TÉCNICO LISBOA

Convolutional Neural Networks versus Feature Extraction

Bruno Miguel Junceiro Sousa

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Andreas Miroslaus Wichert

Examination Committee

Chairperson: Prof. António Manuel Ferreira Rito da Silva
Supervisor: Prof. Andreas Miroslaus Wichert
Member of the Committee: Prof. Jacinto Carlos Marques Peixoto do Nascimento

October 2021

Abstract

Although Convolutional Neural Networks revolutionized the Deep Learning and computer vision fields, state-of-the-art models have been suffering from some problems, due to their increasing depth. These problems are: requirement of enormous labeled datasets for their learning; requirement of huge computational power to process those amounts of data; degradation of their generalization ability the deeper they become. With the purpose of solving those problems, we present a classification architecture which focus on the extraction of the color, texture and shape information from images, using computer vision techniques. This information will be aggregated based on their feature type (modality) and used to train independent models. Each modality will be learned by three different models, with the intention of improving the generalization of the proposed solution. We chose the best performing model to represent the three modalities. From these models, we will use the resulting Softmax probabilities assign to each modality and fuse them using the Dempster-Shafer's rule of combination, as the multimodal fusion approach. To train and experiment on the proposed solution we used the Fruits 360 dataset and evaluated the performance on said experiments with the loss and accuracy metrics. The implemented multimodal classification model was compared with two baseline CNNs, which were trained using the same configurations: batch size, epochs, activation and loss functions. The baseline models achieved accuracies of 93.46% and 93.34%, while the proposed solution achieved 94.80% for the fusion of the color, texture and shape modalities. We were able to improve over this result, using the Dempster-Shafer's rule of combination with only the color and texture modalities, which resulted in a accuracy of 97.40%.

Keywords

Feature Extraction, Convolutional Neural Networks, Image Classification Tasks, Multimodal Fusion, Dempster-Shafer Theory, Deep Learning

Resumo

Embora as Redes Neurais Convolucionais tenham revolucionado as áreas de Aprendizagem Profunda e de visão computacional, os modelos de última geração têm sofrido alguns problemas devido ao aumento da sua profundidade. Problemas como a necessidade de vastos conjuntos de dados etiquetados, a necessidade de bastante poder computacional para processar esses dados e a degradação da capacidade de generalização destes modelos. Com o objetivo de resolver esses problemas, apresenta-se uma arquitetura de classificação que privilegia a extração das características de cor, textura e forma de imagens, utilizando técnicas de visão computacional. Estas características serão agregadas consoante a sua modalidade e usadas para treinar modelos independentes. As modalidades serão aprendidas por três modelos diferentes, com o intuito de melhorar a generalização da solução proposta, e será escolhido um modelo para representar cada modalidade, com base na sua performance. Destes modelos obtêm-se probabilidades Softmax, associadas a cada modalidade, e fundem-se esses valores através da regra de combinação de Dempster-Shafer. Para treinar e avaliar a solução proposta, usou-se o conjunto de dados Fruits 360 e avaliou-se o desempenho com as métricas de perda e exatidão. O modelo de classificação multimodal implementado foi comparado com duas CNNs treinadas usando configurações semelhantes: tamanho de lote, épocas de treino, funções de ativação e perda. O desempenho destes modelos foi de 93.46% e 93.34%, e para a solução proposta foi de 94.80%, para a fusão das três modalidades. Conseguiu-se melhorar esse resultado usando a regra de combinação de Dempster-Shafer apenas com as modalidades de cor e textura, resultando numa exatidão de 97.40%.

Palavras Chave

Extração de Características, Redes Neurais Convolucionais, Tarefas de Classificação de Imagem, Fusão Multimodal, Teoria de Dempster-Shafer, Aprendizagem Profunda

Contents

List of Figures

List of Tables

1

Introduction

Deep Learning neural networks, specifically Convolutional Neural Networks, are major contributors to the current state of the computer vision field. Since the first backpropagated CNN [LeCun et al., 1989], we have been solving image recognition tasks of increasing complexity.

Starting with the Neocognitron [Fukushima, 1980], in 1980, models motivated by the notions presented by Hubel and Wiesel [Hubel and Wiesel, 1962] [Hubel and Wiesel, 1968], such as the receptive fields in the visual cortex of mammalian brains, have been constantly improving the field of digital image processing.

Some of these models, like GoogLeNet [Szegedy et al., 2015] and ResNet [He et al., 2016a], were groundbreaking to state-of-the-art CNNs. Due to these networks, we are currently able to create Convolutional Neural Networks of enormous depth, capable of learning datasets with millions of samples, assigned with thousands of different labels.

Although all the important innovations introduced by the mentioned models, modern Convolutional Neural Networks are facing some shortcomings. Generally, the depth of a neural network is associated with the amount of labeled samples and computational power required for it to successfully learn. So, as expected, the continuous depth increase of CNNs is starting to cause problems, such as lack of

labeled datasets with the amount of samples required and hardware limitations. The computational power required to train state-of-the-art CNNs worsened with each increase of the CNN's depth. This problem may not be of relevance for the creators of the GoogLeNet and ResNet (Google and Microsoft, respectively), because these companies are in the vanguard of computer science and, consequently, have the necessary resources to train these CNN's at their disposal. However, smaller companies and research teams do not have those same resources, which limits their efficient use of networks with this increasing depth.

Additionally, despite deeper CNNs being able to learn more complex features, an excessive depth can cause the degradation of the network's generalization ability.

1.1 Objectives and Proposed Solution

Considering the aforementioned problems, a possible solution would be to resort to computer vision techniques in conjunction with simpler models, such as shallow Neural Networks, in order to extract only certain features from the datasets, making the data samples smaller and easier to process by the model.

We propose a classification architecture, composed of three modules: Feature Extraction, Feature Learning and Multimodal Fusion; that intends to be a solution to the problem of the computational power required to train deeper CNNs.

The proposed model aims to use simpler Neural Network models to learn feature vectors of considerably lower dimensionality than the original images.

The Feature Extraction module is where the images will be processed and their respective features extracted, through use of computer vision techniques, and organized into feature vectors, according to the modality which they represent. The features we intend to extract from the images are the color, texture and shape information. The color information will be extracted using color histograms, to represent the pixel intensity for each color channel, and will be represented in a feature vector that consists of a normalized concatenation of the three color histograms. The texture information will be extracted computing the image gradient and will be represented using a Histograms of Oriented Gradients, which consists of an aggregation of multiple local gradient histograms computed for each region of the image. The shape information will be extracted using several methods, such as edge and corner detection. This information will the be combined in a feature vector we designated as Shape Map.

On the Feature Learning module, we will train several Neural Network models for each modality extracted on the previous module. These Neural Networks will be trained over the same circumstances and we will choose the best performing model for each modality. The reason for having Neural Networks focus specifically on learning only one feature type is the fact that this will greatly increase the generalization of the each model. This approach was inspired by how the human eye has different types of cells

to process color (Cones) and light intensities (Rods).

After choosing the model that will represent the Color, Texture and Shape modalities, we will use those models to predict the labels of the test set, in order to create probabilistic vectors with a dimension equal to the number of labels present in the dataset. These probabilistic vectors will then be combined in the Multimodal Fusion module, using the Dempster-Shafer's theory of evidence [Shafer, 1976]. This multimodal fusion technique will use the probabilistic vectors as mass functions, which will be combined resorting to the Dempster-Shafer's rule of combination. From the final combined mass functions, we will finally obtain the predicted labels for the samples in the test set.

Considering the generalization problem, we will also experiment on the Multimodal Fusion module with combinations of only two modalities, leaving one out at a time. With to this approach, we can evaluate if our solution is capable of generalize for samples without considering all their respective features.

We intend to also create some Convolutional Neural Networks, with the purpose of having a baseline to compare the results obtained through the Multimodal Fusion classification.

1.2 Evaluation Methodology and Dataset

To evaluate both the proposed solution and the Convolutional Neural Networks used as a baseline, we will resort to metrics such as accuracy and loss. These metrics are of great importance to analyze the training experiments that we will perform on these models.

For the Multimodal Fusion module we will also resort to the precision and recall metrics, which give us a better understanding of how the models are performing, at a label level.

The implementation of the proposed classification architecture, and respective experiments, will be supported by the use of the Fruits 360 dataset [Mureşan and Oltean, 2017]. This is a considerably large dataset, with a total of 90,380 images distributed over 113 labels.

1.3 Thesis Outline

The rest of the document is organized in the following chapters:

- **Technical Background** (Chapter 2) - Description of the main concepts that support both the networks explored in the Related Work and the used in the proposed classification architecture. These concepts are aggregated into four topics: "Feature Extraction", "Machine Learning", "Deep Learning" and "Regularization";
- **Related Work** (Chapter 3) - Exploration of the most remarkable models that led to the creation of

the first Convolutional Neural Networks (Section “Road to Convolutional Neural Networks”). Presentation of some of the most groundbreaking CNNs and description of their respective innovations to the field (Section “Convolutional Milestones”). Identification of the problems of state-of-the-art CNNs (Section “Main Shortcomings”);

- **Implementation** (Chapter 4) - Detailed exploration of the implementation of each module from the proposed classification architecture;
- **Experimental Evaluation** (Chapter 5) - Definition of the evaluation metrics intended to evaluate performance the Neural Network models. Exploration of the several experiments performed during the implementation of the proposed solution. Comparison with models trained over the same dataset;
- **Conclusion** (Chapter 6) - Summary of the implementation and discussion of the results. Presentation of possible future work.

