.

```
gcloud builds submit --tag gcr.io/lux-image-lightening/lux
```

When the task is completed, the container appears in the Container Registry page of Google Cloud Platform, where we can start the deployment to Cloud Run.

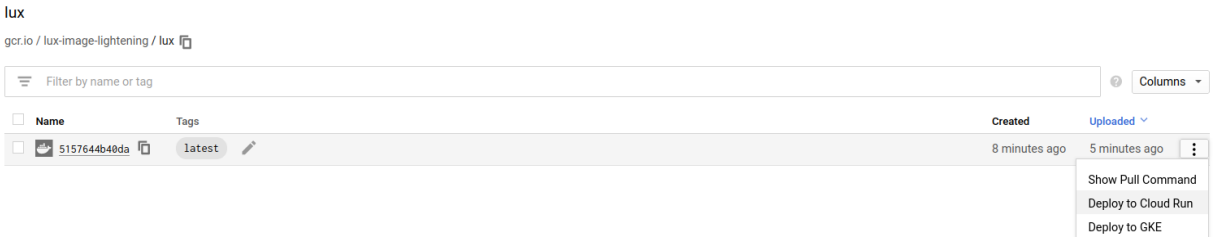


Figure 4.17: The docker image we just built and submitted.

We create a service named “lux” and locate it in Zurich. We select to deploy our newly built docker image.

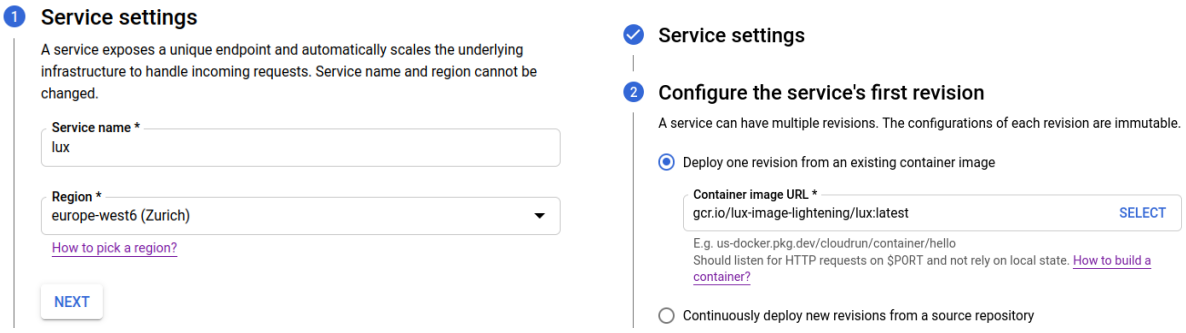


Figure 4.18: Service name, region and deployed image.

Then, we choose the listening port and allocate the resources. We will allocate 4GB of RAM memory and 2 CPUs per instance of the service. We will set the request timeout to 600 seconds (quite high, since processing big images is slow), and the maximum requests per instance to 10. The service will be dynamically instantiated according to the number of requests, we shall limit the number of instances between 0 (we do not want to spend resources when the service is not in use) and 4, since we do not want to run out of credits too soon and a high number of instances could consume much resources.

