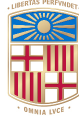


Treball final de grau

GRAU D'ENGINYERIA INFORMÀTICA

Facultat de Matemàtiques i Informàtica
Universitat de Barcelona

Automatic Image Colorization

Author: Guillem Pascual

Supervisor:	Dr. Santi Seguí
Completed in:	Departament de Matemàtiques i Informàtica

Barcelona,	June 30, 2016

Abstract

Colorizing is the act of giving color to grayscale images. A convolutional-neural-network-based method to colorize images without human interaction is presented in this project. Various frameworks, architectures, color spaces and approximations are explored to obtain the final model, capable of correctly restoring the original color of photographies without any further information than the image itself.

The principal aim of this project is to propose an idempotent architecture that could be trained with all kinds of images and yet produce good results. To demonstrate how the process works and show the obtained results, three categories of images will be used along this project: synthetic images representing numbers, landscape images and human faces.

Resum

Coloritzar consisteix en dotar de color a imatges originalment en escala de grisos. En aquest treball es presenta un mètode basat en xarxes neuronals convolucionals per coloritzar imatges sense cap interacció humana. Diferents frameworks, arquitectures, espais de color i aproximacions s'exploraran per obtenir el model final, capaç de restaurar de forma adequada el color original de fotografies, sense cap altre informació que la mateixa imatge.

L'objectiu final d'aquest projecte es presentar una arquitectura idempotent que pugui ser entrenada amb qualsevol tipus d'imatge i produir bons resultats. Per demostrar que el procés funciona i mostrar els resultats obtinguts, es faran servir tres categories diferents d'imatges: imatges sintètiques que representen números, paisatges i cares humanes.

Acknowledgements

First of all, I would like to express my gratitude to Dr. Santi Seguí. Not only for his incommensurable support and assistance throughout this research project, but also for his guidance all along this course. This project would not be the same without him.

I would also like to thank my whole family, for giving me the opportunity to study this degree and for helping me reach thus far. I am particularly grateful to Aida Montserrat, for her patience, her reviews and critiques to improve this project and for constantly encouraging me and pushing me further.

Lastly, I would also like to acknowledge all Universitat de Barcelona's professors, for teaching me the necessary concepts to become who I am, and all my friends and classmates who have accompanied me during this trajectory.

Contents

1 Introduction **1**

1.1 Context . . . 1

1.2 Motivation and Objectives . . . 1

1.3 Memory organization . . . 2

2 State of the Art **4**

3 Artificial Neural Networks **5**

3.1 What are they? . . . 5

3.2 Activation Functions . . . 8

3.2.1 Linear . . . 8

3.2.2 Binary threshold . . . 9

3.2.3 Sigmoid . . . 9

3.2.4 Hyperbolic tangent . . . 9

3.2.5 Rectified Linear Units . . . 10

3.2.6 Other activation functions . . . 10

3.3 Layers . . . 10

3.3.1 Fully Connected . . . 11

3.3.2 Dropout . . . 11

3.3.3 Convolution . . . 12

3.3.4 Pooling . . . 13

3.3.5 Batch Normalization . . . 14

3.3.6 Summation . . . 14

3.3.7 Other layers . . . 15

3.4 Learning algorithm . . . 15

4 Architecture **19**

4.1 Method 1: Convolutional and fully connected layers . . . 19

4.2 Method 2: Fully convolutional networks . . . 20

4.2.1 Finetuning . . . 21

4.2.2 Residual Networks 21
4.2.3 Color space 23
4.2.4 Final architecture 24

5 Implementation 27

5.1 Frameworks 27
5.1.1 Caffe 27
5.1.2 Tensorflow 27
5.2 Network implementation 28
5.2.1 Optimizer 28
5.2.2 Number of parameters 28
5.2.3 Tensorflow specific 29
5.2.4 Visualization 31
5.3 Online website 34
5.3.1 Backend 35
5.3.2 Frontend 36
5.3.3 Communication 36

6 Results 38

6.1 Measure of quality 38
6.2 Datasets 39
6.2.1 Synthetic data 39
6.2.2 Landscapes 41
6.2.3 Faces 46

7 Conclusions 51

7.1 Research conclusions 51
7.2 Future work 52

References 54

1 Introduction

1.1 Context

The main aim of this project is to contribute to the research of the artificial networks field, and specifically to convolutional neural networks. It will be focused on applications of this machine learning sub-field, making special emphasis on its capacity to learn by themselves.

Image colorization, a technique based on coloring an image which is only available in grayscale, has existed from long before. However, most works until recently had to be done either by using Photoshop or such programs or even by completely painting the image by hand. That means the user criteria and experience came to the scene, being perhaps the most important source of information. This process, when done manually, might take up to hours or even days to accomplish, while the solution we seek for should only delay a few milliseconds until it produced a colored image.

Artificial neural networks are perfect for this task. Their training time might be high, but then testing or producing results is done in a matter of milliseconds. More important even, they have the capacity to learn all alone. All the subfields of artificial neural networks are quickly evolving, within a span of fewer than 10 years we have seen innumerable new techniques that overwhelm the already existing. Convolutional neural networks, the main architecture used in this project, are no exception [1].

The *Treball de Fi de Grau* (TFG) makes a particular stress on what has been taught in the subject ”Aprenentatge automàtic i minèria de dades”, as in it applies much of the concepts taught there, from regressions to cross-validation. It also has a relation with ”Taller de nous usos de la informàtica”, but it rather expands the knowledge taught there.

1.2 Motivation and Objectives

Neural networks are not only, as said above, constantly updating but considered to be state of the art in many research fields. The aim of this project is to further research on them and propose new cases of use along with techniques that might outperform existing methods. When this project was started no other known implementations of colorization through convolutional neural network existed, thus it was, aside from research, a new method for colorization automatization. This work should serve as scientific research, personal publication and final project.

As said, other methods exist that color images, but they require the user's attention. My method aims to be completely user agnostic, in the sense that no action will ever be required from the user.

It must also be fast, within seconds the image should be colored and made available to the user. All existing implementations now either require a huge amount of previous time to model a feature space [2] or a vast number of examples like the grayscale image we want to colorize [3, 4]. This method should require neither of those, only the input image will be used, without needing hand-crafted features nor more examples.

All results should faithfully represent the reality. For example, skies should be colored blue and grass should be green. The algorithm must be consistent with the regions, as in all the grass should be painted the same color while preserving its texture and shape.

This project also takes the user into consideration. Aside from giving easy to use methods on the Python implementation, it will provide an intuitive and easy to navigate online website. The user should be able to simply select an image or drag it into the website for it to be colorized, thus making it not only fast and autonomous but accessible without any programming knowledge.

1.3 Memory organization

The structure followed in this project goes bottom-up, it first explains the inherent concepts in artificial neural networks and expands them until all the information necessary for understanding the implemented model has been taught. It is organized as follows:

1. **State of the art:** Related work with this project. Previous works and current implementations considered to be performing best at the moment of writing this memory are introduced. A brief explanation of other methods and shallow comparisons are done.

2. **Artificial Neural Networks:** Artificial Neural Networks (ANN) have not been covered in any course of the degree in Computer Science at the Universitat de Barcelona, thus a previous research had to be done. This section will cover the background information required to develop this project. It aims at explaining what an ANN is and further proceed into explaining Convolutional Neural Networks (CNN).

   This chapter covers the basic structures of the ANN, as well as the most important types of activations and layers. Backpropagation and gradient descend, the basic methods used in deep learning to train networks, will also be explained

3. **Architecture:** This section proposes architectures which are presumed to be able to address the colorization problem. They will be discussed and detailed, explaining the reasoning behind each choice and how each decision was taken.

4. **Implementation:** While the last section introduced the architecture used, this one will center on how they were particularly implemented. It will not only explain the frameworks used, Caffe and Tensorflow, but also enter into details on which optimizer was chosen and which hyperparameters where used and how were they found.

The developed website will also be detailed in this section, explaining the used technologies: Python for the webserver and a combination of HTML/CSS/JS/AJAX for the client rendering and communication.

5. **Results:** Practical examples and discussion on the obtained results. The implemented architectures will be tested and the resulting images will be examined, both qualitatively and quantitatively, to elaborate on the network performance. It will also be noted some alternative modifications done to the network to attempt to improve it.

6. **Conclusion:** It will be written if the proposed objectives has been met or not. Cases of good and bad performance will also be analyzed and conclusions will be extracted from them.

   Some discussion on how to improve the current work and some future ideas will also be mentioned.

2 State of the Art

This section does an overview on some existing and just released methods which also address the colorization problem. They will be mentioned and briefly explained to put this project into perspective and better contextualize it.

As mentioned in the introduction, colorization is not a new problem. Some methods [2, 3, 4] already exist that attempt to solve it. They, however, require a huge previous effort from the user.

The first method to successfully remove all required user intervention, by using convolutional neural networks, appeared on January 2016 [5]. It uses a pretrained VGG Network [6], a neural network designed to classify images among 1.000 different classes using the Imagenet dataset [7], as a mean of initialization. To perform colorization it takes concepts from the Residual Networks [8], which will be explained in detail in the Architecture section. The images used to train, and also to test, are taken from this same Imagenet dataset and the results showed good performance. Its main interest is, as said, being totally unattended while producing realistic images. It has been widely explored and used, for example, to perform colorization on whole videos[1]. It even surprisingly manages to be consistent along consequent frames, which is not trivial.

Some precedents exist, for example Z. Cheng's Deep Colorization[9]. Unlike the first method, their method requires both crafting features and manually labelling the images so that a semantic segmentation is first done. Moreover, for their method to work a final post-processing has to be done to reduce the artefacts which result from the patch-based approach they take. This would not fit in the previous established objectives of not requiring user intervention and being fast, but it is still worth mentioning as the results obtained are amongst the best ones.

Some other recent methods [10, 11] produce much better results on the positive examples. On the other hand, they tend to miss-classify pixels and produce inconsistent results. Some erratic behaviour has been observed where the network attempt to paint outside of the textures and boundaries. Both, however, produce much more saturated results than the networks above, a problem we will introduce later in this document and fully explain and reason about.

This project will also use techniques considered state of the art, as it parts from [5] to construct a more complex method based on completely different datasets and explores beyond the original architecture.

[1] Colorizing Black&White Movies with Neural Networks: `https://www.youtube.com/watch?v=_MJU8VK2PI4`

3 Artificial Neural Networks

3.1 What are they?

A first, correct, definition of ANN is a computational model inspired by biological neural networks used to approximate functions that are not known a priori. They imitate the behaviour of neurons, but using a simplified model where only the electrical signal is taken into account and not the chemical processes.

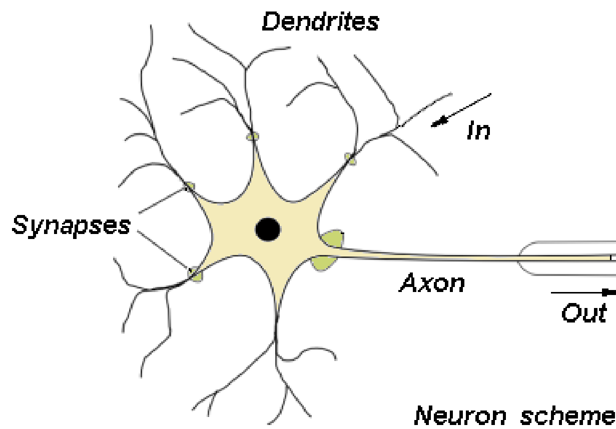


Figure 1: Simplified biological model of a neuron
Image by Virtual Laboratory of Artificial Intelligence, Agh University

Take for example the neuron in Figure 1, one single neuron will receive information from a wide number of neurons through its dendrites. Further on, the synapses will decide whether that input will stimulate or inhibit the neuron activity. For each pair of dendrite and synapse, the result will be multiplied and summed with the others. If the neuron gets activated, because the signal has reached a high enough value, it will send a signal through its axon. Otherwise, it will remain silent (on the electrical perspective, it will send a 0V signal, whether elsewhere it would send a positive or negative voltage).

This biological simplification can easily be converted into a mathematical model, see Figure 2. The dendrites are now the input variable $\vec{x}$, which gets multiplied by a set of weights $\vec{w}$, the synapses. Two more concepts are also introduced, a function which collects all the products (which usually is the summation, but could also be the product), called $f(x)$ and the activation function $g(x)$. The whole operation could now be described in terms of:

$$\vec{y} = g(f(w \cdot x)) \tag{1}$$

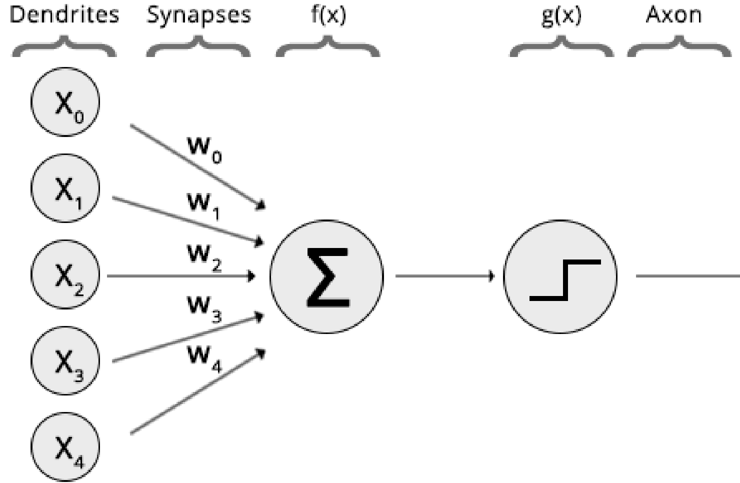


Figure 2: Mathematical model of the biological neuron

Figure 2 also illustrates the simplest and first neuron that was implemented, called the McCulloch-Pitts neuron [12]. From Equation 1, and considering this neuron uses the binary threshold as activation, we have:

$$g(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases} \qquad (2)$$
$$f(x) = \sum x$$

The McCulloh-Pitts neuron, all by itself, is capable of implementing simple lineal separable problems, such as the AND and OR functions, but fails to fit any non-lineally separable model. Such limitation, as will be seen, can be overcame by building larger networks (Figure 3) and appropriately choosing its layers and activation functions. A layer, indeed, is nothing else but a set of neurons put together, each with its connection (weight) to one or more neurons.