2

Technical Background

In this chapter, we will present the concepts which serve as a basis for this thesis. These concepts are divided into four sections:

- **Feature Extraction** (Section 2.1) - definition of the datasets' domain, in the context of image classification problems. Exploration of the most relevant features to be extracted from this type of data;
- **Machine Learning** (Section 2.2) - definition of the most important concepts, such as Artificial Neural Networks. Exploration of the different types of layers (and artificial neurons), the learning process and the major functions used by these networks;
- **Deep Learning** (Section 2.3) - definition of concepts such as Deep Neural Networks and Convolutional Neural Networks. Exploration of the main aspects that differentiate an Artificial Neural Network from a Convolutional Neural Network;
- **Regularization** (Section 2.4) - definition of regularization and justification of its importance. Exploration of the most commonly used regularization techniques in neural networks.

2.1 Feature Extraction

Feature extraction is the process of retrieving the most relevant information (main features) from a set of raw data (dataset). The relevance of the features is determined by the context of the problem we are focusing on. The features of each data sample are generally represented by a feature vector.

The process of extracting features from a dataset can be related to the goal of compression algorithms [Wichert, 2015], since both seek to reduce the information that represents a certain sample, while maintaining its relevant attributes. In data science, this process is known as dimensionality reduction.

The principal benefit of performing feature extraction on a supervised learning scenario is that it can greatly reduce the time required by the classifier to learn the data. Other than this, the selection of good features may improve the classifier's performance, since it can transform the data, making it linearly separable. An optimal selection of features is a subset of the initial attributes that maximizes the score achieved by the classifier.

Since the data that we will take into account consists only of images, we will just consider features generally used in image processing problems. These features are the ones which best describe the most important properties of an image, such as color, texture and shape.

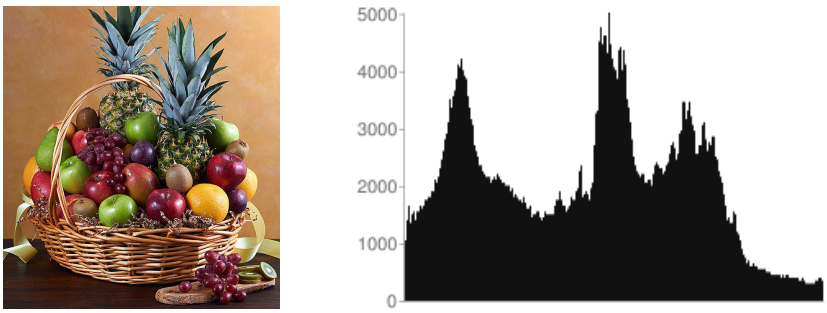
2.1.1 Color as a Feature

The color can be used to differentiate objects present in an image. The easiest way to use color as a feature is to compute the color histogram of an image. This can be useful to identify and segment possible objects from one another and even from their surrounding background.

We can compute a color histogram from both a color image or a grayscaled one. In the grayscale case, we will obtain a vector where every position represents the pixel frequency for each respective level of gray (Figure 2.1(b)). Usually, a grayscale image has an 8-bit pixel depth, which provides a range of 256 levels of gray, thereafter the vector will have the same amount of positions.

In the Figure 2.1, we can observe an image and its respective grayscale histogram. This histogram would be used as a feature vector to represent the color, or more precisely the levels of gray, extracted from the image, shown in Figure 2.1(a).

Focusing on the case where the image has color, its color may be represented by a 3 channel color model. As opposed to the color models preferred in the compression algorithms, such as the YCbCr model in the JPEG compression method, which take advantage of the human eye perceptions of luminance and color to loose the less noticeable information, the computations performed for the color histogram tend to benefit from models such as the RGB. The RGB color space separates the colors in 3 main components: red, green and blue. Each of these components will originate their own histogram, which can be used as 3 feature vectors or combined into a single vector, resulting from the sum of the



(a) RGB image with size (690x755).

(b) Grayscale histogram of the image.

Figure 2.1: Example of a grayscale histogram.

previous 3. Similarly as the grayscale images, each channel from the RGB is restrained by a range of 256 levels, caused by the 24-bit pixel depth images represented in this color model.

If necessary, we could divide the original image in *k* regions and compute the color histogram for each one. Although this grants a higher granularity of the information extracted, it can also result in an unnecessary amount of features which completely defeats the purpose of performing feature extraction.

2.1.2 Texture as a Feature

The texture of an image is a set of metrics, resulting from image processing techniques, that tries to quantify the noticeable texture present in the image. Due to the lack of a concrete definition of image texture, we will consider Linda G. Saphiro and George C. Stockman's definition: "*image texture provides information about the spacial arrangement of color or the intensity of an image or selected region of an image*" [Shapiro, 2001].

There are two approaches to image texture analysis: Structured Approach and Statistical Approach.

The first approach is mostly used in the analysis of artificial textures, that is, textures found in computer generated images. According to this approach, an image is composed by a set of primitive texels that appear regularly or repeatedly throughout the image. A texel is the basic unit of a texture map, used in computer graphics.

The second approach is most widely used for digital images captured from the real world. This approach considers texture as a quantitative measure of the distribution of intensities found in the image.

In the Statistical Approach, we use edge detection algorithms to transform an image into a set of curves or edges. These algorithms are used to find the gradient of an image, which is obtained by the derivative of a 2-dimensional continuous function $f(x, y)$ [Paulus and Hornegger, 2003]. The gradient will identify edges based on the variations and discontinuities in the intensity of a grayscaled image. This gradient contains the information about the direction (2.1) and the magnitude (2.2) of the edge, for each

pixel in the image. In the end, the extracted features will be represented in a vector of *k* gradients, where *k* corresponds to the number of pixels from the original image.

$$\bigtriangledown f(x,y) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right) \tag{2.1}$$

$$\|\bigtriangledown f(x,y)\| = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2} \tag{2.2}$$

Another technique used in this approach is to detect specific types of textures recurring to masks. This technique consists in using the four mask vectors, purposed by Laws [Laws, 1980], for detecting textures associated with levels, edges, spots or ripples. These masks are applied to the image both horizontally and vertically.

2.1.3 Shape as a Feature

The shape information extracted from an image is defined by the contours found by the edge detection algorithms or by application of color segmentation.

The most significant shape to be extracted is the presence of corners in the edges previously detected. We represent a corner with the gain of information associated with a change of the direction (angle) between two edge points. The more curved the angle, the more information it yields. The corner information is represented in bits.

If we think of the Unit Circle, the Information Gain for a straight horizontal line and a straight vertical line is

$$I\left(\frac{\pi}{\pi}\right) = 0\ bits \text{ and } I\left(\frac{\pi/2}{\pi/2}\right) = 0\ bits, \tag{2.3}$$

respectively. The computed gain of information is 0, since the angle remains unchanged for both cases.

Using the same method, the Information Gain for a line that goes from vertical to horizontal is

$$I\left(\frac{\pi}{\pi/2}\right) = 1\ bit. \tag{2.4}$$

On the other hand, if the line goes from horizontal to vertical, its Information Gain is

$$I\left(\frac{\pi/2}{\pi}\right) = -1\ bit. \tag{2.5}$$

Through this, we can detect and extract the corners associated with these changes of direction formed by adjacent edges.

After extracting the corners, the image is now described as a histogram of corners, which will be used as the feature vector that best represents the shapes in the image.

The contours can also be represented in a 2-dimensional vector, where each section of a given contour is represented by the radial distance to the origin and the counterclockwise angle, in a Cartesian coordinate system. This approach results in the image being perceived as a vector of polar coordinates.

2.2 Machine Learning

Machine Learning techniques belong to a subfield of Artificial Intelligence and their purpose is to perform a task or set of tasks, avoiding the necessity of all instructions being implicitly coded by a human. In other words, these techniques create models that learn from data, instead of following strict programming rules.

A Machine Learning algorithm will build a mathematical model, based on the training data, in order to make predictions or decisions with some autonomy. If this algorithm follows a supervised learning approach, it will build its mathematical model from a set of data that has both the inputs and their respective desired outputs, also know as labeled data. This set of data is used as an example in the learning process and it is represented as a matrix, where each array corresponds to a feature vector. These types of algorithms learn a function that will be used to predict the outputs of new inputs, through the optimization of said function, aiming at the maximization of a score.

If the desired task is a classification problem, then the model will only produce outputs contained in the training data. The desired task can also be a regression problem, where the model will be able to produce an output value within the range of outputs learned from the training data [Bishop, 2006].

The Machine Learning techniques that focus on classification tasks can be separated into five branches: analogizers, bayesians, symbolists, evolutionaries and connectionists [Domingos, 2015]. We will only focus on the latter.

2.2.1 Artificial Neural Networks

Connectionist models are loosely inspired by the animal brains. Because of this, the structure of these models is constructed similarly to the biological neural networks found in said brains, and that's why they're known as Artificial Neural Networks (ANNs) (Figure 2.2). These networks are composed of various layers filled with neural units, called artificial neurons. The connections between the neurons have associated weights that increase/decrease the strength of the information that is transferred between them.