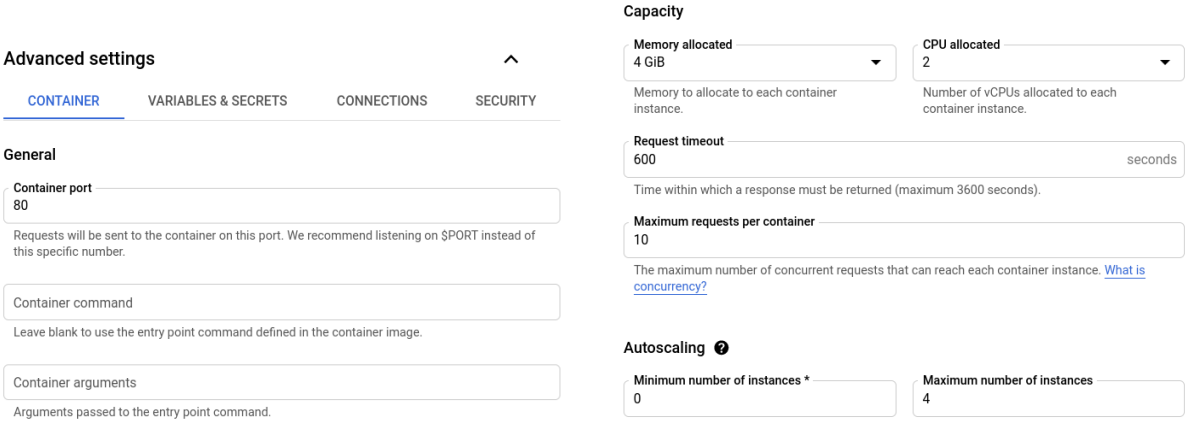


Figure 4.19: Listening port, capacity and autoscaling of the service.

Finally, we allow all traffic. We do not require authentication to invoke the service.

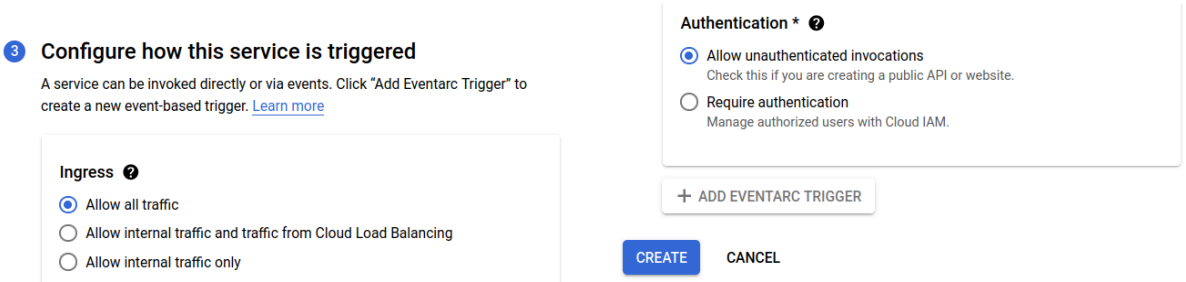


Figure 4.20: The service can be triggered by any user, no authentication is required.

When we click the create button, a URL is assigned to the service lux, where we can access the application. In this case, we get the domain https://lux-z3xqzxjs4a-oa.a.run.app.

The following folder contains the inputs of the examples so they can download them and submitted in order to test the application.

https://github.com/dcabezas98/lux/tree/main/lux/static/examples

When we submit an input image with 2000×3000 pixels (the maximum allowed once padding is added), we still get the desired output after a few seconds. However, the following error appears in the log.

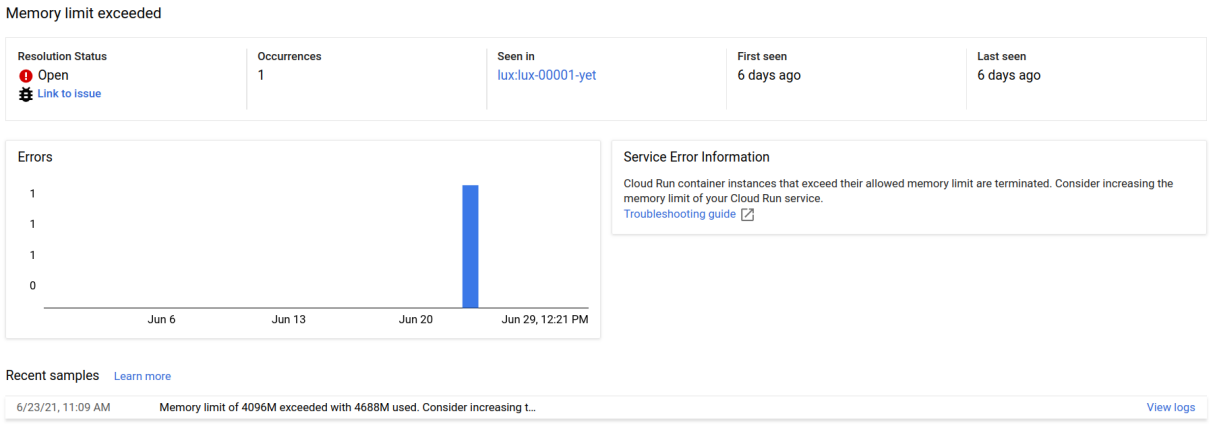


Figure 4.21: The memory limit has been exceeded. Almost 4.6GB were used.