From this we can derive a more specialized definition, a neuronal network must:

1. Contain a set of adaptive weights which can be tuned by a learning algorithm (which will be further discussed in the next sections)
2. Be able to approximate non-lineal functions of their inputs, to be able to solve more complex problems

When creating an artificial neural network, we create networks of neurons. Each block of those networks is called a Layer, and depending on how we connect it with the previous

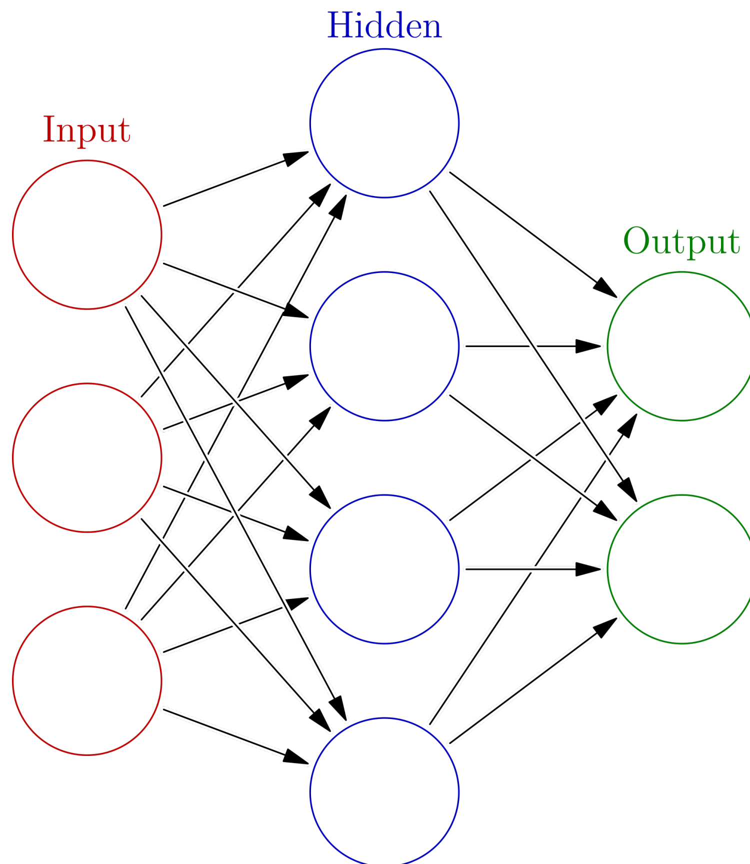


Figure 3: Artificial Neural Network: A Network of McCulloch-Pitts neurons

layer it receives one name or another. For example, on Figure 3 we would have 3 layers: the input layer, one hidden layer and the output one. A network is usually considered to be *deep* when 2 or more hidden layers are used. Taking a closer look, the hidden layer consists of 4 neurons which are fully connected by the previous layer and fully connected to the next one. The term *fully connected* is explained in the Layers sections below.

We can also differentiate amongst some basic types of networks (although much more exist) depending on how we connect them:

1. Feedforward neural network: Layers are only connected in a forward way, there are not any connections between neurons in the same layer nor it can have connections to previous layers. The network is, then, assembled into a Directed Acyclic Graph (DAG).
2. Convolutional neural network: It is a specialization of the feedforward network but using convolutional layers (explained below) and retaining the original shape of the

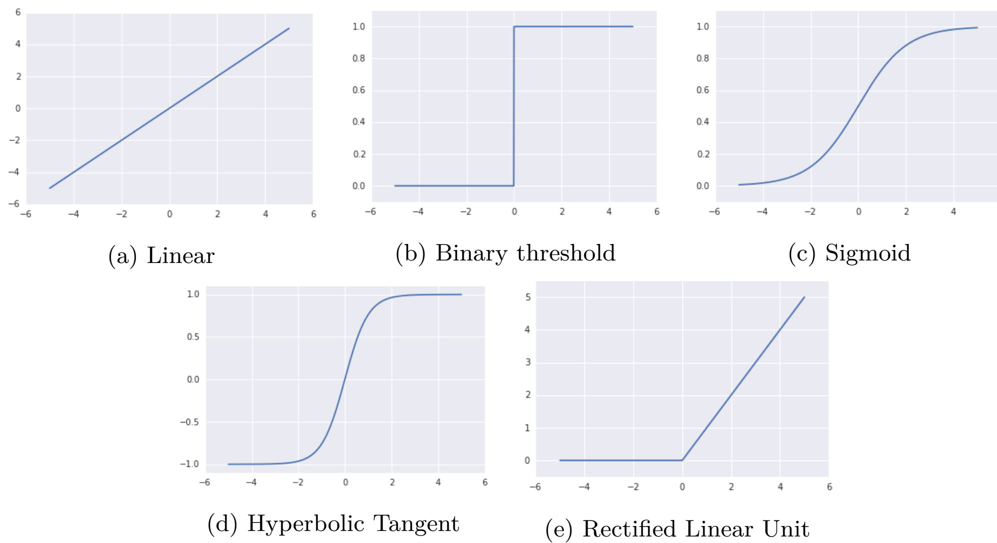
input along the process.

3. Recurrent neural network: This type of network allows connections with previous layers and amongst itself, simulating a short term memory.

3.2 Activation Functions

Activations are a fundamental part of neural networks, as for them to be able to approximate any function, we must allow the network to calculate non-linearities. This section discusses the different types we may use.

The first thing to take into account is that the learning algorithm will use the derivative of all layers and activation functions to compute the error on the prediction done by the network, which implies that a requirement for any activation function is to be differentiable.



(a) Linear (b) Binary threshold (c) Sigmoid

(d) Hyperbolic Tangent (e) Rectified Linear Unit

Figure 4: Activation functions

3.2.1 Linear

The linear activation is the most basic and, as it name implies, it does not perform any non-linearity (Figure 4a)

Its mathematical formula is:

$$g(x) = x$$

3.2.2 Binary threshold

We have already seen this activation function as part of the McCulloch-Pitts neuron (Equation 2), its graphical representation is shown in Figure 4b.

3.2.3 Sigmoid

This function maps an input value into the range $[0,1]$ (Figure 4c) by using the formula below:

$$g(x) = \frac{1}{1 + \mathrm{e}^{-x}} \tag{3}$$

The sigmoid activation has been widely used successfully due to its easy and fast derivative. Knowing that $(\mathrm{e}^x)' = x'\mathrm{e}^x$, we can simply derive it with:

$$g'(x) = g(x)(1 - g(x))$$

That means the same values obtained by the activation of the function, $g(x)$, can be reused when computing the derivative. This makes the function blazing fast by caching the activation results and reusing them later.

3.2.4 Hyperbolic tangent

This function maps an input value into the range $[-1,1]$ (Figure 4d) by using the formula below:

$$g(x) = \tanh(x) \tag{4}$$

Alike the sigmoid function, the derivative of the tanh function can reuse the values from the activation itself:

$$g'(x) = 1 - g^2(x)$$

Caching the activation values will, again, make this function faster. The main difference with the sigmoid activation relies on having a higher slope and a wider range.

3.2.5 Rectified Linear Units

Rectified Linear Units [13] (or ReLU for short) are an improved version of the linear activation which are shown to dramatically improve the performance of a network. Their most important feature is improving the gradient stability (something that will be discussed further on the learning algorithm section), which makes the network able to be deeper and learn better features.

The activation is given by (Figure 4e):

$$g(x) = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases} \quad (5)$$

3.2.6 Other activation functions

Recent papers have shown some other activation functions to have equal or better performance. They will be briefly mentioned but no expanded because they are not being used during this project:

1. Leaky ReLU [14]:

$$g(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha x, & \text{otherwise} \end{cases}$$

   Where $alpha \approx 0.01$

2. Exponential Linear Unit [15]:

$$g(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha(\mathrm{e}^x - 1), & \text{otherwise} \end{cases}$$

3. Parametric ReLU [16]:

$$g(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha x, & \text{otherwise} \end{cases}$$

   Where α is a learnable parameter

3.3 Layers

Depending on how we connect the layer to the previous ones or even on the operation they perform, we might distinguish into a wide range of layers.

3.3.1 Fully Connected

This type of layer, which can also be called Dense layer, connects each neuron of the previous layer with each of the neurons this layer itself has. Therefore, if the previous layer has n neurons and this one has m neurons, the total number of connections done are $n \cdot m$.

Although we now talk in terms of connections, a much better term would be parameters or weights. If we recall Equation 1 and Figure 2, a neuron activation was defined as the dot product (summation of multiplication) on $\vec{x}$ and a set of weights $\vec{w}$, the connections. Thus, from here on we will talk in terms of weights or parameters the neural network must learn to fit a function.

It is clear that the fully connected layer can not be huge, as its size would soon be bigger than the memory we have available. Supposing we work with *float32* data types, each weight would occupy a total of 4 bytes, which gives an allocation of $n \cdot m \cdot 4$ bytes.

One associated problem with fully connected layers is also the overfitting of the training data. That is, a neural network composed solely of fully connected layers will tend to fit too much on the training data and lose the capacity to generalize on new data, which will make new predictions fail. This gets worst the bigger the fully connected layer is (the most neurons it has). Also, stacking too much layers of this type would produce overfit or make the gradients explode (go out of the data range).

3.3.2 Dropout

On the theme of preventing overfitting, the Dropout [17] comes to action. Its working principle is straightforward, some of the neurons in the layer are set to 0, so that their activation is inhibited. Those neurons are chosen at random from a fraction of the total neurons (from 0.1 to 0.7 usually).

Deactivating some neurons implies the other ones must be accordingly scaled, so as to preserve the gradient. It could be implemented by:

$$g(x_i) = \begin{cases} x_i/(1 - ratio), & \text{if } rnd_i > ratio \\ 0, & \text{otherwise} \end{cases}$$

Where rnd_i is a random value per neuron $\in [0, 1]$ and $ratio$ is also a value $\in [0, 1]$.

This effectively reduces the overfitting by allowing the network to not completely rely on each of the activations and rather look for general patterns.

3.3.3 Convolution

A convolution layer is a biologically inspired layer [18] which performs the operation it name implies, a convolution over the image. A convolution (Figure 5) is performed by sliding a kernel (a pattern) of size $k \cdot p$ (usually $k = p$, where k and p are odd) over the image and computing the element-wise multiplication of its elements by the images on each pixel. The main advantage over the traditional convolutions is that features, or kernels, must not be hand-chosen, as the network learns the best features for its input.

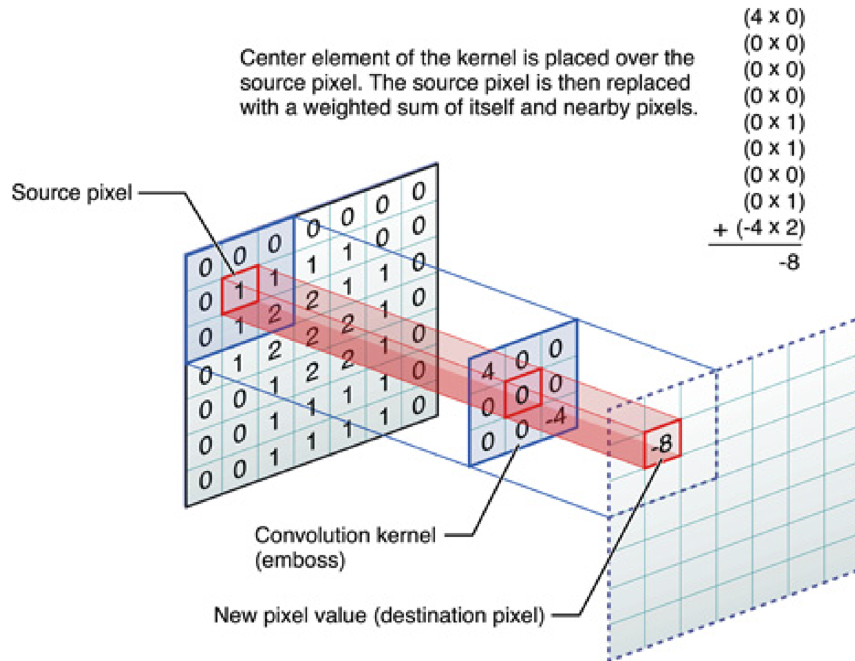


Figure 5: Convolution of a 3x3 kernel on a reference image
Image by Apple

Convolutions have a few advantages over fully connected layers:

1. When working with images, they:
   (a) Preserve the spatial shape of the image along the forward and backward pass
   (b) Find textures on the image
   (c) Find boundaries on the image
2. They greatly reduce the number of parameters of the network

The number of parameters of this layer is equal to the kernel's size. We say this layer has shared weights because it applies the same $k \cdot p$ weights all over the image (it convolves

the $k \cdot p$ kernel), instead of having a weight per neuron/connection like the fully connected layer. We usually let this layer produce more than a single filter, thus having q different filters of size $k \cdot p$, giving a total of $q \cdot k \cdot p$ parameters. Also, it may have an input of r channels (activations) which must also be taken into account, giving the final formula of: $r \cdot q \cdot k \cdot p$ parameters.

This layer allows to tune the following hyperparameters:

1. **Number of filters:** The capacity of the layer, how many filters it will convolve on the image. The shallower the layer, the fewer filters we need as less information is available.
2. **Kernel size:** The size of the kernel(s) to be convolved. As with traditional convolutions, the bigger the filter, the bigger structures it will find.
3. **Stride:** The interval at which the filter is applied to the input, usually 1
4. **Padding:** The kernel needs $k/2$ extra pixels on the image for it to output an image of the same size of the original. This is usually solved by adding a 0 padding to the original image. Some frameworks, like Tensorflow, abstract it to some predefined padding types ('same' and 'valid').

3.3.4 Pooling

When working with CNNs the pooling layer is important because it allows performing non-linear downsampling of the original input image (Figure 6). This layer operates amongst regions of the input and performs an operation on them so that only one output is given. Specifically, it can compute the following operations:

1. **Maximum:** Over the region, output the maximum value found
2. **Minimum:** Over the region, output the minimum value found
3. **Average:** Output the average value of the region
4. **Stochastic:** Output a random value of the region

As with the convolutional layer, we can define the kernel size and stride. It is common in pooling operations to use a kernel size of $2x2$ and a stride size equal to the size of the kernel, that is 2 in this case (k in the general case).

When working with images the most used type is the maximum pooling, because it allows the network to be rotation invariant, in the sense that small changes will not affect it, as the overall intensity will be the same.