We will only consider feedforward neural networks, such as the example shown in Figure 2.2. Feedforward networks are ANNs where the information is fed from the input layer to the output layer, without going through cycles or loops inside the network. These types of structures generally have full-connection between neurons of two adjacent layers. In the other hand, the units that belong to the same

layers will not be connected.

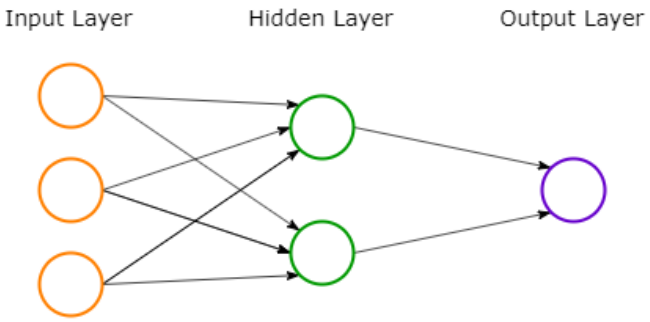


Figure 2.2: Graphic representation of an Artificial Neural Network (ANN).

As presented in Figure 2.2, an ANN is composed of three types of layers:

- **Input Layer** - networks' entry point. Where the data samples are received;
- **Hidden Layer** - where the data is processed and transformed accordingly to the present parameters of the network. Most of the learning process of a network is done on this type of layer. This is the only type of layer that can appear more that once in a single network structure;
- **Output Layer** - networks' exit point. Where the resulting output is calculated, based on all the transformations applied to the data.

In its simplest form, ANNs only have one hidden layer. Even though a network with a single hidden layer can approximate any continuous function, attempting to approximate complex functions with this network may require a large quantity of hidden neurons. As a solution to this problem, it is usually preferred to create a network with more than one hidden layer.

About the number of neurons on the output layer, it depends on the type of problem: one neuron, for regression or binary classification tasks; or the quantity of neurons should be the same as the number of existing labels, for multi-label classification tasks.

Looking from the perspective of the artificial neuron shown in Figure 2.3, we can observe its different components:

- x_1, ..., x_n are the inputs that enter the artificial neuron;
- b_k is a constant value known as Bias, which purpose is to help the model better adjust to the data. Although, the application of this value is optional, it is usually recommended;
- w_{k1}, ..., w_{kn} are the weights associated with each input;
- Net Function is the dot product calculated between the input values and the weights, adding the Bias value. The result is represented as net_k;

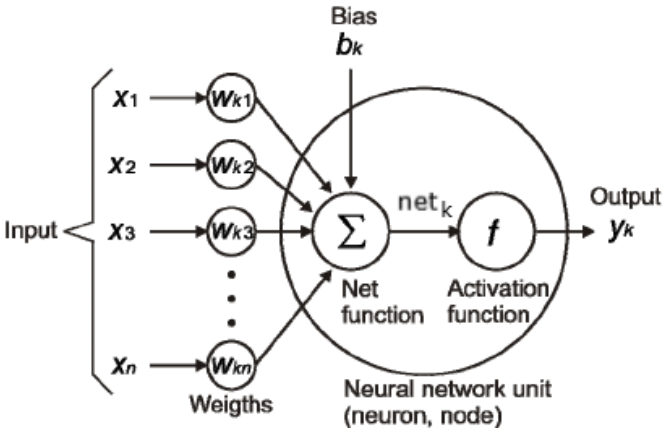


Figure 2.3: Structure of an Artificial Neuron.

- Activation Function is used to choose if the Net Function's result is worth propagating to the next neuron. The result from this function is the output of the neuron (y_k).

Combining the Net Function with the Activation Function, the output of an artificial neuron is computed by (2.6).

$$y_k = f\left(\sum_{i=1}^{N}(w_{ki} \cdot x_i) + b_k\right) \tag{2.6}$$

Generally, the activation function used in the neurons is nonlinear [Kröse and van der Smagt, 1993], because it introduces nonlinearity to a model that mainly performs linear operations, such as the element wise multiplications between the inputs and the weights. The introduction of nonlinearity facilitates how the model generalizes from the learned data samples. We will explore the Sigmoid, ReLU and Softmax activation functions.

The Sigmoid Activation Function (2.7) is used when we intend to output a probabilistic value, since this function only computes values between 0 and 1. The main disadvantage of this function is that the derivative of values closest to 0 or 1 will result in infinitesimal values. This is known as the vanishing gradient problem and consists on the weights receiving values near zero, which prevents these weights from ever having an impact on the overall network.

$$f(net_k) = \frac{1}{1 + e^{-net_k}} \tag{2.7}$$

The ReLU (Rectified Linear Unit) Activation Function (2.8) attenuates the vanishing gradient problem. Although this function is linear for positive values, in its nature is a nonlinear function. This function eliminates all negative activations, since the derivative for this values is 0. ReLU is usually preferred in comparison to the Sigmoid Function, since it leads the models to train faster, without significantly affecting their generalization accuracy. This is due to the simpler mathematical operations used by

ReLU.

$$f(net_k) = max(0, net_k) \tag{2.8}$$

Generally, ReLU is the preferred activation function used on CNN's Convolutional Layers (Section 2.3.1).

The Softmax Activation Function (2.9) is a generalization of the Sigmoid Function, which is mainly used in networks focused on multi-label classification tasks.

When compared with other activation functions, such as Sigmoid and ReLU, this may not be the preferred function. Although, when employing a CNN model to perform multi-label classification with the Cross-Entropy Loss Function, the use of Softmax as an activation function is practically mandatory. This is due to the computations performed by this activation function: normalization of the input values into a vector of probabilities that follow a certain distribution, where the sum of all values equals 1.

$$f(net_k) = \frac{e^{net_k}}{\sum_j e^{net_j}} \tag{2.9}$$

Having built the ANN structure, the next step is to make it learn from a subset of the dataset's samples, designated training set. The most used approach to train a neural network is through use of the backpropagation algorithm [Werbos, 1974]. This algorithm fine-tunes the weights of a neural network, accordingly to the loss metric measured by the model. The loss metric represents how the computed output deviates from the expected output (original label). Accurately tuning the weights will lower the loss value and increase the model's capability of generalization [Rumelhart et al., 1986].

The backpropagation algorithm will iterate over all training samples, repeating this process until the model's loss stabilizes or until we reach a certain limit of training epochs (manually configured). Each backpropagation iteration can be separated into four steps:

- **Forward Pass** - the model will process a training sample throughout the various layers of the network, until it reaches the output layer and computes an output value;
- **Loss Function** - the model will evaluate how much the computed output (o_k) differs from the expected output (t_k). To accomplish this, the model uses a Loss Function, such as MSE (mean squared error) (2.10);
- **Backward Pass** - the objective being the minimization of the loss value, this step works as an optimization problem that aims to achieve that objective. To solve this problem, the model resorts to the gradient descent optimization algorithm [Bishop et al., 1995]. Through the gradient descent algorithm, we compute the partial derivative of the Loss Function, as shown in (2.11), in order to find which weights contribute the most to the model's loss metric. This step is applied throughout

the network, going from the output layer to the first hidden layer;

- **Weight Update** - having the derivatives, we finally update the network weights. Since the objective is to minimize the loss, the weights are updated using the opposite direction of the gradient (2.12). The learning rate (η) is a parameter used to control how fast the model learns. This parameter can have values between 0 and 1. Having a higher learning rate leads to a faster convergence, which means the model will take less time learning an optimal set of weights. Although, if the learning rate is too high it could lead to the gradient descent algorithm never reaching an optimal state.

$$E(w) = \frac{1}{2} \cdot \sum_{k=1}^{N} (t_k - o_k)^2, \text{ where } N = |TrainingSet| \tag{2.10}$$

$$\frac{\partial E}{\partial w_j} = -\sum_{k=1}^{N} (t_k - o_k) \cdot x_{k,j} \tag{2.11}$$

$$w = w_{initial} - \eta \cdot \frac{\partial E}{\partial w} \tag{2.12}$$

The MSE is the simplest loss function used in neural network models. There are other loss functions that we can use in the backpropagation algorithm, such as the, previously mentioned, Cross-Entropy. The Cross-Entropy Loss Function is used to measure the performance of a model that learns multi-label classification tasks (2.13). This function requires the computed outputs to be values between 0 and 1. That's why the Cross-Entropy Loss Function is always used in pair with the Softmax Activation Function.

In the Machine Learning context, the Cross-Entropy function is similar to the Logistic Loss (2.14), because for values close to 1 the loss value slowly decreases. Meanwhile, for values close to 0 the loss value will rapidly increase.

$$E(w) = -\sum_{k=1}^{N} \sum_{j=1}^{M} t_{k,j} \log o_{k,j}, \; M = |Labels| \tag{2.13}$$

$$E(w) = -\sum_{k=1}^{N} (t_k \log o_k + (1 - t_k) \log(1 - o_k)) \tag{2.14}$$

2.3 Deep Learning

Deep Learning belongs to the family of Machine Learning techniques based on ANNs and consists of neural networks composed of multiple hidden layers, designated Deep Neural Networks (DNNs).