Since we must prevent our service from exceeding the resource limit (which could lead to the service being terminated), we shall modify the capacity settings. From the service details, we select *EDIT & DEPLOY NEW REVISION*, then we perform the following modifications to the capacity.

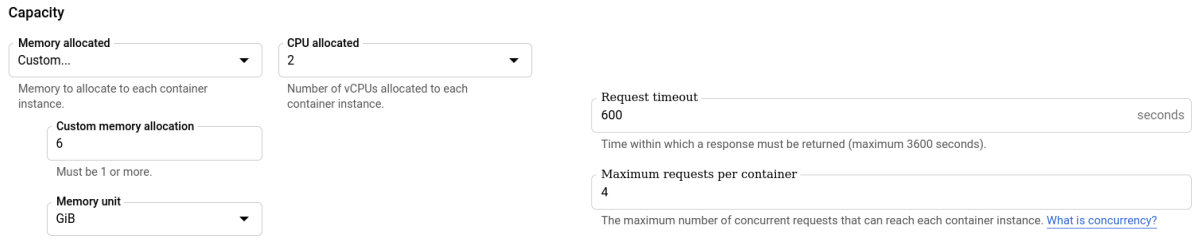


Figure 4.22: New capacity settings for the service. Set we shall allocate 6GB of RAM per instance of the service, which would be enough to handle the previous request (where 4.6GB were used) without errors. Furthermore, we limit the maximum number of concurrent requests to 4, since our app may not handle several images at the same time without exceeding the new limit. We do not increase the maximum number of instances, since we expect very low traffic.

The service could still exceed its memory limit if two or more really large images were submitted at the same time. However, we will not increase the resources or limit the traffic any further, since it is unlikely to happen and we do not want to run out of credits too soon.

Our web application, and our whole project, is finally finished and running.

How to disable the service

We may need to disable the service temporarily at some point. For example, we need to save enough credits for the project presentation. As stated in the guide [43], Cloud Run does nor

offer a direct way to make a service stop serving traffic, but we can still prevent the service from consuming resources just by deleting the member “allUsers” list in the Permissions page.

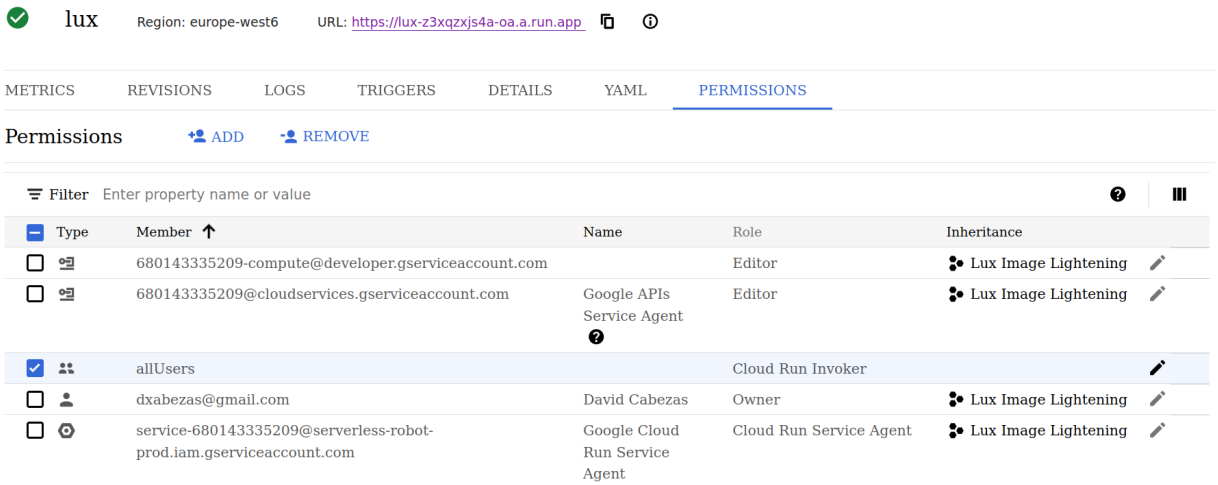


Figure 4.23: List of members. The member “allUsers” has permission for invoking the server. Thus, the server is public.