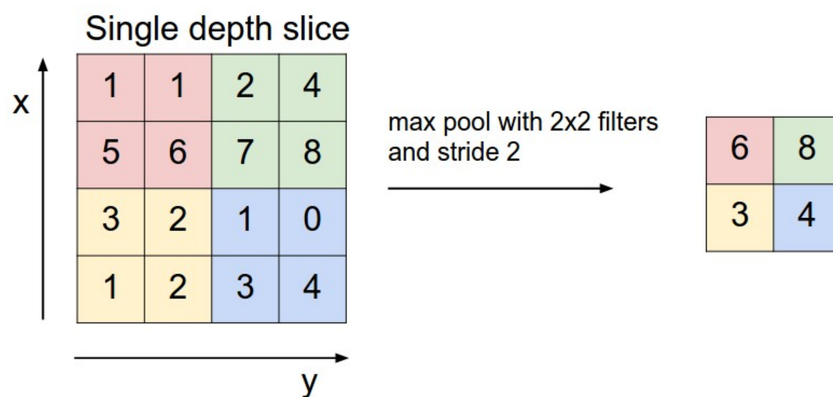


Figure 6: Pooling of a 2x2 kernel, stride 2, on a reference image
Image by Stanford University

3.3.5 Batch Normalization

This layer normalizes its input by a moving average on the mean and standard deviation of the activations. It does, indeed, perform a standardization, the output has a 0 centered mean and standard deviation 1 [19]. It also biases and scales the values by two learnable parameters, also named bias and scale.

It has proven to be useful because it:

1. Stabilizes the gradient and solves exploding gradient problems
2. Helps reduce overfitting
3. Accelerates learning
4. Makes the learning rate influence more superficial, in the sense that finding a good learning rate becomes an easier task

It is being widely used and is thought to replace the local response normalization layer, which was used in conjunction to convolutions to normalize the activations, and even could deprecate the dropout layer.

3.3.6 Summation

This type of layer simply joins two other layers by performing the element-wise summation of the activation of both layers.

$$f(x_1, x_2) = x_1 + x_2$$

It is commonly used in architectures like the residual networks, which will be detailed in section 4.2.2.

3.3.7 Other layers

In the field of Convolutional Neural Networks we also have the Local Response Normalization [20], which performs a neighbourhood value normalization. More related to the field of feedforward networks we have the Highway Layer [21]. Some other layers not used in this project are the RNN related LSTM [22] and the GRU layer [23].

3.4 Learning algorithm

The algorithm to learn the weights or parameters of the network receives the name of *backward propagation of errors*, abbreviated to *backpropagation*. This method only works for supervised learning because it requires the target output to be known so that a loss function (error function) can be computed.

It is a generalization of the single-layer delta rule, updated to work with multi-layered networks. It requires to first forward[2] the input through the network and then backpropagate the errors from the output to the input, making each layer rectify its weights for a proportional part of its error. To correctly reduce the error we iteratively subtract a portion of the error, for example by using the gradient descent algorithm.

In the simpler case (Figure 7), a network with no hidden layers and only and input and output layer, we need to think of the network as an optimization problem, whose last layer outputs the error as:

$$E = \frac{1}{2}(y - \hat{y})^2$$

Where $\hat{y}$ is the network prediction and y the target value[3]. Which, in the one-dimensional space would produce a convex function whose representation can be seen in Figure 8.

We already know the error for the last layer and how to minimize it, by taking its derivative and moving into the direction that minimizes it:

[2] In neural networks feeding some input through the first layer and propagating it until the last is usually denoted as forward

[3] We include the term $\frac{1}{2}$ to cancel, later on the derivative, the 2 from the power

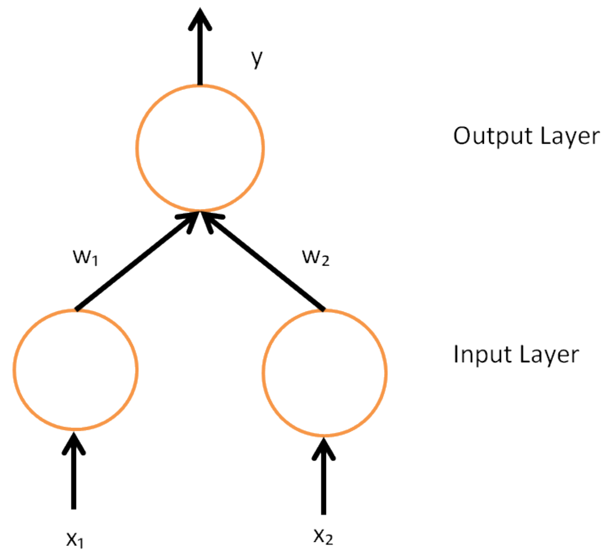


Figure 7: Network with no hidden layer, input fully connected to output

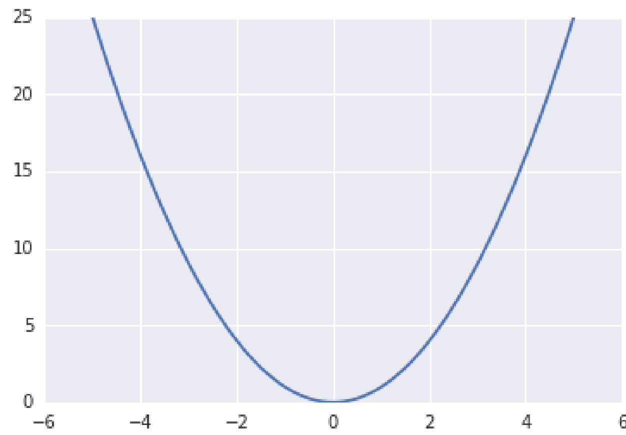


Figure 8: Euclidean error of a 1D function

$$E' = y - \hat{y} \tag{6}$$

In case of multi-layered networks, the contribution of the weights w_{ij} to this layer $layer_j$ can be extrapolated by using the chain rule, that is:

$$\frac{\delta E}{\delta w_{ij}} = \frac{\delta E}{\delta output_j} \frac{\delta output_j}{\delta layer_j} \frac{\delta layer_j}{\delta w_{ij}}$$

However, we can see that $\frac{\delta layer_j}{\delta w_{ij}}$ is the same as the output of the following layer. That means we have to start backpropagating the error from the last layer to the first one. We may also note that the derivative of an output with respect to its input is given by:

$$\frac{\delta output_j}{\delta layer_j} = \frac{\delta g(x)}{\delta x}$$

That is the derivative of the activation function with respect to its input. Thus, this connects with the statement made in Section 3.2, that all activation functions must be differentiable. If we suppose we are using the sigmoid function $g(x) = \frac{1}{1+e^{-x}}$, then

$$\frac{\delta output_j}{\delta layer_j} = g(x)(1 - g(x))$$

Finally, the term $\frac{\delta E}{\delta output_j}$ is straightforward in the output case of the output layer, where we simply need to compute the derivative of the error (Equation 6). However, in case of a layer in the middle of the network, the error E is given by the accumulated error of the other layers, that is:

$$\frac{\delta E(output_j)}{\delta output_j} = \frac{\delta E(net_{j+1}, ..., net_{j+k})}{\delta output_j} = \sum \left(\frac{\delta E}{\delta output_j} \frac{\delta output_j}{\delta layer_j} \right)$$

Then, we can generalize this expression to:

$$\frac{\delta E}{\delta w_{ij}} = \sigma_j output_i$$

$$\sigma_j = \frac{\delta E}{\delta output_j} \frac{\delta output_j}{\delta layer_j} = \begin{cases} (y - \hat{y})g'(x), & \text{if } j \text{ is output} \\ (\sum \sigma_i w_{ji})g'(x), & \text{otherwise} \end{cases}$$

Finally, we must update the network weights. It can be done by multiplying the error computed with a negated learning rate α. We change its sign because we want to find the minimum of the optimization problem. The term α, called learning rate, must be chosen so that the function converges fast but the steps are not too big:

$$\Delta w_{ij} = -\alpha \frac{\delta E}{\delta w_{ij}} \tag{7}$$

This last Equation 7 is called the Gradient Descend, which minimizes the error iteratively by taking steps. However, as easy as it looked with the convex function in Figure 8, a more realistic error function might be the one in Figure 9. This figure makes it clear that a small α would make the minimization process get stuck in a local minimum. However, choosing a big α would make the function not converge and oscillate in the global minimum. Choosing the right α is crucial for the network to work.

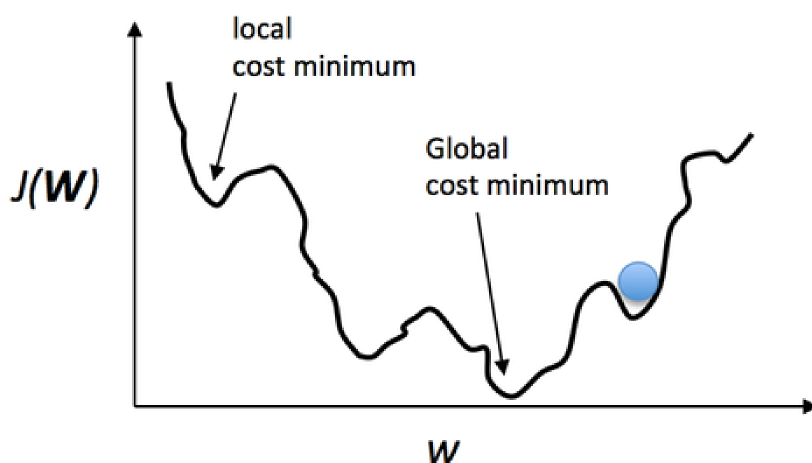


Figure 9: Real error function of a multi-layered network

Nowadays gradient descend is not used as is, but using the so called Stochastic Gradient Descend. The main reason behind this is that the first would require the whole dataset to be forwarded and backpropagated at once. Datasets usually consist of thousands or even millions of data, which makes this impossible. Input is then submitted either one-by-one or using mini-batches, small groups of data.

4 Architecture

In this section it is written an explanation on each of the tested architectures. It is explained how and why the final architecture was built and all the logical reasoning to come up with it.

4.1 Method 1: Convolutional and fully connected layers

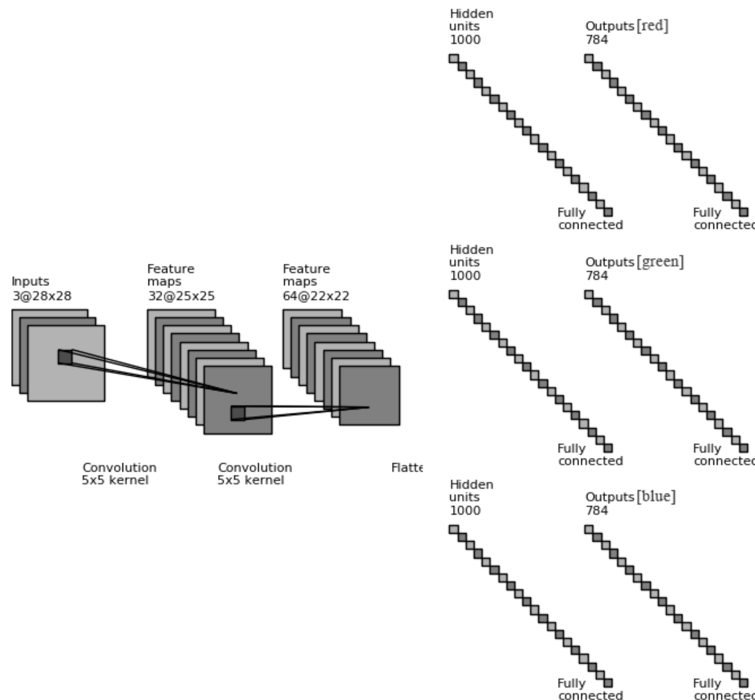


Figure 10: 2xConvolutions of a 5x5 kernel without padding + 3x(1 fully connected layer of 1000 units + 1 fully connected layer of 784 units)

The initial version of the neural network was a combination of convolutional, pooling and fully connected layers (Figure 10). It can be said this first attempt is naive and simple, as it attempts to reconstruct the color with a small number of layers.

First, a common set of convolutional layers with ReLU activations is placed to allow the network to find textures. They are shared amongst the 3 resulting channels, RGB, as the textures will be the same independently on which channel will be used. The idea is to let these convolutional layers to find boundaries, textures and more complex representations

as they get deeper. A system like this one, which has a common set of layers first and then divides into more branches might be called *early-fusion* [24], while the opposed would be a *late-fusion* network. In the *early-fussion* approach, the features are considered to be dependent between them, which makes sense in our particular case.

Then, each channel of the RGB space is rebuilt separately by splitting the network into 3 different branches. Each of them contains a set of 2 fully connected layers with ReLU activations. For this to work, the last layer must have exactly the same size of the input image (for example, if working with $28x28$ images, it would be $28 \cdot 28 = 784$), because we must reshape them to the original size at the end.

The error, or loss, was computed as the sum of the euclidean distances between each of the resulting channels and the respective target channels.

In order to explain why an alternative architecture was used, we need to analyse this network from the point of view of the number of parameters. The network worked fine when an image of low resolution, for example $28x28$, was used. But when trying with bigger images of size $256x256$, the last layer had to be of $256 \cdot 256 = 65536$ neurons. That means it would require a total of $65536 \cdot 1000 = 65536000$ parameters only for the last layer. That exceeds the maximum size of an integer ($2^{15} - 1 = 32767$) and is about 260MB of memory ($65536000 \cdot sizeof(float) = 65536000 \cdot 4 \approx 260MB$), which makes it impractical for any learning process to succeed.

It became clear that a network using fully connected layers would not be able to accomplish the task due to memory constraints and computational power limits. So, although this architecture worked with small images, the demonstration can be seen in Results, it was unable to use full-size images.

4.2 Method 2: Fully convolutional networks

A network with no fully connected layers, consisting of convolutions and non-linearities only, is usually called a fully convolutional network. As we have already said, the big advantage of convolutions is its low number of parameters and its capacity to detect repeating patterns. Consequently, they are ideal when working with high-resolution images where one expects to find common textures.

The network used in this project uses a wide variety of concepts that need to be explained before attempting to understand the network itself.