Although, the Universality Theorem says that any continuous function f,

$$f : \mathbb{R}^N \to \mathbb{R}^M, \tag{2.15}$$

can be realized by any network with one hidden layer, given enough hidden neural units, the DNNs represent the opposite idea. According to the concept of Deep Learning, it is preferable to have a DNN with many hidden layers, than a "shallow" ANN, with just one large hidden layer. Having a deeper network can decrease the probability of the model being stuck in a local minimum, during the training process [Kawaguchi, 2016].

Deep Neural Networks require a structure with multiple hidden layers due to their ability to extract and learn complex features from data (feature learning). This ability differentiates DNNs from "simpler" ANNs, which are not capable of extracting features on their own.[1]

To take advantage of this ability, a DNN requires large sets of labeled data, in order to learn the features following a certain hierarchy. Considering an example in the context of an image processing problem, in which we intend to identify fruits, the network will use its first hidden layer to extract and encode edges, the following hidden layer to compose and encode these edges into arranged shapes, all the way to the last layer that will determine if and which fruit it is present on the processed image.

Being able to extract the features directly from the data, these networks avoid thus far the need to perform manual feature extraction. This can be seen as an advantage, because feature extraction tends to be conditioned by what we consider as relevant features. Although, this can always present itself as a disadvantage, because increasing the complexity of the features will also increase the number of hidden layers and/or hidden units needed in the network. This comes as a downside, for the deeper the structure is, more time and computational power it will require to learn from the data.

Trying to mitigate the problem of the computational power required, the batching approach appeared as a possible solution. Through batching, the model learns from a batch of training samples at a time, rather than processing each sample individually, which decreases the time and power required for the training computations.

Even though a Deep Learning network can extract the optimal features on its own, we still need to tune certain hyperparameters, such as the number of hidden layers and their respective sizes. By changing the network structure, through those parameters, we can vary the degrees of abstraction [LeCun et al., 2015a].

One of the most common uses for DNNs is solving computer vision (object identification) problems. The Deep Learning structures mostly used for these tasks are known as Convolutional Neural Networks.

2.3.1 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a subset of the Deep Learning networks. These are usually feedforward DNNs and their main purpose is to perform image processing tasks, such as recognizing objects in images. Besides images. CNNs can also be used for other types of data, such as audio.

[1] MathWorks' "What Is Deep Learning?"

Being a type of Deep Learning networks, CNNs are able to learn features directly from data, which means that the relevant features are learned as the network processes the training samples. This characteristic, associated with the convolution operation, is what makes these networks so useful for computer vision tasks and other tasks that require 2-dimensional data.

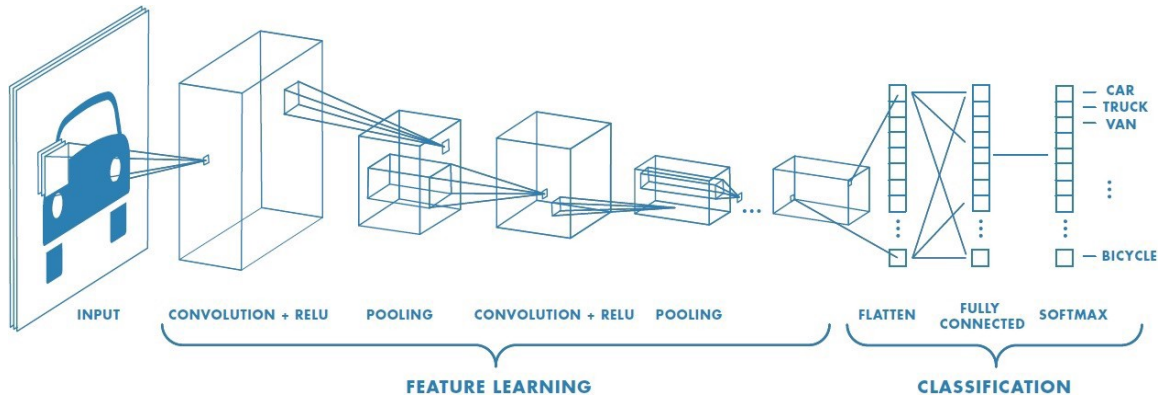


Figure 2.4: Example of a Convolutional Neural Network (CNN).

The structure of a Convolutional Neural Network comprises one input layer, several hidden layers and one output layer. However, what differentiates a CNN from a common Deep Neural Network are its hidden layers. These hidden layers can be of the following types: Convolutional Layer, Pooling Layer or Fully-Connected Layer. In the Figure 2.4, we show an example of a CNN used for object recognition tasks.[2]

The first hidden layer present in a CNN is always a Convolutional Layer. Conventionally, this first layer is responsible for extracting low-level features, such as edges, color and gradient orientation (features explored in Section 2.1).

The Convolutional Layer is responsible for the application of the convolution operation to the data samples. Convolution is a linear operation, common in computer vision tasks, that consists in performing a dot product between a weight matrix, known as filter (e.g. 3x3 filter on (2.16)), and a portion of the input image, known as receptive field (e.g. 3x3 receptive field on (2.17)).

$$\begin{bmatrix} w(-1,-1) & w(-1,0) & w(-1,-1) \\ w(0,-1) & w(0,0) & w(0,-1) \\ w(1,-1) & w(1,0) & w(1,-1) \end{bmatrix} \tag{2.16}$$

$$\begin{bmatrix} f(x-1,y-1) & f(x-1,y) & f(x-1,y+1) \\ f(x,y-1) & f(x,y) & f(x,y+1) \\ f(x+11,y-1) & f(x+11,y) & f(x+1,y+1) \end{bmatrix} \tag{2.17}$$

This operation results in a single value (calculated by (2.18)) and it is applied to the whole image, sliding the filter from left to right and top to bottom.

[2] Image from MathWorks' "Introduction to Deep Learning: What Are Convolutional Neural Networks?"

$$g(x,y) = \sum_{s=-1}^{1} \sum_{t=-1}^{1} w(s,t) \cdot f(x+s, y+t) \tag{2.18}$$

In the digital image processing domain, a filter is a small matrix, designated kernel, convolution matrix or mask. It has multiple purposes, from blurring or sharpening an image to the detection of edges. We accomplish these purposes mainly by computing a convolution between a certain kernel and an image. For example, when extracting the textures (Section 2.1.2) or shapes (Section 2.1.3) from an image, we usually resort to static kernels for the edge detection, such as the Prewitt (2.19) or Sobel (2.20) kernels (from [Prewitt, 1970] and [Sobel and Feldman, 1968], respectively).

$$M_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \qquad M_y = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} \tag{2.19}$$

$$M_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad M_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{2.20}$$

The convolution operation results in a feature map with the results from all the dot products calculated (Figure 2.5), which have a higher value when the feature present in the filter matches with the pixels on the receptive field. To ensure that this operation succeeds, the filters must have the same depth as the input images.

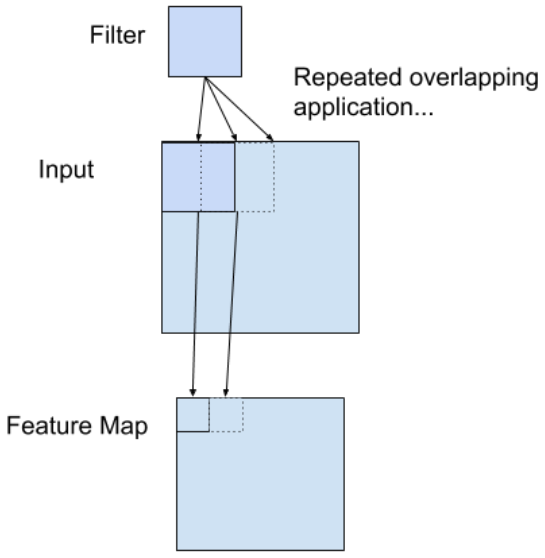


Figure 2.5: Example of the Convolution operation.

Usually, Convolutional Layers use ReLU as the activation function that is applied to the resulting feature maps.

Convolutional Layers have four hyperparameters:

- **Filter size** - tuple that determines the width and height of the filters used for the convolution operation;
- **Number of filters** - defines how many filters will be used in this layer. The higher the number, the more features will be extracted, since the filters serve as feature identifiers. This parameter is directly associated with the depth of the output, which relates to the quantity of feature maps created;
- **Stride** - measure that controls how much the filter is shifted on the input images. This can be useful to avoid receptive fields from overlapping;
- **Padding** - controls the dimensionality of the feature maps by adding more pixels (normally with zero value) around the receptive field. This can be useful to regulate the size of the resulting feature maps, thus preserving more information.

The width and height of the feature maps are calculated according to:

$$Output_{size} = \frac{Input_{size} - Filter_{size} + 2 \times Padding}{Stride} + 1 \tag{2.21}$$

The Pooling Layer is used as a downsampling technique that is applied to all feature maps. Reducing the dimensions of the feature maps, it will require less computational power to process the data and it will lessen the probability of overfitting. This downsampling will maintain the dominant features, because they are positional and rotational invariant.

In this layer, a filter is applied to the input image, in order to obtain a value for each subregion of that image. The filter shifts with a stride equal to its length. The two main downsampling approaches used on these layers are:

- **Max Pooling** - extracts the maximum value from the subregion covered by the filter (Figure 2.6);[3]
- **Average Pooling** - computes the average of the values from the subregion covered by the filter.