After we delete that member, the service is no longer accessible. Cloud Run instances the containers dynamically, so no container will be instanced given the impossibility to receive requests.

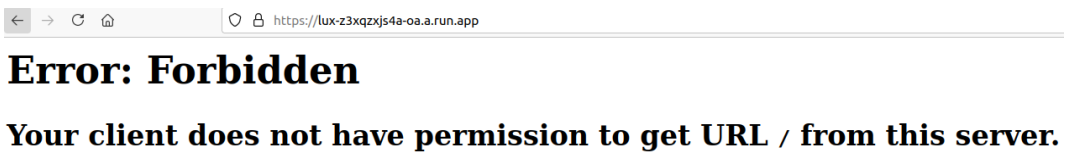


Figure 4.24: The server cannot be invoked.

Figure 4.25: We can check how many instances of the service exist at a given time by consulting the Metrics page. If no user has permission to invoke the server, the number of instances should be 0, and no resources would be allocated.

To restart the service, we go to the Triggers page and realize that the option “Allow unauthenticated invocations” has been unchecked. Checking this option back creates the member “allUsers” and grants it permissions to make the service accessible again.

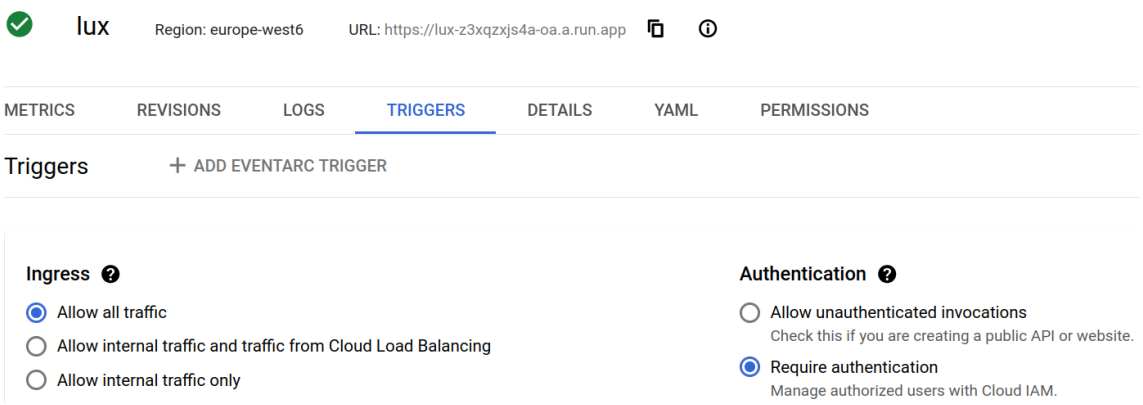


Figure 4.26: The option "Allow unauthenticated invocations" is no longer marked, checking it back will restart the service.

4.5 Failed attempt: TensorFlow.js and GitHub Pages

We shall describe our first attempt to make the generative model accessible to non-expert users.

Our first idea was to use TensorFlow.js to load the model and run it in a browser. This way, the user's machine would be the one that would run the model and no cloud server would be needed to process the images. In this case, access to the model could be provided with GitHub Pages, and the application would be running indefinitely instead of during a free trial.

However, this attempt was unsuccessful. An error related to the names of the layers occurs when loading the model JSON unless we set `strict :false`, which produces undesired outputs. In the issue https://github.com/tensorflow/tfjs/issues/755, it is stated that TensorFlow for JavaScript may have problems when loading models that have been trained in several sessions. We cannot afford training our model in a single session, since the training process takes more than 70 hours and the maximum session time allowed in Google Colaboratory is 12 hours. TensorFlow for Python does not suffer from this problem, and that is the reason we switched to Flask to produce the application.

The code for this attempt can be found in the following repository.

https://github.com/dcabezas98/imglux

The halfway-built web page (only the main functionality is implemented) is in this URL: https://dcabezas98.github.io/imglux.

When the user enters the page, it has to wait until the model is loaded.