4.2.1 Finetuning

Finetuning, also called transfer learning, is a technique based on initializing the network's weights with the weights of another already-trained network. Take for example the VGG network, which has been trained to recognize 1000 different types of objects. Its first 10 convolutional layers must have learned to recognize complex patterns for those objects. If we were to recognize another object, even if it is not among those 1000 classes, it makes sense to use those layers as our first layers too, because the network would already know what to look for. The alternative to finetuning is initializing the network either by random values or using probability distributions like the gaussian.

The main advantages of using finetuning over a plain random initialization are:

1. Better initialization yields more stable gradients, avoiding the common vanishing (tending to 0) and exploding (tending to ∞) gradients problems
2. Faster learning, as the network already has good representations of the objects

Also, when transferring weights from one network to another, we may choose to: *a)* freeze them and stop the network from modifying them at all, *b)* allow them to be modified but at a slower rate compared to other layers (by selectively decreasing its learning rate) or *c)* treat them as another variable to be optimized.

The first approach is used when the trained network already has good representations for the problem we are facing. We consider its filters to be good enough to detect all kind of objects we might need. Likewise, the second case also assumes this, but allows small modifications for a better fitting on the current data.

Common sources of fine tuning are the VGG and the Alexnet [20] (or modifications of it), as its first convolutional layers are proven to have learned complex representations. Googlenet [25] and residual network are now being used too.

For example (Figure 11), the convolutional layers from the VGG network are used to initialize a new network which must predict dog breeds. Note that only the convolutional layers are taken, and its weights are frozen, but the remaining fully connected layers are trainable, so that it can adjust to the new categories to predict.

4.2.2 Residual Networks

Intuition might tell that deeper networks, consisting of more chained convolutional layers, might lead to more complex representations of the input data. This, although it is true, comes with hard to solve problems:

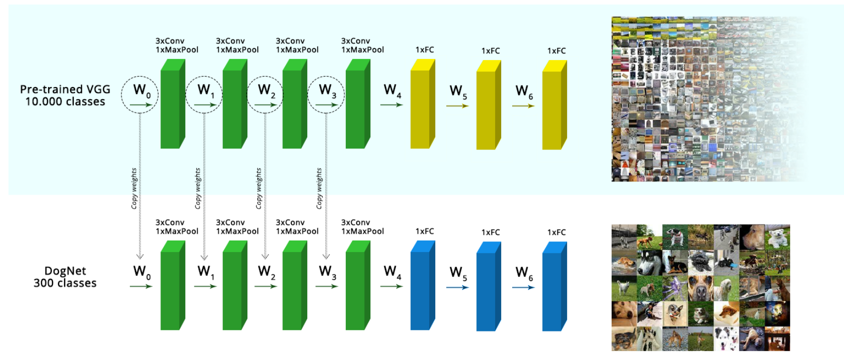
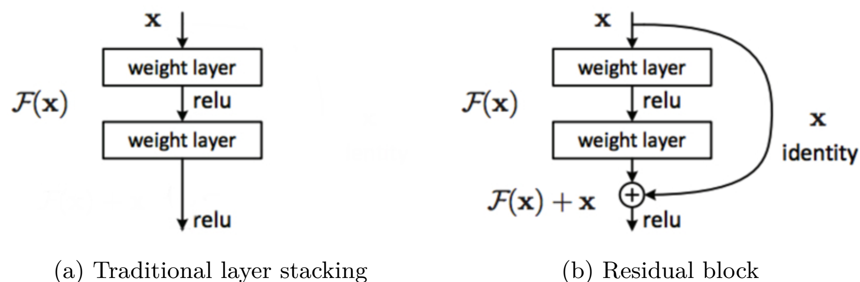


Figure 11: Finetuning of the VGG convolutional layers to classify doog breeds

1. Exploding or vanishing gradients: The gradient might tend to 0 (thus having no direction) or to explode and go out of the data range. This problem has been recently solved by the batch normalization layer, but classic strategies like better initialization or gradient clipping are still useful and help to prevent these issues.
2. Degrading gradient: A deeper network will see its gradient lose its reference to the input data, as in it will go so deep that the gradient will no longer hold meaningful information. This makes the accuracy to get saturated at a lower point than normal. Then, either it does not improve at all it worsens over time.

A recent paper from Microsoft defines an architecture they call Residual Networks which addresses and solves these problems. A normal neural network would stack its layers in a linear fashion (Figure 12a) while the Residual Networks take a different approach, they use residual blocks (Figure 12b).



(a) Traditional layer stacking

(b) Residual block

Figure 12: Tradicional layer stacking approach vs residual blocks

A block is composed of two convolutional layers and two non-linearities (ReLU on the paper), with the particularity that before the last ReLU the input is summed to the activations. This allows the network to propagate some remaining information of the original input (thus the name of *residual* networks).

It has been shown that using this strategy a network can go as deep as 1000 layers on the CIFAR-10 [26] dataset and of 152 layers (8 times the size of the VGG) on the imagenet dataset. Not only they successfully converge, but they also improve the previous best accuracy.

4.2.3 Color space

Although not strictly related to CNNs, choosing the right color space is fundamental for the network to work faster and better. The immediate and naive choice would be to use RGB color space, which consists of 3 channels that mix lighting (intensity) and colors all together. This, however, implies there is a high correlation between channels: increasing either one of them would increase the overall intensity of that pixel. While RGB is a good color space for humans to easily comprehend and visualize colors, it is not practical when it must be used for machine learning algorithms.

When looking for non-correlated color spaces, the most habitual are [27]:

1. YUV: It is usually employed for video or image processing, commonly seen in the PAL and NTSC specifications. It defines a luminance component (Y) and two chrominance channels (UV) where color is encoded.
2. HSV: From Hue/Saturation/Value, it may also be called HSI (intensity) or HCI (chroma or colorfulness) amongst others. The main advantage of this color space is being intuitive. Colors might increase intensity in a natural manner, increasing only Value, unlike RGB where you would have to increase each channel or YUV where you would likewise have to increase UV. It also features a separation (Value) of intensity from color.
3. L*a*b*: This color space is one of the created by *Comission Internationale de l´Éclairage* (CIE) and considered standard in many applications. All colors from CIE try to imitate human vision and, as such, are based upon it. They achieve to be uncorrelated between lightness (L) and color (a,b) but fail to be easily interpretable.

Doing a brief recap we can easily see that the input image, a grayscale image, could be used as the luminance/intensity channel of either of these color spaces. Strictly checking the values one would see that the Y/V/L channels (respectively to each color space) are not exactly the same as the grayscale conversion of an RGB image, but the error is so low that any difference would not be noticeable. This means we could reduce the problem of outputting a 3 channels RGB image to a 2 channels YUV/HSV/L*a*b* image, as the remaining 3rd channel would already be the submitted image.

When choosing the final color space, HSV got discarded due to even being intuitive, it is not as powerful when it comes to encoding colors. Between YUV and CIE L*a*b*, the

decision was in favour of L*a*b* for a simple reason, it is more widely used and considered to be standard, while being backed by the CIE organization.

Tests were done to assure that using a color space with 2 channels for an image would be better than RGB, and all of them showed significant improvements. The final architecture was then completely based off the L*a*b* approach.

4.2.4 Final architecture

The final architecture is shown in Figure 13. It will be first globally described and then individually inspected.

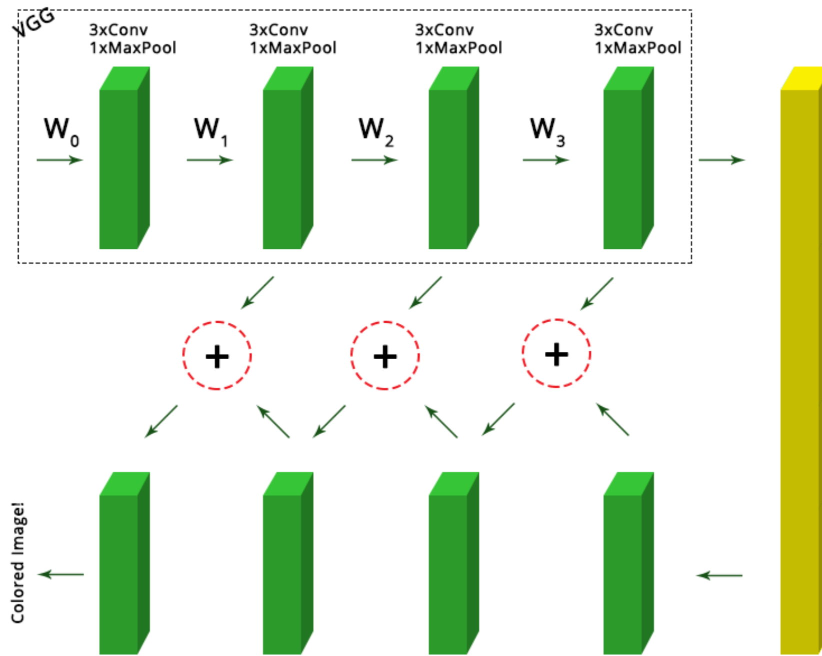


Figure 13: Final fully convolutional architecture for image colorization. Each green block consists of one or more convolutions. The yellow block is a convolution of a 1x1 kernel. The summation is equivalent to the summation of the batch normalized green block above and the convolved upsample of the activation of the green block below.

First of all, the image is forwarded through a subsection of the VGG network. Specifically, it includes all the first 4 blocks of convolutions, but where a block of three convolutions is used in the original VGG network, here a block of only two is used, due to memory constraints. All the weights of this section are fixed, so that they are not taken into account during the optimization process.

Then, an upsampling process is done in each of the *upsampling blocks*, described below.

The VGG performs a Max Pooling after each convolution block, so, if we use an initial image of size $256x256$, taking into account the convolutional layers have padding and we have a total of 4 poolings, the final size will be $16x16$. To restore the original size the network will also need to do 4 upsamplings.

Upsampling block: One important characteristic of the VGG network is that the number of parameters is constant all along the network. Whenever a pooling is done the resulting activations are reduced by half, but the number of filters on the next convolutions is doubled. The upsampling blocks, Figure 14, will follow the same logic, whenever the image (or activations) is doubled in size, the number of filters in the filters bank will be halved, thus achieving the constantness in parameters.

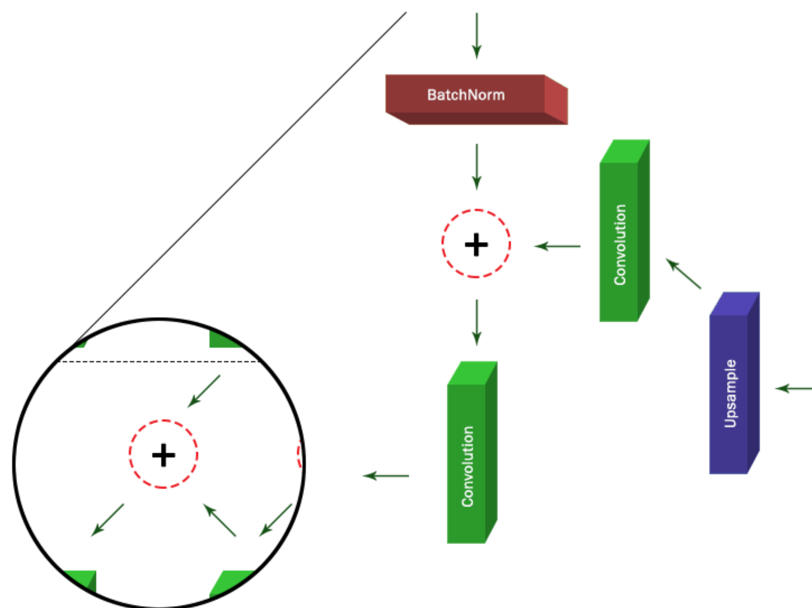


Figure 14: Closer look of the upsampling block

The upsampling might be done by using a bilinear filter, but for performance, a simpler nearest neighbor filter with 4-connectivity has been used. Both have been tested and no real impact is perceptible in the final image, neither color nor texture-wise.

Taking the same idea as the residual networks, the activation from the same level on the VGG is summed to the upsampled image, so that the gradient is more stable. To make it even more stable, before summing, the activation of the VGG network is forwarded through a batch normalization layer.

To allow the network to have more complex representations along with a bigger number of parameters, a convolution is also done to the upsampled image and to the summation

result. Otherwise, the network would have no parameters to optimize (as the VGG is fixed and neither the batch normalization nor the summations have any significant parameter).

The next block simply takes as input the result of the last convolved summation, and does the same operation but at a bigger size and with a lower number of filters.

Activations: The first tests were done using ReLUs as non-linearities after each convolution. The network completely failed to color the image and would often output either grayscale images or sepia-colored images. Our color space, L*a*b*, uses numbers in the range $[-128, 127]$, which we normalize to $[-1, 1]$. It is obvious then that if all our weights range from $[0, \infty)$, due to the ReLU, we can not represent the lower negative half.

That is why all the network now uses hyperbolic tangents as activation functions. Their derivatives are still easy and fast to compute and behave as expected, expanding the activations to the whole range of the color space.

Network output: After the last upsampling block two more convolutions are performed, one with the same number of filters as the block and the last with only 2 filters. As we want to output a 2-channels image, this is exactly what the last layer does, converge to 2 outputs.

As a measure of error, or loss function, the mean of the euclidean distance between the target image and the resulting one is used:

$$E = \frac{1}{N}(y - \hat{y})^2$$

Where y is the target image and $\hat{y}$ the resulting image. N depends on the batch size, the number of images fed at the same time to the network, but in general terms that is:

$$N = batch \cdot 256 \cdot 256$$

The chosen batch size will be explained in more detail in the implementation section.

5 Implementation

To implement the network two frameworks have been used, Caffe [28] and Tensorflow [29]. Both of them with its advantages and disadvantages. Many more libraries and frameworks exist, just to name a few: Theano, Neon and Microsoft CNTK.

5.1 Frameworks

5.1.1 Caffe

Its main backers are the Berkeley Vision Group. Caffe's main feature is its performance and speed, being the fastest among all the frameworks at the time of writing this project. It is built in C++ and exposes a Python interface. All the networks must be written in "Protocol Buffers" from Google[4], which although makes the files clear and well structured, it is not practical when one must write repeated structures. Its Python API allows defining custom layers but with a high performance cost. It is mostly focused on visualization and testing rather than network definitions.

Being the precursor of convolutional neural networks, it has a fine-tuned code and perfect integration with GPU acceleration through NVIDIA CUDA (and recently OpenCL). Part of its high performance comes from its hand-written derivatives. All layers must specify the derivatives and it does not include any automatic-differentiation system. It is also one of the best systems when it comes to memory usage, being able to fit large networks just fine.