Even though both approaches are used for dimensionality reduction, Max Pooling is usually better, because it also performs noise suppression (discards noisy activations).

The last layer in a CNN is a Fully-Connected Layer, whose purpose is to transform the feature map received into a N-dimensional vector, where N is the number of labels associated with a given classification task. This transformation learns nonlinear combinations from the feature maps and correlates extracted features to the labels, in order to predict the label for the input sample. The values on the

[3] Image from http://cs231n.github.io/convolutional-networks/

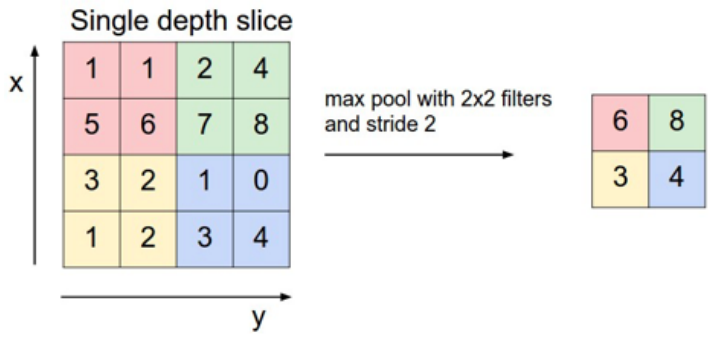


Figure 2.6: Example of the Max Pooling operation.

N-dimensional vector represent the probabilities of the processed image belonging to each label (values between 0 and 1).

Generally, Softmax is the activation function used in a Fully-Connected Layer. The Softmax function normalizes the probabilities from the output vector, making their sum equal to 1. When using Cross-Entropy as the model's loss function, the use of Softmax on the last layer is strictly required.

Similarly to ANNs, Convolutional Neural Networks train through the backpropagation algorithm. These networks learn the features that minimize the loss metric for the given task, using the stochastic gradient descent. Most of CNNs use Cross-Entropy as their loss function.

On the Fully-Connected Layer, the backpropagation algorithm will have the same behavior as in the common hidden layers from an ANN. Where the algorithm differs is in the Convolutional and Pooling layers. On the former layer, the backward pass for the convolution operation is realized by computing the convolution between the input and the filters spatially-flipped. On the latter layer, during the forward pass the algorithm needs to keep track of the indexes associated with the downsampling activation, in order to efficiently route the gradients on the backward pass.

There are different approaches to how a CNN can be used to perform image classification tasks. The most common approaches are:

- **Train a CNN from scratch** - this requires a large amount of labeled data and can take a while to process it all;
- **Transfer learning** - fine-tuning a pre-trained model, which consists of feeding our data to a previously trained network. Takes less time to learn and it is most useful in tasks where we do not have large amounts of labeled data;
- **Use CNN as a feature extractor** - use the network to learn the features (filters) from the dataset and then use those features as input for another Machine Learning model.

Choosing which approach to follow will depend on both the quantity of available data and on what it is intended by the task at hand.

When comparing ANNs to CNNs, the main difference between them, for a given image classification task, is that the former requires the features to be manually extracted and it is constrained by what we consider as relevant features, while the latter, being a Deep Learning network, extracts the features autonomously. Motivated by digital image processing algorithms, CNNs are able to extract as many features as there are filters in their convolutional layers. The complexity of these extracted features is related to the depth of the network. The first convolutional layer will extract low-level features, which will increase in complexity by the multiple convolutions applied throughout the network.

Adding to this, artificial neurons from CNNs can share the same filter (weight matrix), across all receptive fields convolved with that filter. This reduces the memory required, since the model will not have to store a weight vector for each receptive field [LeCun et al., 2015b].

Another difference between these models is that an ANN (as well as other Machine Learning models) tend to converge as the size of the dataset increases, whereas a CNN (being a Deep Learning model) will scale with the data.

Although, it may seem that CNNs are better than ANNs, this is not the case. When choosing which one we should use for a certain image classification task, the decision has to consider the size of the labeled dataset that we have available, as well as if there are enough computational power to rapidly process that data.

2.4 Regularization

One of the major problems associated with Machine Learning is how easily a model becomes overfitted. Overfitting consists in a model learning most of the small details (noise) from the data samples used in its training [Kröse and van der Smagt, 1993], resulting in a reduced training loss and, simultaneously, in an increased testing loss. This is justified by the model being too tuned to the data samples it already learned: the training samples. Because of their complex nature, Deep Neural Networks models are more prone to overfitting.

As a solution to the overfitting problem there is a technique known as regularization. This technique improves the testing loss by slightly modifying the learning algorithm, leading the model to compute better generalizations. Usually, regularization modifies a machine learning algorithm by penalizing its coefficients. But if it is a Deep Learning algorithm, the penalization will be applied to the weight matrices of the artificial neurons.

When using regularization, we need to be careful not to create the opposite problem: underfitting. If the regularization leads to a large number of weights being nearly equal to zero, it will result in a model whose generalizations are too broad. Hence, the value of the regularization coefficient should be optimize in order to create a well-fitted model.

In the following sections (2.4.1 and 2.4.2), we will detail some of the most common regularization techniques.

2.4.1 Lasso (L1) and Ridge (L2)

Even though Lasso (L1) and Ridge (L2) are two different regularization techniques, their approach is similar. In both techniques, the regularization is done by adding a regularization term to the loss function used by the model [Sun and Su, 2015]. Applying this change on a neural network will result in a decrease of the weight matrices' values.

The main difference between these two techniques is the regularization term that is added to the loss function.

In L1, the loss function with regularization is:

$$LossFunction_{L1} = LossFunction_{Original} + \frac{\lambda}{2m} \times \sum \|w\| \tag{2.22}$$

Adding this regularization term will lead to a penalization of the absolute value of the weights. The weights may be reduced to zero by this term, which is useful if we intend to compress our model.

In L2, the loss function with regularization is:

$$LossFunction_{L2} = LossFunction_{Original} + \frac{\lambda}{2m} \times \sum \|w\|^2 \tag{2.23}$$

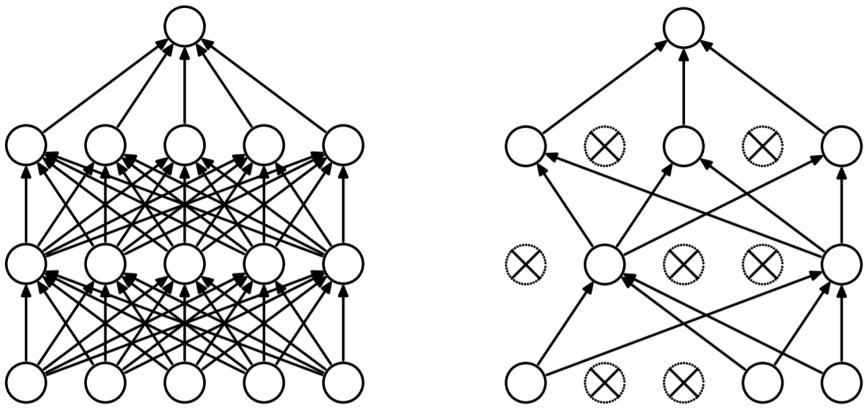
Adding this regularization term will lead the weights to values very close to zero, but not exactly zero. For this reason, the L2 regularization is also known as weight decay [Hinton, 2012]. Usually, when comparing both regularization techniques, L2 is the preferred choice.

The λ in both (2.22) and (2.23) is the regularization parameter of the regularization term. This is the hyperparameter that we need to fine-tune in order to optimize the model.

2.4.2 Dropout

Dropout is a regularization technique in which some artificial neurons are randomly selected, at every training iteration, to be temporarily removed from the network, along with their respective input and output connections [Srivastava et al., 2014]. This is the most frequently used regularization technique for Deep Learning models, because it introduces randomness in large network structures, which consecutively allows them to learn more complex features.

The Dropout regularization has the hyperparameter *p*, which represents the fraction of neurons that will be dropped in each iteration. Neurons can be dropped from both the input and hidden layers, as shown in Figure 2.7 [Srivastava et al., 2014].



(a) Neural Network before Dropout. **(b)** Neural Network after Dropout.

Figure 2.7: Application of Dropout regularization on a Neural Network.

This technique has a different behavior for the training and testing processes. In the former, for each training iteration of each data sample, a random fraction of p neurons, and their respective connections, will be removed from the structure (Figure 2.7(b)). In the latter, all neurons remain in the structure, however the activation functions of each will be reduced by a factor of p, in order to compensate for the dropped connections during the training process.

3

Related Work

In this chapter, we will explore the most remarkable steps in the history of the Convolutional Neural Networks. These steps are divided between two sections:

- **Road to Convolutional Neural Networks** (Section 3.1) - exploration of the major steps in the evolution of the neural networks, from the Perceptron to the first CNN;
- **Convolutional Milestones** (Section 3.2) - exploration of the most groundbreaking CNNs and analysis of the innovations each of them introduced to the Deep Learning field.

In the last section of this chapter, (Section 3.3), we succinctly denote some of the major shortcomings associated with modern Convolutional Neural Networks. We also expose the problem that justifies the objective of this thesis.