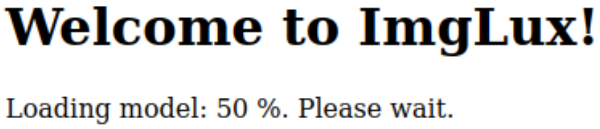


Figure 4.27: Waiting for the model to load.

Then, a form is displayed, where the user can submit the input image. Once submitted, the input image is shown under the form.

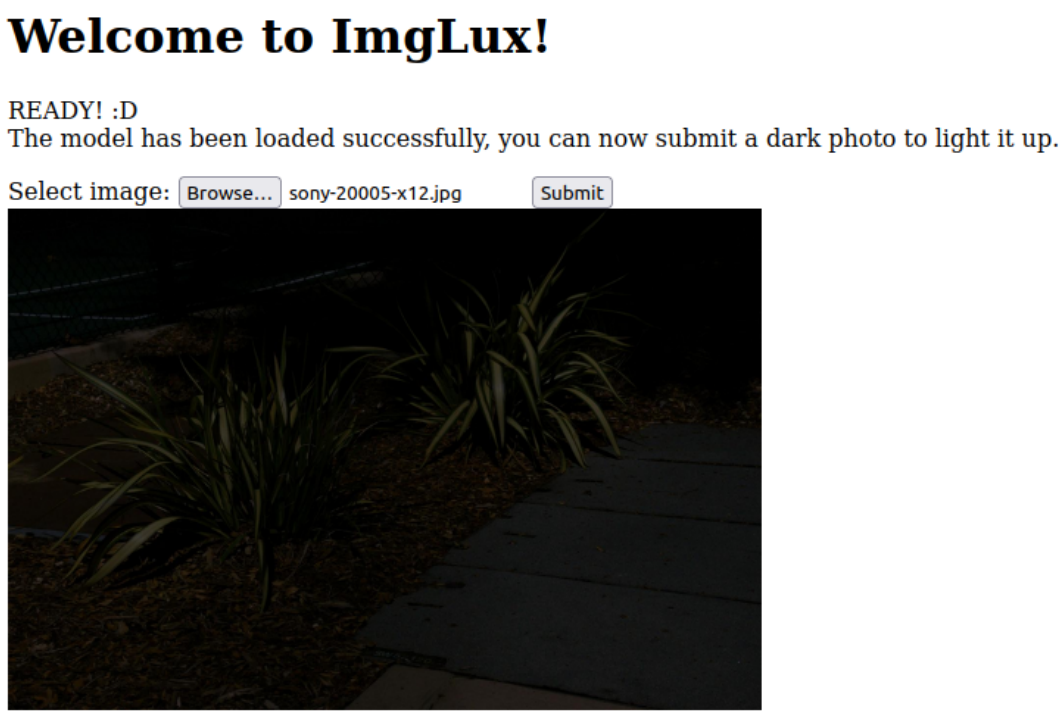


Figure 4.28: The input image has been submitted.

After a few seconds, the output is displayed. As we can see, this alternative produces undesired outputs.

Figure 4.29: The output is displayed, but it is not satisfactory.

Then, new images can be submitted and processed without the need to reload the model.

The failure of this attempt led to its dismissal. Consequently, we decided to build the web application in Python to avoid the aforementioned problem.

CHAPTER 5

Conclusions and future work

To finish the documentation of this project, we shall include an appraisal and a list of possible improvements.

5.1 Appraisal of the project

All in all, despite some problems and the lack of time, we are satisfied with the resulting software, since it accomplishes the objective that we set at start: it gives non-expert users the ability to use a deep learning model to adjust the brightness of extremely dark images.

It is true that our model does not perform correctly on some occasions, the biggest drawback is that the model distorts images when the resolution is not high enough. However, it not only surpasses our benchmark (histogram equalization) but also generates decent outputs most of the time.

The web application is simple and provides easy access to the generative model. Although it requires a large amount of memory to be allocated, all the necessary features are covered.

During the development of this project, the author has faced the typical problems related to image translation and has studied how generative deep learning techniques can be applied to solve them. To achieve this, the author has worked with one of the most popular machine learning tools, TensorFlow-Keras. Besides, the use of Docker containers and Google Cloud was necessary for the deployment. The handling of all these techniques, tools, and platforms, together with all the knowledge acquired during the development of the project, might serve as a basis for the development of future deep learning projects and even a professional career in the field of deep learning for image processing.