5.1.2 Tensorflow

Recently open-sourced, it is the framework used internally by Google. Although it is not as fast as Caffe nor does manage memory as fine, it has a few advantages:

1. It includes auto differentiation, so when creating new operations or layers, there is no need to hand-write the derivatives, as they are automatically calculated
2. Fully exposed API to Python and many more languages, including network definition (graph) with built-in support for language specific constructions (such as for-loops, conditions...)

It features a complete set of functions for traditional neural networks, convolutional and recursive. Newer papers and techniques can easily be implemented thanks to its support for complex math operations and a supportive community.

[4] Available at `http://code.google.com/apis/protocolbuffers/`

More technical aspects of Tensorflow will be discussed below, referring specifically to the implemented architecture and better illustrating its functionality.

5.2 Network implementation

As Tensorflow came later, this project was started with Caffe. As soon as Tensorflow was available and stable enough, the whole architecture was ported to Tensorflow. Even though two different frameworks have been used, the core of the layers are exactly the same, so an initial general explanation will be done.

5.2.1 Optimizer

To optimize the network an Adam [30] optimizer has been used, tweaking the *epsilon* parameter. It has been tested that problems like imagenet require an epsilon of $\approx [1, 0.1]$. To find the appropriate learning rate and epsilon combination, a grid-search has been used. That is, all combinations of learning rate in $[1 \cdot 10^{-1}, 1 \cdot 10^{-4}]$ and epsilons $[1, 1 \cdot 10^{-4}]$ have been tested until the best one has been found. The net which converged faster and produced better results was using learning rate $1 \cdot 10^{-3}$ and epsilon $1 \cdot 10^{-3}$. To make it work better, a decaying exponential learning rate has also been used, every 1000 iterations the learning rate is exponentially decreased by 0.95.

The batch size has also been tested to find the best one, compromising performance and results. A batch size of 1 image does ≈ 5.25 iterations per second, but fails to converge with good enough results. A batch size of 32 is slow, each iteration takes about 120 seconds, and the network takes too long to achieve good results. Any number higher does not fit in the memory of the GPU. Eventually, the batch size was set to 8, which yields goods results after a decent amount of time.

It has also been tested whether the network generalizes better by applying an L2 regularization to each weight (smoothing them) of either $(0.001, 0.003, 0.05)$ or none at all. At the end, the version with no regularization worked better and produced better results.

5.2.2 Number of parameters

As said, the VGG network maintains the number of parameters constant (except the last that reduces dimensions for the next fully connected layers). Each one ordered as $(width, height, channels)$[5]:

[5] The 16x16x1024 convolution is the one with a 1x1 kernel

$$
\begin{array}{ccccc}
256x256x64 \rightarrow & 128x128x128 \rightarrow & 64x64x256 \rightarrow & 32x32x512 \rightarrow & 16x16x512 \\
 & & & & \downarrow \\
256x256x64 \leftarrow & 128x128x128 \leftarrow & 64x64x256 \leftarrow & 32x32x512 \leftarrow & 16x16x1024
\end{array}
$$

The second line corresponds to the upsampling blocks, after upsampling, convolving and summing. Finally, two more convolutions are performed to obtain the ab channels: $\rightarrow 256x256x32 \rightarrow 256x256x2$.

We can calculate the number of parameters of a given convolutional layer by applying the formula: $input_filters \cdot kernel_height \cdot kernel_width \cdot filters + bias$. The $bias$ has size equal to $filters$ in the equation above, as it is 1D. Knowing the upsample block consists of 2 convolutions, one with the same number of filters as the input and the other halved, we can easily calculate each of blocks parameters:

$$
\begin{aligned}
&Initial && =512 \cdot 1 \cdot 1 \cdot 1024 && =524288 \\
&Upsample4 && =(1024 \cdot 3 \cdot 3 \cdot 512 + 512) + (512 \cdot 3 \cdot 3 \cdot 256 + 256) && =5899008 \\
&Upsample3 && =(256 \cdot 3 \cdot 3 \cdot 256 + 256) + (256 \cdot 3 \cdot 3 \cdot 128 + 128) && =885120 \\
&Upsample2 && =(128 \cdot 3 \cdot 3 \cdot 128 + 128) + (128 \cdot 3 \cdot 3 \cdot 64 + 64) && =221376 \\
&Upsample1 && =(64 \cdot 3 \cdot 3 \cdot 64 + 64) + (64 \cdot 3 \cdot 3 \cdot 32 + 32) && =55392 \\
&FinalConv1 && =32 \cdot 3 \cdot 3 \cdot 32 + 32 && =9248 \\
&Output && =32 \cdot 3 \cdot 3 \cdot 2 + 2 && =578
\end{aligned}
$$

We should also take into account that batch normalization layers have also 2 parameters. We have 4 layers of this type, which are 8 parameters in total. So, the network has

$$
Total = 524288 + 5899008 + 885120 + 221376 + 55392 + 9248 + 578 + 8 = 7595018
$$

parameters to optimize.

5.2.3 Tensorflow specific

As the final architecture is done and presented in Tensorflow, a deeper explanation on how it works will be done in this section. Tensorflow has, as they say, a data oriented flow. When working with Tensorflow one has to think in terms of the graph, not as a functional programming language.

The graph is nothing but the assembled tree of all the operations, variables and placeholders defined. When you create either of these using the API it provides, no real calculus or operation is ever performed. It just stores the reference as a node in the graph and postpones the real operation.

To actually execute anything first a session must be started. Sessions are specific to devices, as operations, allowing to execute on GPU or CPU seamlessly. When the session is running, you can evaluate or run any expression on the graph. Such evaluation yields the real value. Doing so Tensorflow manages to:

1. Reduce all Python $\leftrightarrow$ C++ communication, which would slow down all processing
2. Keep an organized directed acyclic graph which can be navigated or executed independently

When using this framework, the so far called weights or parameters are renamed to Variables. Any Tensorflow Variable will be a learnable parameter in the network, if not specified otherwise. Tensorflow allows to keep a hierarchical graph of variables by using *variable_scope*s. They allow all variables inside the scope to be initialized in the same manner and keeps references to them grouped by the same common prefix, the scope's name.

As Tensorflow operates separately, in the session, to pass variables from the Python/C side to the actual computational session, one must use placeholders. Unlike variables, placeholders are not trainable and its only purpose is for data loading.

It also comes with a wide range of functions to initialize weights and a huge variety of optimizers to train all the variables. Tensorflow does not, however, provide already done layers, as it only makes available mathematic operations. Supplied with this project code there are a few Python helper files which extend the framework:

1. *Dataset.py*: Usually datasets are big and do not fit in the main memory (RAM memory). This file provides classes to procedurally load from the drives. Only a batch is loaded at once. Furthermore, the data is loaded asynchronously from a thread, using the Python *Queue* object which is thread-safe by default. The training and loading processes can be done at the same time, improving performance and reducing the amount of time it takes to train the network.

   Supplied classes allow to load from a) text files, where each line is a combination of the image path and its class, b) LMDB files, using the Caffe format for image serialization and de-serialization and c) native numpy arrays, that although does not reduce memory usage it facilitates obtaining cyclic batches.

2. *Models.py*: It includes the definition of the VGG network graph, a reduced version of it, and some base classes to simplify models:

(a) LearningRateDecay: Tensorflow allows to have an exponential decay over the learning rate, but involves quite a few lines. This class simplifies it by grouping everything in a simple object, with no additional calls.

(b) Optimizer: Wraps the optimizer, allowing to easily clip gradients or apply decaying learning rates. The methods exposed by this class are the same as the exposed by the original Tensorflow optimizers, to ease the usage.

(c) Model: When developing more than one model or architecture, many lines of code are the same. This class groups them and reduces the amount of code needed to get a functional model, letting the user concentrate in the important aspects.

3. *Layers.py*: Defines all layers as classes. They all inherit from a base class which defines default values that can be tuned, for example, if variables are trainable or not. They are all based in the Tensorflow basic operations.

   The main point on using classes is they make it easy to access the variables declared in the layer itself, as they are saved as attributes of the class itself.

4. *Network.py*: It basically defines all layers, but as bare functions instead of classes. They are more straightforward than the class based approach and might be faster to use if one simply wants to quickly define a model. It also implements methods to load saved numpy pre-trained models.

5.2.4 Visualization

An important part when doing neural networks is not only the results but the visualization of the network, which implies:

1. In our particular case, visualizing the results
2. Monitoring and validating the loss and accuracy (if it applies) of the network, to see possible under-fittings or over-fittings
3. Visualizing the learned filters on the convolutional layers, to see and verify they make sense

Here Jupyter notebooks (previously known as IPython notebooks) have been used. They are considered as the best option because they allow to interactively program and see the results inline. All of it within a browser and without needing physical access to the machine.

Furthermore, it allows a wide range of languages to be used, like Python, Julia and R. Due to its fast programming and extensive libraries available, Python has been the language used. Caffe at some points had problems with Python and Julia was used as a backup.

To ease the process, OpenCV was used to do RGB $\leftrightarrow$ L*a*b* conversions and image manipulations, such as resizing and reading from disk. Matplotlib served as the main plotting library, both for showing images and numeric results.

Tensorflow also comes with a visualization software called Tensorboard, which enables us to: a) see a complete graph of our network (Figure 15), b) see all tensors in the graph and c) inspect values all along the execution, including plots with average values, deviations... It has been widely used to see the real progress and its values will be further discussed.

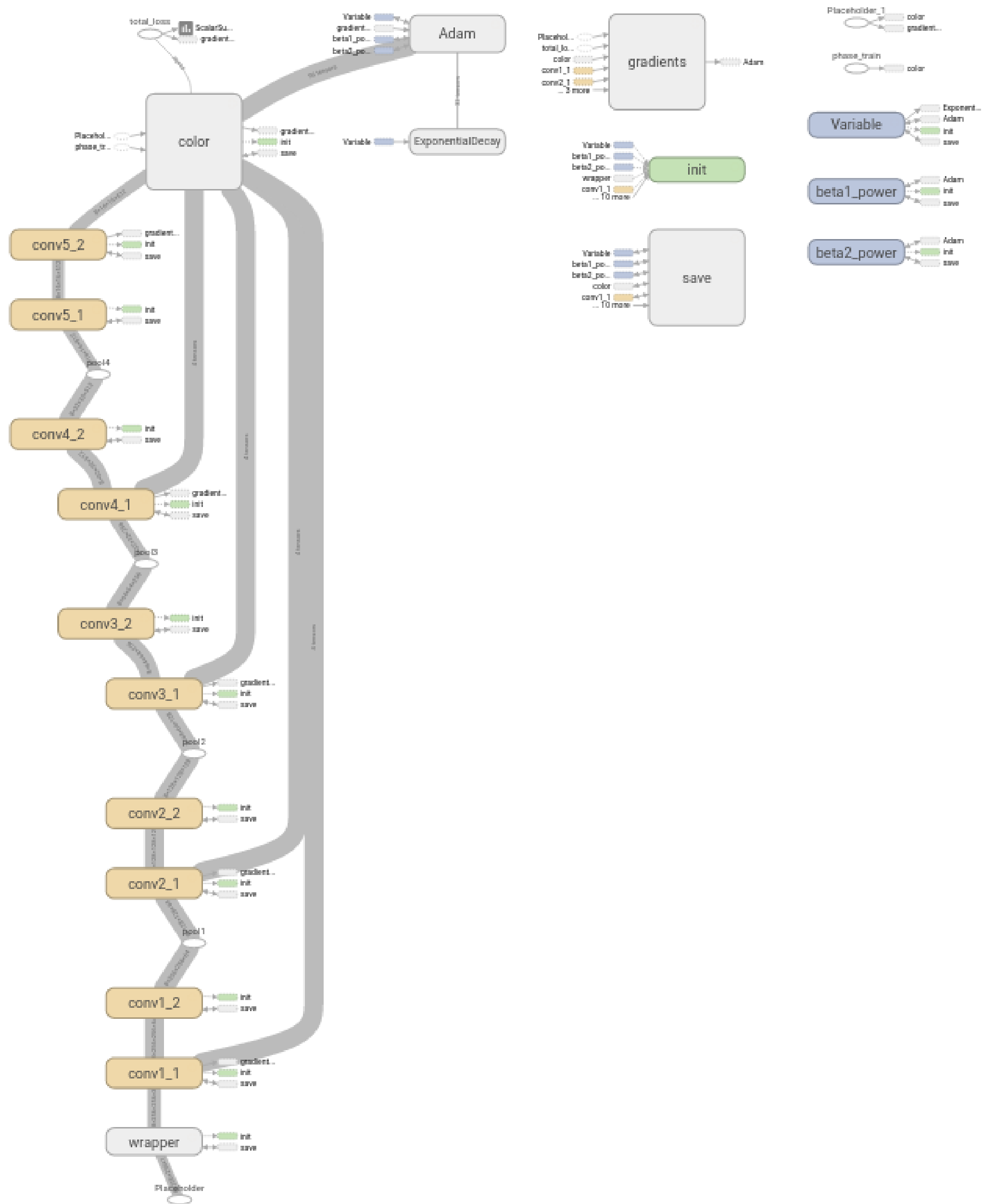


Figure 15: Graph of the colorization network

5.3 Online website

Any user should be able to use this technique, but it is clear that the Python files provided require at least some basic programming knowledge. The programmed website allows inexperienced users to colorize images and at the same time serves to do quick tests.

The website consists of three sections, one where the user is able to upload an image and the system will automatically colorize it; and two informative sections, the first which links to this document and the later which gives information about the author.

The user should also be able to supply images not in grayscale, thus the website must automatically convert them. Of course, if the input image already has color, it will be completely discarded and not used in the colorization process.

From the implemented website it will be analysed the backend (the server) and the frontend (the actual user-interactive side). The interaction will be explained in a different subsection, as it involves both sides. The whole implementation is supplied as source code and found in the *WebServer* folder.

It is completely integrated with the interface Tensorflow gives and runs fine using CPU only, so it could be deployed to a system which does not have GPUs.