3.1 Road to Convolutional Neural Networks

In this section we will explore the most relevant steps on the evolution of the Machine Learning models, from the first “artificial neuron”, known as Perceptron, to the Deep Neural Networks mostly used for

image classification tasks, known as Convolutional Neural Networks. We will analyze what these models consist of and how each one of them influenced the state-of-the-art of CNNs.

3.1.1 Perceptron

In 1958, Frank Rosenblatt proposed the Perceptron algorithm to solve binary classification tasks [Rosenblatt, 1958]. Rosenblatt's algorithm was inspired by the artificial neuron model presented by McCulloch and Pitts, in 1943 [McCulloch and Pitts, 1943]. The Perceptron is also known as Single-layer Perceptron and it is considered as the beginning of the connectionist models (Section 2.2.1).

The Perceptron algorithm learns a binary classifier using a threshold function. This function is able to map the input vector, $\mathbf{x}$, to a corresponding output value, y. The output value is binary, that is, it can only assume the values 0 and 1 (3.1).

$$y = f(net) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

The threshold function (3.1) classifies the input $\mathbf{x}$ with either a positive or a negative label, according to its position in relation with the hyperplane, known as decision boundary. The parameters $\mathbf{w}$ and b are the model's weight vector and bias value, respectively, and represent the spatial disposition of the decision boundary.

$$w_i = w_i + \eta \cdot (t - y) \cdot x_i \tag{3.2}$$

After training over a data sample, the weight vector is updated. The weight update step of the algorithm uses the error between the expected label for the input, t, and the computed output, y, multiplied by both the input and a learning rate, η, as represented in (3.2).

The major limitation of the Perceptron algorithm was that, as a linear classifier, it could only converge if the training set was linearly separable. This was shown by Minsky and Papert, when they tried to solve the XOR problem with the Perceptron [Minsky and Papert, 1969].

Input		Output
0	0	0
0	1	1
1	0	1
1	1	0

Table 3.1: XOR Function.

The XOR function consists in the *exclusive-or* operator, in which the output is 0 if both outputs have the same value and 1 if they are different (Table 3.1).

This limitation motivated the evolution of the connectionist models to the next step: the Multi-layer Perceptron.

3.1.2 Multi-layer Perceptron

The Multi-layer Perceptron is a connectionist model, from the class of feedforward ANNs (Section 2.2.1), and consists of a network with multiple perceptrons (artificial neurons). These perceptrons are organized in three or more layers: input layer, output layer and one or more hidden layers. In the 1980s, this was a recurring model used to solve speech recognition, image recognition and machine translation tasks.

The most relevant innovation this model brought over the Single-layer Perceptron was the ability to solve both binary and multi-label classification tasks. Adding to this, the Multi-layer Perceptron was able to use nonlinear activation functions in its neurons, which allowed it to solve tasks with nonlinearly separable data, such as the XOR problem, and even realize regression tasks [Debao, 1993]. According to this, Multi-layer Perceptrons are considered universal approximators, contrary to Perceptrons that can only approximate linear functions.

Using linear activation functions the resulting decision boundary will be linear, because it is computed as a weighted sum of all the activation functions present in the network. However, if the activation functions are nonlinear, such as the sigmoid or the hyperbolic function (explored in Section 2.2.1), then the model's decision boundary will be represented as a nonlinear combination of each neuron's linear boundary.

Being an Artificial Neural Network, the Multi-layer Perceptron learns from data through both the backpropagation and gradient descent algorithms, as explored in Section 2.2.1.

3.1.3 Receptive Fields in the Visual Cortex

Between the 1950s and 1960s, Hubel and Wiesel presented the notion of receptive fields [Hubel and Wiesel, 1962]. Through analysis of the brains of several mammals, they noticed that the area known as visual cortex contained neurons that would individually respond to subregions of the visual field. The receptive field is the region of the visual field where a certain visual stimulus affects the activation of a single neuron. Hubel further noted that each brain hemisphere had its own visual cortex, which represented the respective visual field associated with each mammalian eye [Hubel, 1995].

In 1968, Hubel and Wiesel identified the presence of two different types of visual cells in the mammal brain [Hubel and Wiesel, 1968]:

- **Simple Cells** - These cells react to visual stimuli when their receptive fields receive an input with a pattern that follows a specific position and orientation. The reaction from these cells is maximum for patterns previously learned;

- **Complex Cells** - These cells also react to visual stimuli, but their receptive fields are significantly larger than those present in Simple Cells. Contrary to Simple Cells, Complex Cells are not sensitive to the position or orientation of the pattern received in their receptive fields. Their main purpose is to integrate the patterns processed by the Simple Cells, allowing them to change position and orientation.

Hubel and Wiesel proposed a model in which Simple Cells and Complex Cells would be arranged in cascading layers, in which the received patterns increase their complexity from one layer to the following. This model was inspired by the hierarchical organization of these cells in the visual cortex [Hubel, 1995].

Unlike the ANNs, where there is full-connection between neurons from adjacent layers (Section 2.2.1), providing every unit with a global view of the input, in Hubel and Wiesel's proposed model, each neuron (cell) has a local view of the input. Thus, only the output layer will have access to the global view of the input, due to the integration of several local views realized by the layers of Complex Cells.

3.1.4 Neocognitron

The Neocognitron is a hierarchical multilayered ANN, proposed by Fukushima in 1980 [Fukushima, 1980] and further analyzed throughout that decade, in his following works [Fukushima, 1988, Fukushima, 1989].

The Neocognitron model serves as an extension of the Cognitron [Fukushima, 1975], taking inspiration from the cascading model proposed by Hubel and Wiesel [Hubel, 1995], in order to create an image recognizer invariant to size, shape and spatial shifts of the patterns. This network was widely used to recognize handwritten characters, among other pattern recognition tasks and was the first neural network to require its artificial neurons to be distributed along the structure, allowing them to share their weights.

Motivated by the Simple and Complex Cells presented by Hubel and Wiesel [Hubel and Wiesel, 1968], the Neocognitron also has two main types of cells:

- **S-cells** - These cells resemble the Simple Cells found in the visual cortex. Their purpose is to perform a localized pattern matching on the pixels of the received input, that are inside the receptive field. This pattern matching is realized by the dot product operation between the preferred stimulus and the pixels in the receptive field. The resulting output value will be greater if the pattern present in the receptive field is similar to the preferred feature. The connections that enter these cells can be modified while the network is learning;

- **C-cells** - These cells resemble the Complex Cells found in the visual cortex. Their purpose is to recognize the feature in the previous S-cells even if they have their size, shape or position changed. This is accomplished by how strong these cells react when a feature is found in the

directly connected s-cells, regardless of the position of the recognized feature. The connections that enter these cells are fixed and are not affected by the learning process.

The Neocognitron is arranged in a cascade of consecutive pairs of layers. Each pair is composed of one layer of S-cells followed by one layer of C-cells, as shown in Figure 3.1. The first layer, U_0, is composed of a single plane of cells, that represents the input image. The $U_{S\#}$ and $U_{C\#}$ layers are composed of several cell planes and represent a layer of S-cells and C-cells, respectively.

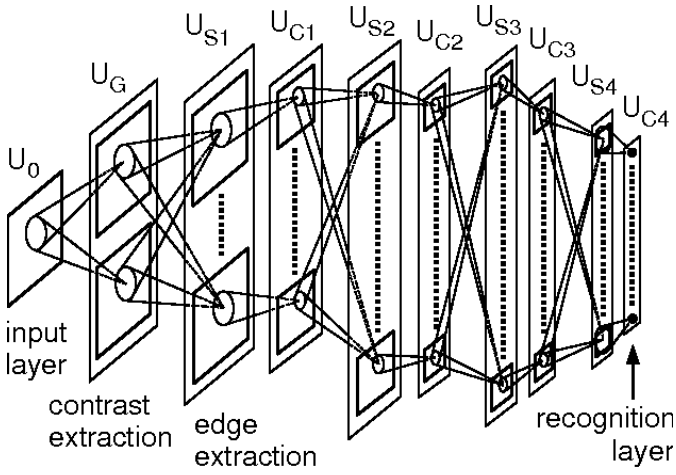


Figure 3.1: Structure of the Neocognitron, proposed in [Fukushima, 2003].

All the S-cells in the same plane will recognize the same preferred feature in different positions of the input, performing similarly to a feature extractor applied to the full image.

The planes of C-cells behave as blurring filters, in which the spatial information of the features is discarded (downsampling). As any filter, these planes have to be tuned in order to avoid falling over one of the edges: if we apply too much invariance on the C-cells, then features might get filtered out, and if we do not apply enough invariance, then similar patterns might get recognized as being different, due to minor changes between them [Barnard and Casasent, 1990].

The images go from the input layer to the recognition layer, being processed hierarchically by a cascade of interlaced layers of S-cells and C-cells (Figure 3.1). On the recognition layer, the model has several planes with only one cell, which corresponds to the several labels determined by the classification task. Each image will be associated with the label which corresponds to the cell with the highest output value, as observed on the example from Figure 3.2.

Analyzing the example shown in Figure 3.2, we can perceive that the local features extracted are integrated as the information goes deeper into the network and at the recognition layer we obtain the global features which will be used to classify the input image.