Finally, we shall remark that image translation problems have a high difficulty and new solutions are constantly being investigated, many of them based on generative deep learning.

5.2 Future work

Due to lack of more time and computational resources, most of the decisions we took during the project can probably be improved. To conclude this memoir, we describe possible improvements that could be incorporated to the project.

The architecture of the generative neural network we employed may not be the best for this problem. We could have done more in-depth research on image-to-image translation architectures and tried other alternatives.

Likewise, we could have performed more epochs during the training process to see if we could achieve any improvement. Nonetheless, this would require more computational resources.

With respect to the web application, we intended to structure and deploy it in a better way, but we had no time for it. The image translation would have been a microservice apart from the web application, and both the application and the image processing service would communicate through Google Cloud File Storage. This way, the web application would not have needed so much RAM to be allocated and could be hosted for free. Only the image translation service would require to allocate a large amount of resources. Right now, 6GB of RAM are allocated for each instance of the application, even if the user only navigates through the examples, which does not requires such amount of memory.

Moreover, a drag-and-drop form could also be a fast and intuitive way to submit the images to the application.

Finally, we shall remark that a web application is not the only way to make the model accessible to the public. A bot for Telegram, Twitter (or another chat application or social network) that receives an image and processes it with the model would be an interesting interface for the model. The user would only have to send the image as they normally would to a contact, and the output would be received as any image is received.

Bibliography

1. Sudhakar, S.: Histogram Equalization for Image Contrast Enhancement. Towards Data Science (2017). https://towardsdatascience.com/histogram-equalization-5d1013626e64.
2. Hanbury, A.: Constructing cylindrical coordinate colour spaces. *Pattern Recognition Letters.* **29** (4): 494–500 (2008).
3. Cybenko, G.: Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems.* **2** (4): 303–314 (1989).
4. Abu-Mostafa, Y. S., Magdon-Ismail, M. & Lin, H.-T.: Learning from data. AMLBook New York, NY, USA (2012).
5. Foster, D.: Generative deep learning: teaching machines to paint, write, compose, and play. O'Reilly Media (2019).
6. Chen, C., Chen, Q., Xu, J. & Koltun, V.: Learning to see in the dark. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 3291–3300 (2018).
7. Isola, P., Zhu, J.-Y., Zhou, T. & Efros, A. A.: Image-to-image translation with conditional adversarial networks. *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1125–1134 (2017).
8. LeCun, Y. *et al.:* Backpropagation applied to handwritten zip code recognition. *Neural computation.* **1** (4): 541–551 (1989).
9. LeCun, Y., Bottou, L., Bengio, Y. & Haffner, P.: Gradient-based learning applied to document recognition. *Proceedings of the IEEE.* **86** (11): 2278–2324 (1998).
10. Krizhevsky, A., Sutskever, I. & Hinton, G. E.: Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems.* **25:** 1097–1105 (2012).
11. Geirhos, R. *et al.:* Comparing deep neural networks against humans: object recognition when the signal gets weaker. arXiv preprint arXiv:1706.06969 (2017).
12. Redmon, J., Divvala, S., Girshick, R. & Farhadi, A.: You only look once: Unified, real-time object detection. *Proceedings of the IEEE conference on computer vision and pattern recognition*. 779–788 (2016).
13. Wang, T.-C. *et al.:* High-resolution image synthesis and semantic manipulation with conditional gans. *Proceedings of the IEEE conference on computer vision and pattern recognition*. 8798–8807 (2018).