The website, as seen in Figure 16, is available at: `http://g-build-server.asuscomm.com:5000/`

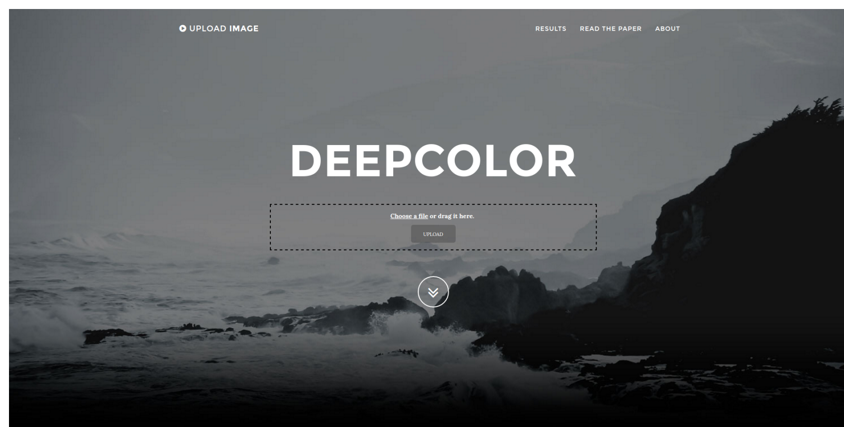


Figure 16: Landing page of the implemented website

5.3.1 Backend

It is completely programmed using Python. The main reason behind using Python and not any other server-scripting language, like PHP, is that Tensorflow already exposes a complete API to Python, so no piping or communication with another Tensorflow process will be required.

Flask: It is one of the many webservers programmed for Python. Other alternatives, such as DJango and Bottle exist, but they come bloated with much more middleware and utilities which will never be used for this project (like an admin panel in case of DJango). Flask is lightweight and fast, and programming a website is easy and costless.

Creating a user visitable page with Flask is easy, it is defined as a Python function and decorated with a helper function from Flask. The decorator, usually *@app.route(path)*, takes a parameter which corresponds to the URL for that page. Then, two options are available, either directly returning a string containing HTML code or parsing and rendering a template.

This project uses the templates engine provided by Flask, Jinja2. This engine allows to pass variables, even objects, arrays and dictionaries, from the Python side to the HTML template. Those variables get substituted and sent to the user as bare HTML. Templates allow to make a clear separation between the controller (Python) and the view (HTML), so they help to keep clean and understandable code.

This website is stateless, it maintains no information of the session, the user nor the uploaded images, so it does not require of sessions nor databases.

Gunicorn: Although Flask itself is capable of handling incoming connections and serving requests, it is not powerful and saturates quite easily. Thankfully, we are not obligated to use it, we can use external webservers, like Gunicorn, nginx or Apache. Again, with the aim of keeping the code small, having as few dependencies as possible and being lightweight, Gunicorn has been chosen.

Gunicorn integrates perfectly with Flask and no adjustment has to be done in order for it to work. Even more important, our website is input/output (I/O) intensive, as it must read and convert an image, blocking meanwhile. A normal webserver, as Flask, would leave all connections hang until the conversion finished. Gunicorn, however, allows us to spawn more than 1 worker, so that when one is blocked the others can still work. This project uses 5 workers, although more could be used.

5.3.2 Frontend

The user receives a website done with HTML5, CSS3 and Javascript, based on a free available template called Grayscale[6]. Although it uses a template, none of the functionalities here described were implemented beforehand, particularly the uploading process or the communication with the backend.

HTML5: Important features of the HTML5 language have been used for this website. To convert the input image to grayscale a *<canvas>* element is used combined with the Javascript object *FileReader*. Basically, the submitted image is read using the FileReader and drawn into the canvas, which allows to separately select each channel. It then averages the 3 RGB channels, obtaining a grayscale image. Drag and drop is also doable thanks to HTML5 and jQuery.

Javascript: It is used mainly through the jQuery library, which simplifies the syntax and provides useful functions for DOM manipulation. It is used to update the dragged images and the texts and elements that display the uploading and colorization progress.

It also simplifies all the AJAX communication, used to interchange messages with the backend without making the user reload the page or follow any link. This enables the user to do a transparent upload and colorization.

Also, jQuery introduces nice visual effects, like the upper menu and the smooth page scrolling when following links or visualizing the colorization result.

CSS3: It also uses an external library, called Bootstrap, to make the web fluid and adaptable to different screen sizes. It comes into special importance when visiting the website from a mobile device, because it is able to correctly rearrange to maintain a pleasant navigation.

Custom styles are used to make uploading process give feedback to the user. Also, achieved through CSS, the user can hover the mouse over the colorized image to see the original grayscale image.

5.3.3 Communication

As already advanced, AJAX is used to communicate the client with the server. The advantages of using AJAX are: a) being able to show a progress, both of the state, uploading or colorizing, and the percentage of the image uploaded and b) keep the user in the same page, without extended loading times.

[6] Available at `http://startbootstrap.com/template-overviews/grayscale/`

Sending the image to the server is achieved through the Javascript object *FormData*, which enables to do *multipart/form-data* uploads. That means the body of a POST request might have different objects inside it, for example, files. The server then receives the image and colorizes it, converting to grayscale first if necessary. It also automates the L*a*b* conversion.

Sending back the colored image to the client is trickier. The easiest way would be to store the image in the server and then send a link to that image to the client, but that would mean occupying storage in the server, which is limited. HTML5 introduced a URL based method to display images, files and even music. An image can be encoded into an URL by first encoding it in base64, so that it is encoded with printable characters. Then, the URL is formed as: "*data:image/png;base64,*" + "*encoded_base64_image*".

As this encoding works as a URL, it can be placed in the *src* field of an *<img>* node, making it display the image as if it was downloaded from a real URL. Changing this attribute can easily be done using jQuery, we send the encoded URL as the response to the AJAX request that uploads the image, thus making only a single connection in all the process.

6 Results

This section discusses the results obtained from the previously explained architectures and its later implementation. It introduces a standard way to quantitatively measure the quality and also discusses the images from a qualitative, human perceived, point of view. Examples of where the network gives good results and where it fails to produce the expected results will also be given and analysed.

6.1 Measure of quality

It has been said that to measure the error on the network and to make it learn, the euclidean distance between the target image and the obtained image has been used. This is a good function for a neural network as it is convex and can be easily derived, but it might not fit the way a human perceives the reality.

When working with images the quality measurement is usually done with Peak Signal-to-Noise Ratio (PSNR) [31]. This ratio is particularly used when using reconstructions of lossy codecs during image compression or decompression. It makes sense to think of this problem as a reconstruction problem (we are indeed recovering color), thus this measure is suitable. Its main advantage over a bare euclidean distance is that it approximates the human perception and its results are easier to interpret given the original and reconstructed images.

A higher PSNR value means the reconstruction is of better quality, being ∞ the best possible reconstruction and 0 the worst. Figure 17 shows the different values of PSNR into context. The ratios of these images, however, have been obtained from a grayscale image and not an RGB, so they are merely for illustration purposes and should not be directly compared with the RGB use given from here on. PSNR might be calculated by:

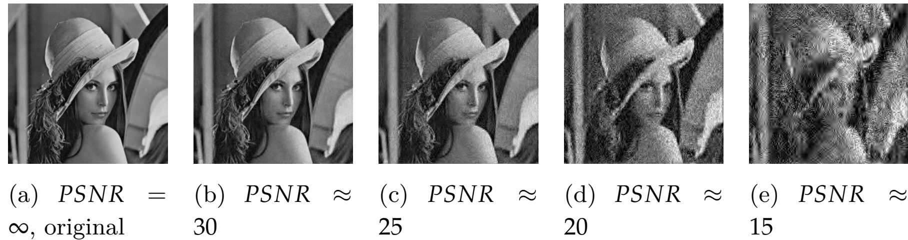
$$PSNR = 10 \cdot log_{10} \left(\frac{MAX_I^2}{MSE} \right)$$

Where MAX_I is the maximum pixel value for the image and MSE is the mean squared error:

$$MSE = \frac{1}{256 \cdot 256 \cdot 3} \sum_{k=0}^{3} \sum_{i=0}^{256} \sum_{j=0}^{256} (y(i,j,k) - \hat{y}(i,j,k))^2 \tag{8}$$

An important note, as seen in Equation 8, is that the comparison is done in the RGB color space. Although we could use the L*a*b* space, doing it in the RGB color space

comes into importance when comparing with other methods, for example state of the art implementations. Also, images should be in the range $[0, 255]$, thus $MAX_I = 255$.



(a) $PSNR = \infty$, original (b) $PSNR \approx 30$ (c) $PSNR \approx 25$ (d) $PSNR \approx 20$ (e) $PSNR \approx 15$

Figure 17: PSNR values for Lena
Image by Rice University, Digital Image Processing ELEC 539

6.2 Datasets

To train and test the framework three completely different datasets have been used. The first one consists of synthetic images generated with the purpose of doing an initial test and verification of the problem. The other two, landscapes and faces, are used to prove the architecture is capable of obtaining good results whatever input is given. Of course, each dataset is trained separately and then tested with images of that same category.

Before showing the datasets results, a special mention must be done. The loss graphic obtained on each dataset will be shown, however, it may be misleading: there is a wide range of images on the datasets, for example, we might find landscapes with a blue sky or orange sunsets. If the network tries to minimize the first, the second would raise, and vice-versa. That is why even though during some iterations the loss might seem to be stuck, the resulting images have notable differences. The main interest on the loss is: a) to see if it seems to converge up to a point and b) to make sure the train and test loss are about the same, so that no overfitting is happening.

6.2.1 Synthetic data

The original problem relies on coloring images of size 256x256, which is more than the usual 28x28 or 80x80 most neural networks use. Approaching the problem by trying to colorize the full 256x256 resolution landscape images would have been a bad idea because:

1. It was uncertain if the problem, colorizing an image, would be able to be solved
2. The time spent to pre-process the original images would have been much greater than the one to generate these ones

Especially because the first reason, it was decided that the initial network should work with images taken from the MNIST[32] dataset. Each number from 0 to 9 was assigned a color and painted to obtain an RGB version of the digit (Figure 18).

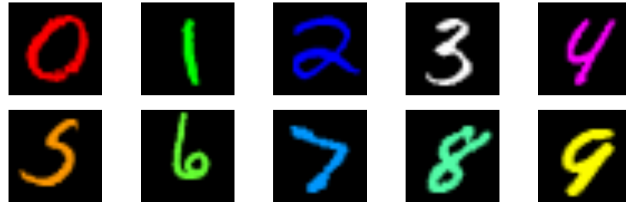


Figure 18: RGB colored MNIST dataset

This set consists of 60.000 training examples and 10.000 validation examples, generated at a resolution of 28x28. It was meant to do quick tests on the architecture and validate that it was producing the expected results.

This dataset was used to test the first architecture explained in section 4.1. As demonstrated, using high resolution images would not work, but the small ones were still useful to clarify some insights of how the process worked.

MNIST is rather easy to classify with good accuracy, for instance a Support Vector Machine alone can achieve less than 2% error rate [33]. Results, as seen in Figure 19, are almost perfect, in the sense that the color is well chosen and applied correctly on the number itself. It is likely that the network has learned to classify numbers and applies the color based on that classification. The most noticeable side effect of this algorithm is that it seems to blur the resulting image, as seen in the samples.

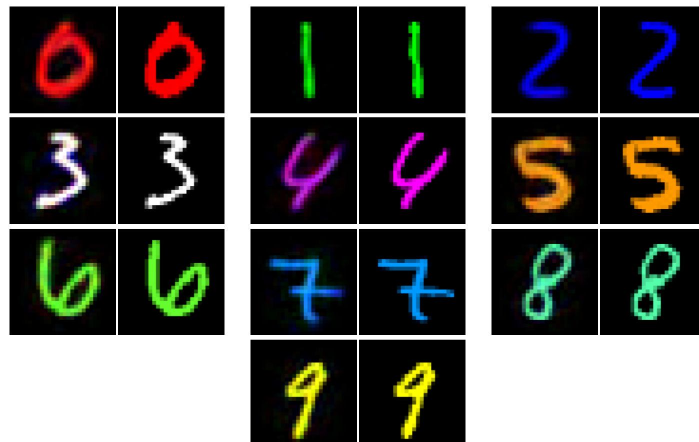


Figure 19: Predicted colorization for each number in the synthetic dataset.
In each block, left is the predicted image and right the original

6.2.2 Landscapes

The first network has been trained on over 25.000 landscape images and tested on a different validation set of 5.000 images. It gives, approximately, an 80/20 train/test split (or hold out) to the dataset.

This dataset was obtained from highly rated websites taken by professional photographers. It is a semi-automatically subset of images selected from a bigger dataset of over 2 million images. Some examples can be seen in Figure 20. It can be easily seen that we can expect a moderated amount of modifications in the images, performed to visually enhance them.

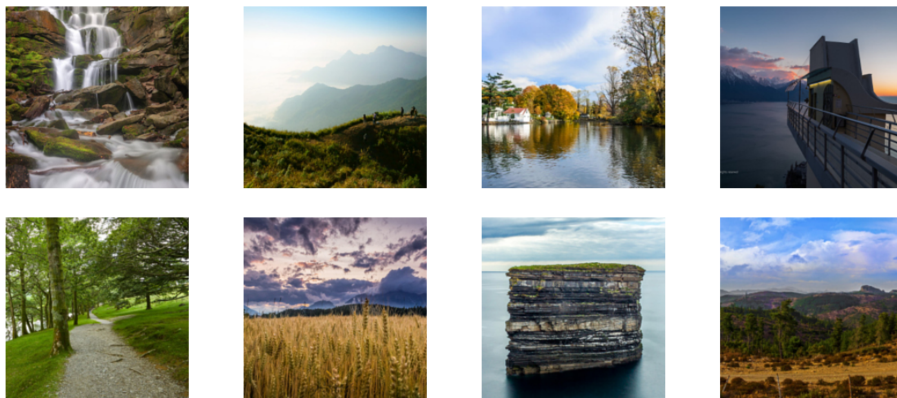


Figure 20: Samples taken from the landscapes dataset

Also, it contains samples with noticeable differences in common patterns. For example, in Figure 21 it can be seen the sky in more than 3 different tonalities. This, as will be explained later in this chapter, might be a problem for the network.