There are two approaches we can choose from, when creating the Neocognitron structure. The first approach is to use unsupervised learning to learn the preferred features from the dataset and then

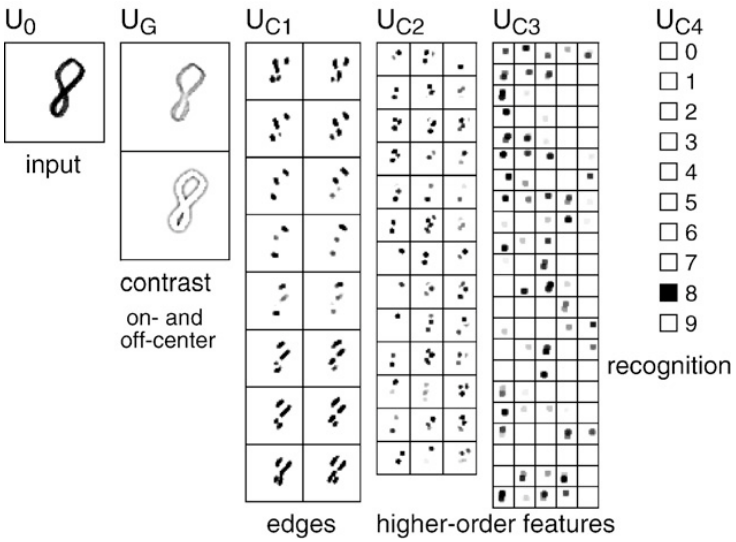


Figure 3.2: Example of the Neocognitron's pattern recognition process [Fukushima, 2003].

define what feature each S-cell plane should recognize. The second approach is to manually define a set of preferred features on the first layer of S-cells. These features usually correspond to lines of varying orientation.

The Neocognitron learns and updates its S-cells' preferred features using a self-organizing competitive learning algorithm [Fukushima, 1989]. Through this algorithm, when searching for a feature in the input, all the S-cells with overlapping receptive fields will compete over which one will recognize that feature. For each plane in a layer of S-cells, the cell that computes the highest output for the given feature will be chosen to represent its respective plane. Following this step, the connections to this plane will become stronger, due to the weight sharing. In the end, each S-cell will have the preferred feature associated to the stimulus that caused it to win the "competition".

Although the Neocognitron performs downsampling in its C-cells, based on the position shifts and distortion of the features, one of this model's limitations was the lack of rotation invariance applied on those respective cells. However, on later iterations of this network, there were some changes implemented to correct the aforementioned limitation [Li and Wu, 1993, Satoh et al., 1999].

This model was of great importance, because it served as a major inspiration to the creation of the first Convolutional Neural Networks.

3.1.5 Map Transformation Cascade

The Map Transformation Cascade (MTC) was proposed by Ângelo Cardoso and Andreas Wichert [Cardoso and Wichert, 2010] and follows the principles presented in previous biological motivated models, such as the Hubel and Wiesel's cascading model (Section 3.1.3) and the Neocognitron (Section 3.1.4).

The motivation behind the MTC was to create a simpler and easier to interpret model than the Neocognitron. Given this, the MTC employed general learning methods, such as previous established algorithms (kNN, k-Means and Backpropagation), instead of handcrafted approaches like the Neocognitron's self-organizing competitive learning algorithm.

Identical to the Neocognitron, the MTC structure is composed by two types of cells (S-cells and C-cells), organized into layers. These layers only have cells of the same type (S-layers and C-Layers) and are grouped into pairs, known as stages. Each stage has an S-layer followed by a C-Layer. In Figure 3.3, we can observe the MTC stages (one per line) and their respective layers.

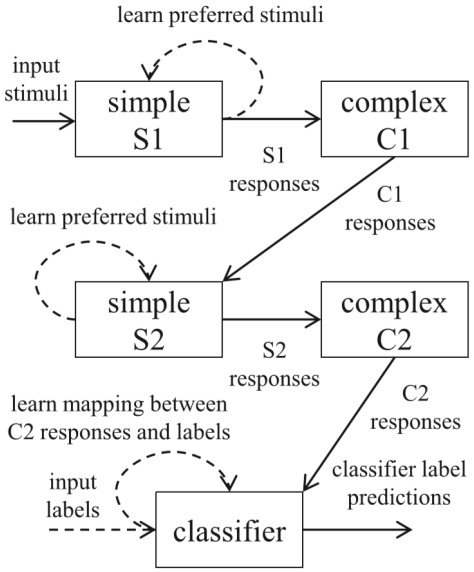


Figure 3.3: The Map Transformation Cascade structure used to learn and classify handwritten digits from the ETL1 dataset [Cardoso and Wichert, 2013].

The S-cells objective is to map the received input image into a feature space. Through this operation, we obtain a feature description of the processed image.

In order to perform the mapping operation, the S-layers have to be trained so their cells react to the stimulus associated with the correct preferred features. The training of these layers is performed through unsupervised learning, such as the k-Means clustering algorithm.

All the input images are tiled with a mask. Each position of this mask represents a receptive field containing a sub-region of the input. The k-Means algorithm creates the clusters computing the Euclidean distance between all these sub-regions. In the end, the resulting centroids of the clusters correspond to a unique feature and are organized in a dictionary of features.

The mapping operation resorts to this dictionary to associate each receptive field (of each image) to one preferred feature. This approach is known as "winner-take-all".

The C-cells objective is the application of invariance to spatial shifts and distortions onto the resulting

output of the previous layer of S-cells. These cells perform downsampling by blurring the feature space.

Similarly to the Neocognitron, MTC's C-layers remove the positional information of the features, transforming them into sparse binary vectors. These vectors will have value 1 for the position assigned to the feature present in the receptive field and value 0 for all the other positions. The operation performed in the C-layers is called transformation operation.

Oppositely to S-layers, the C-layers are fixed, which means they are not affected during the learning process.

The MTC network is not fully-connected, which means that through its structure the artificial neurons only have access to a local view of the original input. All local views are hierarchically integrated through the network layers. It is only in the last layer, known as Recognition layer, that the network has access to the global view of the original input.

After the information is processed at all stages, through cascades of mapping and transformation operations, the resulting patterns are used on the Recognition layer to train its classifier. This layer uses supervised learning, such as the kNN classifier, to learn the extracted features (Figure 3.3). The kNN classifier resorts to the Euclidean distance to assign the most appropriate label to each image.

When classifying new images, these will be processed throughout the model. Firstly, the MTC extracts the features present in the images using mapping and transformation operations. Lastly, it computes the associated labels accordingly to the output given by the Recognition layer's classifier.

The operations only related to the training of the MTC model are represented in the Figure 3.3 by dashed lines. All these operations are discarded after the model has learned the preferred features and trained the classifier.

In comparison to the Neocognitron, the MTC loses more information in its S-cells, due to the "winner-take-all" approach, in which an image is only assigned with the most similar feature. Regardless of all the discarded information, this model achieves similar results to the Neocognitron. In a follow up paper, Ângelo Cardoso and Andreas Wichert showed that the MTC improves the loss rates for datasets under extreme amounts of noise [Cardoso and Wichert, 2014].

One of the most important innovations of the MTC is how it achieved similar results to other models, such as the Neocognitron, with shallower network structures.

3.1.6 The First CNNs

In 1989, a division of AT&T Bell Laboratories created a neural network to recognize handwritten ZIP Code digits, using convolutions between the images' receptive fields and a set of manually designed kernels [Denker et al., 1989].

Later that year, Yann LeCun, and other computer scientists under AT&T Bell Laboratories, [LeCun et al., 1989] presented an improved version of the previous neural network. This network used the

backpropagation and gradient descent algorithms to autonomously learn the filters, present in its Convolutional layers, directly from the dataset of handwritten digits (Section 2.3.1). The structure of the proposed network is depicted in Figure 3.4.

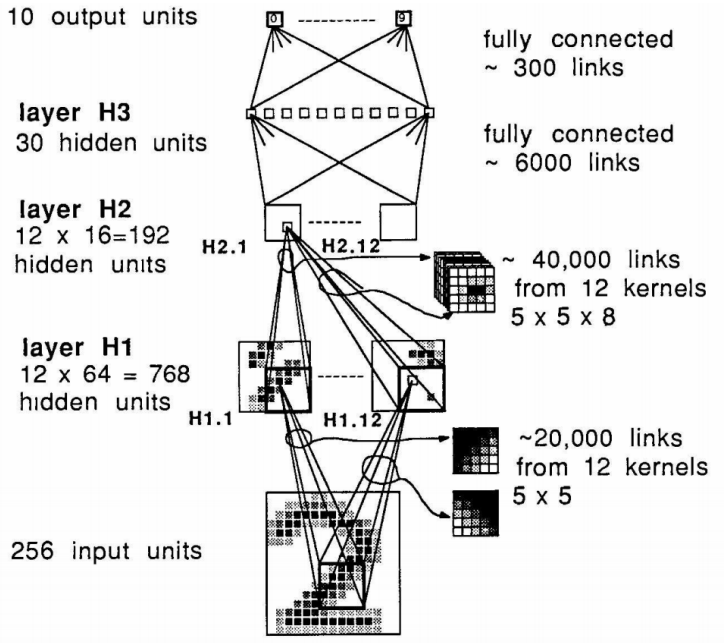


Figure 3.4: Structure of the first CNN to use the Backpropagation algorithm to learn its convolutional filters [LeCun et al., 1989].