14. Liu, Y.-L., Lai, W.-S., Yang, M.-H., Chuang, Y.-Y. & Huang, J.-B.: Learning to see through obstructions. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 14215–14224 (2020).
15. NVIDIA: Deep Learning Frameworks. NVIDIA Developer. https://developer.nvidia.com/deep-learning-frameworks.
16. Levoy, M.: Digital Photography Course. Stanford University (2010). https://graphics.stanford.edu/courses/cs178-10/schedule.html.
17. Berzal, F.: Redes Neuronales and Deep Learning. ISBN: 9781731265388. Independently published (2018).
18. Robins, M.: The Difference Between Artificial Intelligence, Machine Learning and Deep Learning. Intel (2020). https://www.intel.es/content/www/es/es/artificial-intelligence/posts/difference-between-ai-machine-learning-deep-learning.html.
19. Mitchell, T.: Machine Learning. ISBN: 9780071154673. McGraw-Hill (1997).
20. Schmidhuber, J.: Deep learning in neural networks: An overview. *Neural networks.* **61:** 85–117 (2015).
21. Han, J. & Moraga, C.: The influence of the sigmoid function parameters on the speed of backpropagation learning. *International Workshop on Artificial Neural Networks. Springer*. 195–201 (1995).
22. Brownlee, J.: A Gentle Introduction to the Rectified Linear Unit (ReLU). Machine Learning Mastery (2019). https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks.
23. Himanshu, S.: Activation Functions: Sigmoid, ReLU, Leaky ReLU and Softmax basics for Neural Networks and Deep Learning. Medium (2019). https://medium.com/@himanshuxd/activation-functions-sigmoid-relu-leaky-relu-and-softmax-basics-for-neural-networks-and-deep-8d9c70eed91e.
24. Negrini, E.: Universal Approximation Theorem, G. Cybenko. Worcester Polytechnic Institute (2019). https://users.wpi.edu/~msarkis/BrownBag/Elisa1.pdf.
25. Rudin, W.: Functional analysis. McGraw-Hill, New York (1973).
26. Pérez, J.: Análisis Funcional en Espacios de Banach. Departamento de Análisis Matemático, Universidad de Granada. https://www.ugr.es/~fjperez/textos/Analisis_Funcional_en_Espacios_de_Banach.pdf.
27. Rudin, W.: Real and Complex Analysis. ISBN: 9780070542341. McGraw-Hill Education (1987).
28. Leshno, M., Lin, V. Y., Pinkus, A. & Schocken, S.: Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural networks.* **6** (6): 861–867 (1993).
29. Kingma, D. P. & Ba, J.: Adam: A method for stochastic optimization. *3rd International Conference for Learning Representations, San Diego 2015*.
30. Ruder, S.: An overview of gradient descent optimization algorithms (2016). https://ruder.io/optimizing-gradient-descent/index.html.
31. Cohen, N.: Adam: A Method for Stochastic Optimization. The Hebrew University of Jerusalem (2015). https://moodle2.cs.huji.ac.il/nu15/pluginfile.php/316969/mod_resource/content/1/adam_pres.pdf.
32. Rumelhart, D. E., Hinton, G. E. & Williams, R. J.: Learning representations by back-propagating errors. *nature.* **323** (6088): 533–536 (1986).
33. Yamashita, R., Nishio, M., Do, R. K. G. & Togashi, K.: Convolutional neural networks: an overview and application in radiology. *Insights into imaging.* **9** (4): 611–629 (2018).
34. GNU: GIMP user manual. GNU. https://docs.gimp.org/2.6/en.

35. Dumoulin, V. & Visin, F.: A guide to convolution arithmetic for deep learning. arXiv preprint arXiv:1603.07285 (2016).
36. Kingma, D. P. & Welling, M.: Auto-encoding variational bayes. arXiv preprint arXiv: 1312.6114 (2013).
37. Goodfellow, I. J. *et al.:* Generative adversarial networks. arXiv preprint arXiv:1406.2661 (2014).
38. Mirza, M. & Osindero, S.: Conditional generative adversarial nets. arXiv preprint arXiv: 1411.1784 (2014).
39. Odena, A., Dumoulin, V. & Olah, C.: Deconvolution and Checkerboard Artifacts. Distill (2016). https://distill.pub/2016/deconv-checkerboard.
40. TensorFlow: Pix2Pix tutorial. https://www.tensorflow.org/tutorials/generative/pix2pix.
41. Ronneberger, O., Fischer, P. & Brox, T.: U-net: Convolutional networks for biomedical image segmentation. *International Conference on Medical image computing and computer-assisted intervention. Springer*. 234–241 (2015).
42. Google: Build and deploy a Python service. Google Cloud. https://cloud.google.com/run/docs/quickstarts/build-and-deploy/python.
43. Google: Managing Services: Disabling an existing service. Google Cloud. https://cloud.google.com/run/docs/managing/services#disable.