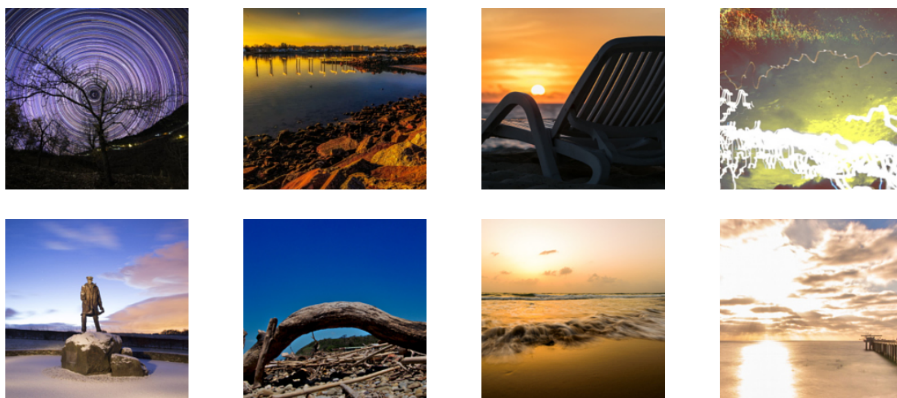


Figure 21: Images with a wide range of color tonalities on the sky

Before qualitatively discussing the actual results, it must be noted that to make 10000 iterations the network spends approximately 5 hours. The network trains with 25000 images, so that means that after those iterations, with batch size 8, it will have seen $10000 \cdot 8/25000 = 3.2$ epochs. Seeing all the images is considered to be an epoch, so that means it will have seen all images 3 times.

In Figure 22 a subset of the train and test loss can be seen. The optimizer used, Adam, achieves to dramatically reduce the error in the first few iterations, as it can be seen in the first huge slope. Then, the network seems to be stuck in a fixed range. While this is true, visually the image changes greatly. The overall loss does not vary because, as it has already been stated, many representations of the same elements exists. When one is minimized, the other ones are maximised. To help the network learn which features are important, so that it can appropriately minimize each texture, a decaying learning rate (Figure 23) has been used. Every 1000 iterations it is exponentially reduced by 0.95.

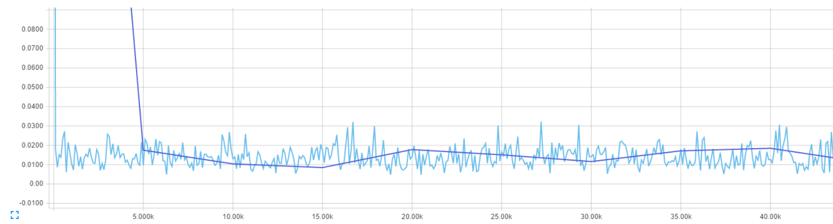


Figure 22: Train (light blue) and test (dark blue) losses of the network

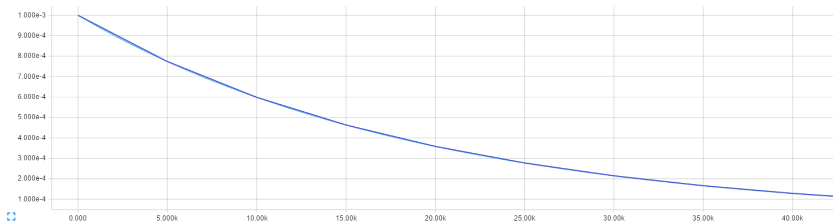


Figure 23: Exponential learning rate decay, 0.95 decay every 1000 iterations

In Figure 24 we can see a comparison at 40000 iterations (12.8 epochs, 20 hours), 80000 iterations (25.6 epochs, 1 day 16 hours) and 120000 epochs (38.4 epochs, 2 days 12 hours). There are a few things to talk about, but here the main discussion will be centered in the effect of letting the network do more iterations. It can be seen in all 4 examples that the more iterations, the more orange they are painted. The network first learns to paint blue skies and then tries to specialize at painting orange skies. The reason for this is simple, there are significantly more examples of blue skies than there are of orange.

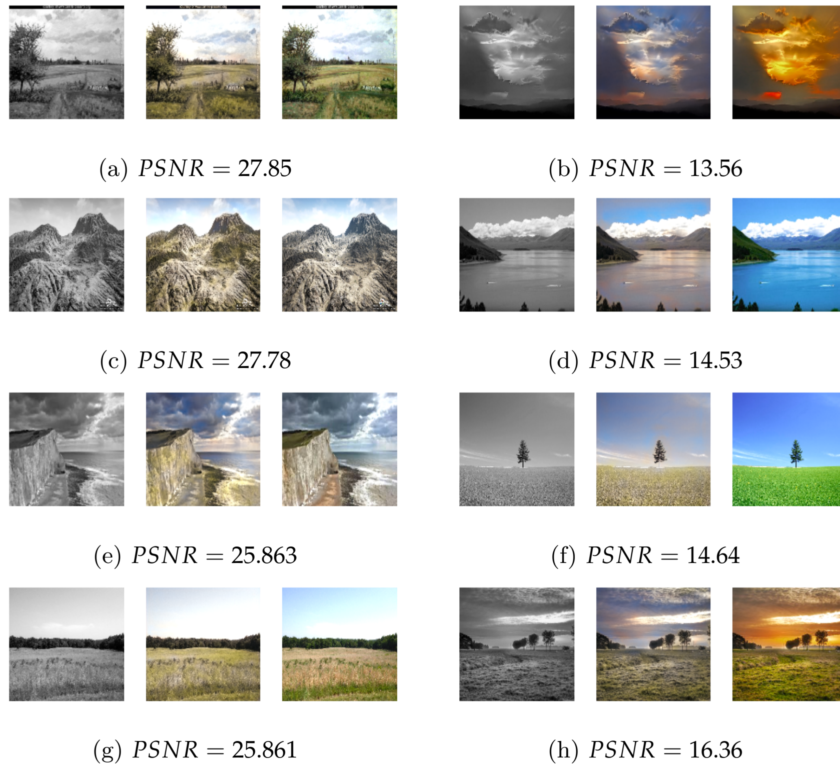
Therefore, it makes sense for the network to first paint them blue, as that would minimize the error. This seems to indicate that if we let the network do more iterations, we would see even more orange tonalities and perhaps they would be better placed. The first column of Figure 24 is a clear case of mispainted orange, but still, there are images where orange is

Figure 24: 4 images colorized along 3 different iterations of the network. The first row is the grayscale input and last one is the original image

correctly placed.

Quantitative analysis: When calculating the PSNR ratio of the landscapes, the average value is 20.79dB for the trained network. Figure 25 show cases where a) the PSNR is high, mainly because the predominant textures are grass, trees, sky and clouds. And b) the PSNR is bad, the original image has vivid colors and the network is unable to reproduce it.

A closer look to the images used for the PSNR reveal that original images with low intensity colors, non saturated, perform much better than those that have intense colors. Also, one might see that the resulting images have not as vivid colors as the original ones.



(a) $PSNR = 27.85$ (b) $PSNR = 13.56$

(c) $PSNR = 27.78$ (d) $PSNR = 14.53$

(e) $PSNR = 25.863$ (f) $PSNR = 14.64$

(g) $PSNR = 25.861$ (h) $PSNR = 16.36$

Figure 25: Images with the best PSNR on the left and with the worst PSNR on the right. Each block, columns from left to right: Grayscale image, Colorized image, Ground truth

The problem, even if it seems so, comes not from the saturation, but rather from the loss function used. As said, the euclidean distance is the measure of error on the trained images, but the problem here is that we have many colors for the same object (for example, blue and orange skies or even water of multiple colors).

To minimize the error the network will, intuitively, search for the mean color between all the representations, as that would minimize the global error amongst them all, and not with a single one only. These results can be easily confirmed by looking at the images in Figure 26: places where the network knows not how to color (because it has seen too many representations with too many different colors) are either painted sepia or with the mean color which represents that texture. For example, it can be easily seen that all grass is painted the same green, dark and not saturated, color.

Figure 27 shows more examples of colorized images. It can be appreciated images with impressing results but also some images where the networks fails to partly colorize some grass or water, leaving them almost gray.

(a) Predominant sepia color on the resulting image

(b) Sepia colored water

(c) Non-saturated colors

Figure 26: Common cases where the network fails.
Columns from left to right: Grayscale image, Colorized image, Ground truth

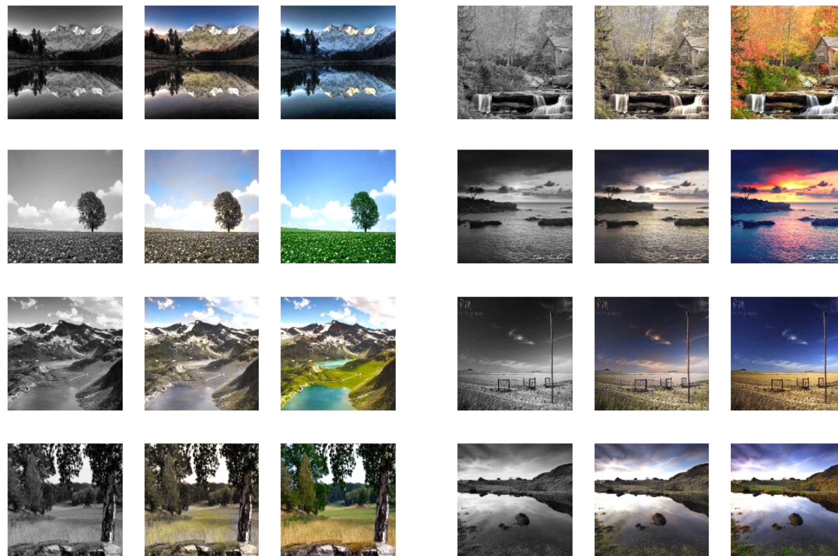


Figure 27: Good and bad examples of colorized images
Per block, columns from left to right: Grayscale image, Colorized image, Ground truth

6.2.3 Faces

The Faces in the Wild [34] dataset has been used for this purpose. This dataset, as already said, serves to prove that the network architecture is able to generalize to new problems without needing any change in its structure, but also to see whether it is able to learn human characteristics or not. Some samples from the set can be seen in Figure 28.

Figure 28: George W. Bush faces taken from the Faces in the Wild dataset

Figure 29 shows some more examples taken from the dataset. There are important features to note from these ones:

1. Not all images are the same size. They are all, therefore, normalized to 86x86, which is the most common size amongst all of them
2. There are some images which are not faces. We can see some bodies and even cropped faces
3. Grayscale images can also be found on the dataset. It would be better if all of them were colored

It consists of 30.281 faces, ordered by clusters of people. When working with neural networks it is important to have a random order of images, so that when performing stochastic gradient descend by minibatches the network sees different examples, thus having a correct expected value for the gradient direction. After being randomly shuffled, they have been split into 80% for training and 20% for testing.

Although the network architecture is the same, it is trained on images considerably smaller, of size $86x86$. The network does 10000 iterations much faster than before, taking only about 32 minutes. As this network is faster to train it will only be presented at 120000

Figure 29: Random images selected from the Faces in the Wild dataset

iterations. Again, using a batch size of 8 that makes a total of 39.6, almost 40, epochs. This is about the same number of epochs the landscapes version does.

Qualitative analysis: Figure 30 shows some cases selected from the test images where the network performs amazingly good. It can be seen that it has learned to:

1. Distinguish faces color but not based from the lighting, as most have the same but have different results, but from the human characteristic itself. We could even say it has learnt the racial differences in human faces.
2. Differentiate men from women. It can be seen that none of the men in the figure has its lips painted, while women have their lips of an appreciably different color.
3. Detect faces, because only the face region is consistently painted. It fails to paint backgrounds, but that is understandable and even expected because the images given for training were, mostly, faces.

Of course, there are also some other images (Figure 31) the network does not accurately colorize. It is clear that some cases, like the first image, fail because the image has not a face in it, but generally talking, when it fails it seems to be due to illumination issues (rows 2 and 3), because the orientation and illumination are strange (row 4) or due to a color averaging problem as we have already seen with the landscapes dataset (rows 5 and 6).

Unexpectedly, although the network has been trained with faces, it is able to colorize some other parts of the human body, Figure 32. This figure also gives the intuition that the network has learnt that the face is not always centered, as in the second and third rows, the shirts are not painted.

Quantitative analysis: The average PSNR for the network trained for 120000 iterations

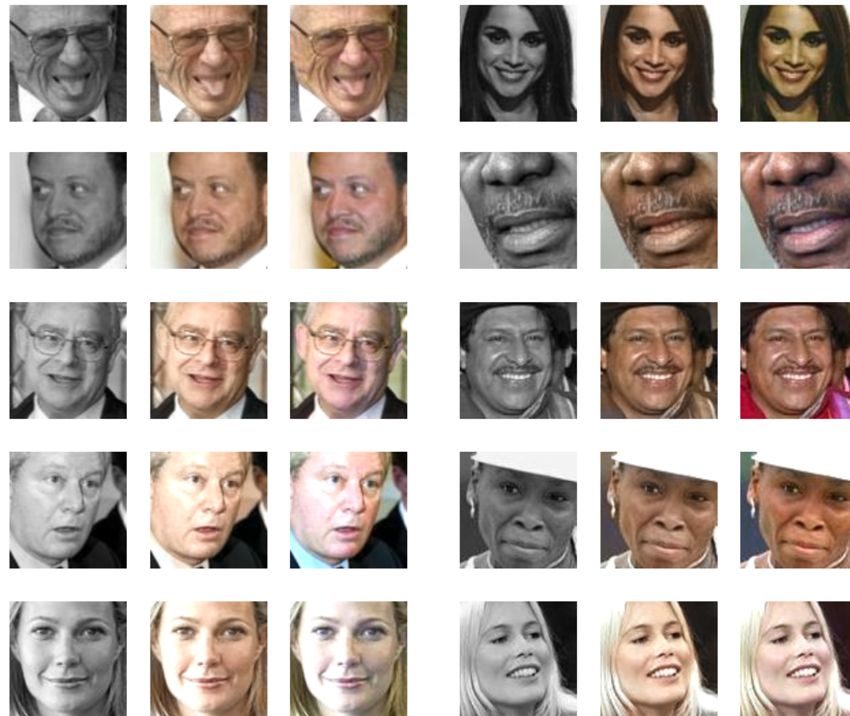


Figure 30: Samples of colored faces.
In blocks of 3, columns from left to right: Grayscale image, Colorized image, Ground truth

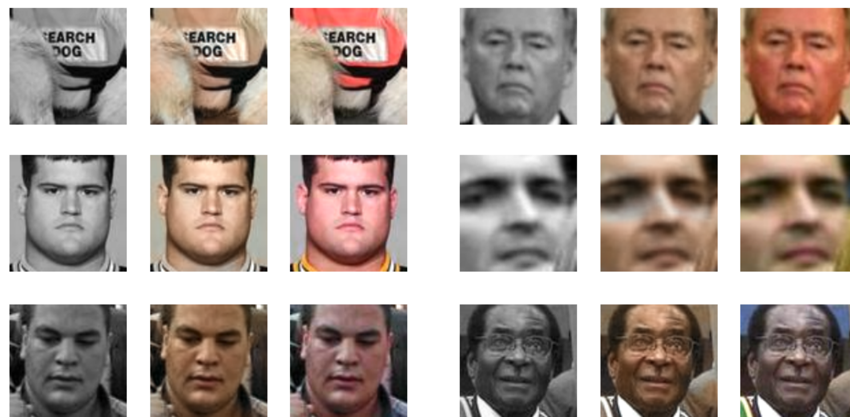
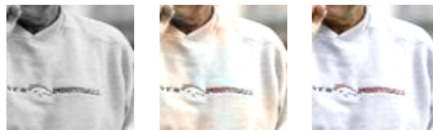


Figure 31: Images the network fails to properly colorize
Each block, columns from left to right: Grayscale image, Colorized image, Ground truth

and tested on the whole test dataset, obtains a value of about 27dB. This value is high when compared with the other values obtained in other papers and researches.

(a) Correctly painted hand

(b) Painted torso but omitted shirt



(c) The network paints the neck but not the shirt

Figure 32: Behaviour of the network when presented with images without faces. Columns from left to right: Grayscale image, Colorized image, Ground truth

In Figure 33 it can bee seen side by side which images obtained the best PSNR ratios and which ones the worst. Closely examining them, it could be argued that the PSNR ratio here is not correctly representing the problem: colorizing faces. The PSNR is taking into account the whole image and not the face itself, and as can be seen in the figure the images obtaining bad results are not due to the face itself but most likely due to the background color being completely different. Figure 34 does a colorization that, looking only at the face, is subjectively worse than the here seen, and yet obtains a PSNR higher than those *bad cases* faces.

Seeing these images and following this reasoning, the real PSNR value for the faces dataset would be higher, maybe even close to 30dB.

(a) $PSNR = 35.03$ (b) $PSNR = 17.44$

(c) $PSNR = 34.69$ (d) $PSNR = 18.03$

(e) $PSNR = 34.52$ (f) $PSNR = 18.33$

(g) $PSNR = 34.01$ (h) $PSNR = 18.80$

Figure 33: Images with the best PSNR on the left and with the worst PSNR on the right. Each block, columns from left to right: Grayscale image, Colorized image, Ground truth

Figure 34: Image subjectively worst with $PSNR = 20.95$
Columns from left to right: Grayscale image, Colorized image, Ground truth

7 Conclusions

This section is divided into two subsections. The first one explains the conclusions given the results shown in the section above, discussing whether the initial objectives have been achieved or not. The last one proposes new approaches to solve the problem and sets a continuation to the work already done.

7.1 Research conclusions

The results obtained are satisfactory, concretely:

1. **Colorization**: The network is able to colorize images, it produces images which are colored, beautiful and consistent, as shown and explained in previous sections. It is also able to work with any category or class of image as long as the network is first trained on that same class. It could also be trained on multiple categories at once and it would be available to predict all of them equally. Of course, this last approach would involve more problems caused by the euclidean distance averaging the results.

   From the quantitative point of view, we have seen that the PSNR gives an accurate value for the perceptible differences on the reconstructed image and the original one. The landscapes dataset achieves about 20dB, but mostly due to the average problem and the textures with multiple representations, like the sky. The faces dataset achieves a much better value of 27dB, and would be better if only the faces would be taken into account and not the background.

   Qualitatively, most landscapes images could pass as authentic images if the user would not have seen the original. It correctly paints all textures and objects, but fails to do so in vivid colors. It should be taken into account, however, that given a grayscale image the network has no way of telling if, for example, the sky should be blue, orange or purple. That is why perceptually, the colorization is acceptable and valid.

   The same applies to the faces dataset. All faces tested are painted consistently and, more important, the color they should be. That means the network seems to have learnt to differentiate races and genders and is able to apply the correct tonality in each case.

   The synthetic dataset has served its purpose too, testing and validating the first architecture to know its limitations and how to improve it.

2. **Research**: As stated on Section 1.2, one of the purposes was to contribute in the research. It could be considered as successfully done too, as novel techniques have been applied to implement and further developed in the colorization field.

   Residual networks have been used to further improve the results, combined with the hyperbolic tangent activations and the L*a*b* color space has been key to obtain the final outcome.

Both Caffe and Tensorflow, pioneer frameworks for deep learning, have been used. Although some small differences between them have been observed, at the end results were good whichever was used. It required, however, appropriately tuning all hyperparameters, because they did not coincide.

3. **Online website**: The user is able to test the algorithm on images of his own. It is easy to use, widely available and works perfectly by correctly integrating with the network trained with Tensorflow and producing the expected colorized images.

4. **Overall objectives**: The method introduced is, as was stated:

   (a) Fast: If using a GPU NVIDIA Tesla K40, images are obtained in approximately 0.5 seconds. An NVIDIA GTX 970 also produced an image in about 0.7 seconds. Using an Intel N3700 CPU raises this number to an average of 2.3 seconds.

   Results would improve if better hard drives were used, for example by using solid state drives. RAM and CPU/GPU also takes an important role, as well as network connection when using the provided website.

   (b) User agnostic: This method does not require the user to hand-craft features nor to supply more images than the one desired to colorize. By simply submitting an image the user obtains the final result and no interaction is required, thus achieving the desired objective.

   (c) User friendly: Alongside with the website, Python code and IPython notebooks are supplied to show the actual implementation and to make available methods for testing purposes.

7.2 Future work

Without doing fundamental modifications to the current architecture, some ideas that could be tested and researched are:

1. Using a different loss function. It is clear that the problem comes from the loss function we are using, so changing it or helping it with other functions might yield good results. For example, when working with Caffe, the CIEDE2000 [35] L*a*b* similarity function was used.

   The main advantage over a plain euclidean distance is that it has two terms to correct the hue and the saturation of the image. Tweaking them showed better results on some images but the overall result was worse, more areas were painted sepia.

   This function also has a high performance cost, due to the high amount of operations it does. Results prove not to be good enough to compensate for this huge cost. PSNR might also be a good option.

2. Changing the problem to be a classification task instead of a regression by performing a per-pixel classification against the target color. Some of the cited papers have already

tried it and the colors have indeed more intensity, but new problems arise, such as inconsistencies along textures or even unexpected colors on some objects.

3. Trying to help the network with more information. For example, in this project it has been tried to use the Gram matrix, like used in Style transfer networks [36], but the overall PSNR was better without it. Similarly, another way of adding some more global information about the image, tonalities and global saturation might be by using the histogram of the images. This approach has been tested but further research and time is required to be able to extract conclusions.

In the search for better results, it is worth exploring other solutions aside from CNNs. For example, the following types of artificial neural networks are becoming more popular as days pass:

1. Variational Autoencoders [37] have already been used to generate artificial human faces. From a theoretical point of view they should be capable of colorizing images, but they are still hard to train as there is not a way to have a completely stable gradient yet.
2. Generative Adversarial Nets [38] and specifically DRAW architectures [39] have also been used to generate images over the MNIST dataset.
3. Some uprising types of networks which could eventually be useful for this problem are Generative Moment Matching Networks [40] and Probabilistic Program Induction [41] based networks.

It might also be a good idea to ask more complex questions than colorization alone: can the network learn not only to colorize but also to apply a different style to the image? For example, we might want a beach to be painted with daylight or with the orange tones of a sunset. One interesting approach might be to further explore the gram matrices, as used in Style Transfer, or by generating intermediate textures as done in Texture Networks [42].

References

[1] D. Mishkin, N. Sergievskiy, and J. Matas, "Systematic evaluation of CNN advances on the ImageNet", *arXiv preprint arXiv:1606.02228*, 2016 (cit. on p. 1).

[2] M. Kawulok, J. Kawulok, and B. Smolka, "Image Colorization Using Discriminative Textural Features.", in *MVA*, Citeseer, 2011, pp. 198–201 (cit. on pp. 2, 4).

[3] B. Sheng, H. Sun, S. Chen, X. Liu, and E. Wu, "Colorization using the rotation-invariant feature space", *IEEE computer graphics and applications*, no. 2, pp. 24–35, 2011 (cit. on pp. 2, 4).

[4] R. Irony, D. Cohen-Or, and D. Lischinski, "Colorization by example", in *Eurographics Symp. on Rendering*, Citeseer, vol. 2, 2005 (cit. on pp. 2, 4).

[5] R. Dahl. (2016). Automatic Colorization, [Online]. Available: `http://tinyclouds.org/colorize/` (cit. on p. 4).

[6] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition", *arXiv preprint arXiv:1409.1556*, 2014 (cit. on p. 4).

[7] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database", in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, IEEE, 2009, pp. 248–255 (cit. on p. 4).

[8] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition", *arXiv preprint arXiv:1512.03385*, 2015 (cit. on p. 4).

[9] Z. Cheng, Q. Yang, and B. Sheng, "Deep Colorization", in *Proceedings of the IEEE International Conference on Computer Vision*, 2015, pp. 415–423 (cit. on p. 4).

[10] R. Zhang, P. Isola, and A. A. Efros, "Colorful Image Colorization", *arXiv preprint arXiv:1603.08511*, 2016 (cit. on p. 4).

[11] G. Larsson, M. Maire, and G. Shakhnarovich, "Learning Representations for Automatic Colorization", *arXiv preprint arXiv:1603.06668*, 2016 (cit. on p. 4).

[12] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity", *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943 (cit. on p. 6).

[13] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines", in *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, 2010, pp. 807–814 (cit. on p. 10).

[14] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models", in *Proc. ICML*, vol. 30, 2013, p. 1 (cit. on p. 10).

[15] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)", *arXiv preprint arXiv:1511.07289*, 2015 (cit. on p. 10).

[16] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification", in *Proceedings of the IEEE International Conference on Computer Vision*, 2015, pp. 1026–1034 (cit. on p. 10).

[17] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting", *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014 (cit. on p. 11).

[18] M. Matsugu, K. Mori, Y. Mitari, and Y. Kaneda, "Subject independent facial expression recognition with robust face detection using a convolutional neural network", *Neural Networks*, vol. 16, no. 5, pp. 555–559, 2003 (cit. on p. 12).

[19] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift", *arXiv preprint arXiv:1502.03167*, 2015 (cit. on p. 14).

[20] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks", in *Advances in neural information processing systems*, 2012, pp. 1097–1105 (cit. on pp. 15, 21).

[21] R. K. Srivastava, K. Greff, and J. Schmidhuber, "Highway networks", *arXiv preprint arXiv:1505.00387*, 2015 (cit. on p. 15).

[22] S. Hochreiter and J. Schmidhuber, "Long short-term memory", *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997 (cit. on p. 15).

[23] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation", *arXiv preprint arXiv:1406.1078*, 2014 (cit. on p. 15).

[24] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei, "Large-scale video classification with convolutional neural networks", in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, 2014, pp. 1725–1732 (cit. on p. 20).

[25] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions", in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 1–9 (cit. on p. 21).

[26] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images", 2009 (cit. on p. 23).

[27] A. Ford and A. Roberts, "Colour space conversions", *Westminster University, London*, vol. 1998, pp. 1–31, 1998 (cit. on p. 23).

[28] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional Architecture for Fast Feature Embedding", *arXiv preprint arXiv:1408.5093*, 2014 (cit. on p. 27).

[29] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin

Wicke, Yuan Yu, and Xiaoqiang Zheng, *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*, Software available from tensorflow.org, 2015 (cit. on p. 27).

[30] D. Kingma and J. Ba, "Adam: A method for stochastic optimization", *arXiv preprint arXiv:1412.6980*, 2014 (cit. on p. 28).

[31] Q. Huynh-Thu and M. Ghanbari, "Scope of validity of PSNR in image/video quality assessment", *Electronics letters*, vol. 44, no. 13, pp. 800–801, 2008 (cit. on p. 38).

[32] Y. LeCun, C. Cortes, and C. J. Burges, *The MNIST database of handwritten digits*, 1998 (cit. on p. 40).

[33] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition", *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998 (cit. on p. 40).

[34] T. L. Berg, A. C. Berg, J. Edwards, and D. Forsyth, "Who's in the picture", *Advances in neural information processing systems*, vol. 17, pp. 137–144, 2005 (cit. on p. 46).

[35] G. Sharma, W. Wu, and E. N. Dalal, "The CIEDE2000 color-difference formula: Implementation notes, supplementary test data, and mathematical observations", *Color Research & Application*, vol. 30, no. 1, pp. 21–30, 2005 (cit. on p. 52).

[36] L. A. Gatys, A. S. Ecker, and M. Bethge, "A neural algorithm of artistic style", *arXiv preprint arXiv:1508.06576*, 2015 (cit. on p. 53).

[37] D. P. Kingma and M. Welling, "Auto-encoding variational bayes", *arXiv preprint arXiv:1312.6114*, 2013 (cit. on p. 53).

[38] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets", in *Advances in Neural Information Processing Systems*, 2014, pp. 2672–2680 (cit. on p. 53).

[39] K. Gregor, I. Danihelka, A. Graves, D. J. Rezende, and D. Wierstra, "DRAW: A recurrent neural network for image generation", *arXiv preprint arXiv:1502.04623*, 2015 (cit. on p. 53).

[40] Y. Li, K. Swersky, and R. Zemel, "Generative moment matching networks", in *International Conference on Machine Learning*, 2015, pp. 1718–1727 (cit. on p. 53).

[41] B. M. Lake, R. Salakhutdinov, and J. B. Tenenbaum, "Human-level concept learning through probabilistic program induction", *Science*, vol. 350, no. 6266, pp. 1332–1338, 2015 (cit. on p. 53).

[42] D. Ulyanov, V. Lebedev, A. Vedaldi, and V. Lempitsky, "Texture Networks: Feed-forward Synthesis of Textures and Stylized Images", *arXiv preprint arXiv:1603.03417*, 2016 (cit. on p. 53).