Despite the difference in the learning approaches, both these networks were inspired by the two main types of cells presented by Hubel and Wiesel [Hubel and Wiesel, 1968].

The Convolutional layers resemble the layers of Simple Cells, given that they are composed of several synchronized artificial neurons able to recognize certain features present in the images. The neurons from a Convolutional layer are tuned during the learning process and are able to share their weights between them (Section 2.3.1).

Following each Convolutional layer there is a Pooling layer, which resembles a layer of Complex Cells. This layer is used to reduce the dimensionality of the feature map outputted by the previous Convolutional layer. The proposed Pooling layers used Average Pooling as their downsampling approach (Section 2.3.1). Similarly to the Neocognitron (Section 3.1.4), the proposed CNN uses its Pooling layers to add spatial shift and distortion invariance to the model.

While the Convolutional and Pooling layers were responsible for extracting the features present in the images, the proposed model used two Fully-Connected layers: the first to integrate all the local feature maps into a global feature view and the second to classify the images according to the previously extracted features (Figure 3.4).

This network proved to produce better results in comparison to the structure with manually designed

kernels. Furthermore, it was able to process and learn a broader range of classification tasks and image datasets. Considering these reasons, this model became the basis for Convolutional Neural Networks, and consequently for state-of-the-art computer vision tasks. This model is considered to be the first step in the path of the groundbreaking CNNs that emerged over the years until the present day.

3.2 Convolutional Milestones

In this section, we will present some CNNs that are considered groundbreaking advancements for the Deep Learning field. These structures brought many innovations, some of which still serve as a basis to the state-of-the-art in Convolutional Neural Networks.

3.2.1 LeNet-5

The LeNet-5 network, created by Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, in 1998, was one of the first CNN structures [LeCun et al., 1998]. This network had the purpose of recognizing handwritten digits: MNIST dataset[1] [LeCun, 1998]. It was used by several banks to recognize numbers written on checks.

This Convolutional Neural Network is considered, by many, as a defining moment in the computer vision field. Combining digital image processing techniques, such as the convolution operation, with a neural network capable of learning through use of the gradient descent and backpropagation algorithms, LeNet-5 managed to achieve great results when classifying written or printed digits, even if those digits had some variability.

LeNet-5 was composed of eight layers, including both the input and output layers, and processed images of size 32x32 pixels (Figure 3.5). Although the MNIST dataset's images have a size of 28x28 pixels, they were padded with zeros in order to become 32x32.

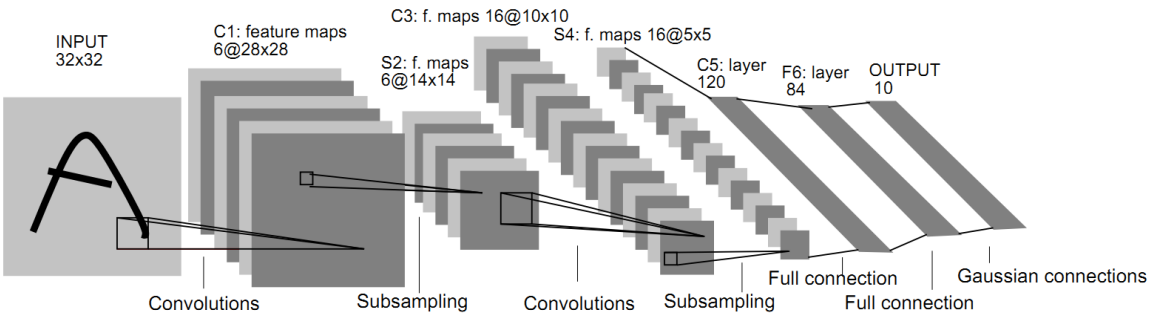


Figure 3.5: Structure of the Convolutional Neural Network LeNet-5 [LeCun et al., 1998].

[1] MNIST dataset

Besides the input and output layers, the network has three Convolutional layers and two Pooling layers, interspersed. All Convolutional layers use 5x5 filters, with stride 1. For every convolution operation performed, the number of feature maps increases, with the first Convolutional layer resulting in 6, while the last results in 120. Both Pooling layers use 2x2 filters, with stride 2 and Average Pooling as their downsampling approach. For every downsampling step, the feature maps halved their size.

About the activation functions, all layers use the hyperbolic tangent function (tanh), except the output layer which uses the Softmax function.

Since the intended task was to recognize digits, we have 10 artificial neurons in the output layer, one for every digit (from 0 to 9).

Despite LeNet-5 being one of the first CNNs, it was enough to understand that, in order to extract features from higher resolution images, we would require networks with more depth (more convolutional layers) and that this problem would be constrained by the computational power available.

3.2.2 AlexNet

The AlexNet network, created by Alex Krizhevsky in 2012, was a variant of LeCun's structure (Section 3.2.1), only it had more layers and some other innovations [Krizhevsky et al., 2012]. This structure was build to be executed on a GPU, which guaranteed faster times. AlexNet won the 2012 ImageNet Large Scale Visual Recognition Challenge (ILSVRC)[2], with a 17% top-5 loss rate.

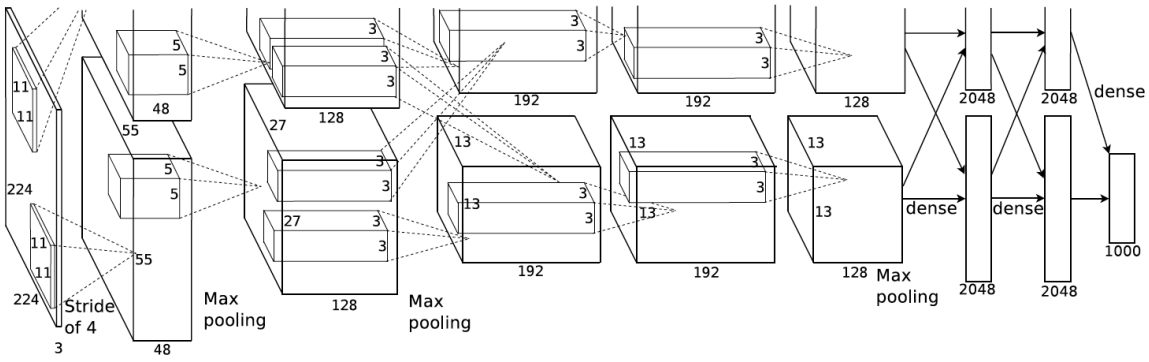


Figure 3.6: Structure of the Convolutional Neural Network AlexNet [Krizhevsky et al., 2012].

AlexNet was composed of five Convolutional layers, three Pooling layers and two Fully-Connected layers, besides the input and output layers (Figure 3.6). Krizhevsky created this structure with ReLU as the activation function for all Convolutional and Fully-Connected layers, which was one of the major innovations of this network. Using ReLU greatly improved the network training performance, comparatively to using the tanh or sigmoid activation functions, and prevented the vanishing gradient problem.

[2] ImageNet dataset

Instead of following the same downsampling approach used in LeNet-5, this model's Pooling layers used the Max Pooling downsampling approach. Similarly to LeCun's network, AlexNet's output layer also used Softmax as its activation function and had 1000 neurons, one for each label present in the ILSVRC dataset [Deng et al., 2009].

About the innovations of this CNN, in addition to the ReLU activation function, AlexNet also improved the use of GPUs to process the training of these types of neural networks, by parallelizing the training between two GPUs [Krizhevsky, 2014], as shown in Figure 3.6. Each GPU was assigned with half of the first layer neurons and throughout the network it only processed the convolutions of neurons sequentially connected to those neurons. Despite this distribution of workload, there was still communication between the GPU's in certain layers.

Due to its increased depth, in comparison to LeNet-5, AlexNet started to suffer from overfitting. Trying to reduce this problem, Krizhevsky resorted to two techniques: Data Augmentation [Simard et al., 2003] and Regularization (Section 2.4.2). The first technique consisted in artificially enlarging the dataset, recurring to label-preserving transformations. The methods used in AlexNet generated new images through geometric transformations (translations and horizontal reflections of the original images) and color transformations (applying small changes to the intensities of the RGB channels of the original images). The second technique consisted on the application of two different regularization approaches: using L2 regularization (Section 2.4.1), with a weight decay value of 0.0005, when training the model through the stochastic gradient descent; and using Dropout regularization (Sec. 2.4.2) on both Fully-Connected layers, with a probability of 0.5. The use of Dropout doubled the amount of time required for the model to converge its loss rate.

Despite the initial overfitting problem, the increased depth of this model made it capable of learning more complex features, which was a big improvement over the LeNet-5 network.

3.2.3 GoogLeNet

The GoogLeNet network, created by Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke and Andrew Rabinovich, in 2014, (under Google Inc.) [Szegedy et al., 2015], is regarded as one of the first truly Deep Neural Networks, because its depth was greater than any previous CNN. This network won the ILSVRC challenge in 2014 with a top-5 loss rate below 7%. The origin of the name GoogLeNet came from the combination of the names Google and LeNet-5, simultaneously referencing the company that created the network and paying tribute to LeCun's CNN.

GoogLeNet was composed of 22 layers, as shown in Figure 3.7, which made it one of the deepest models at that time. This network introduced the concept of inception modules (Figure 3.8) and that's why it is also known as Inception v1.

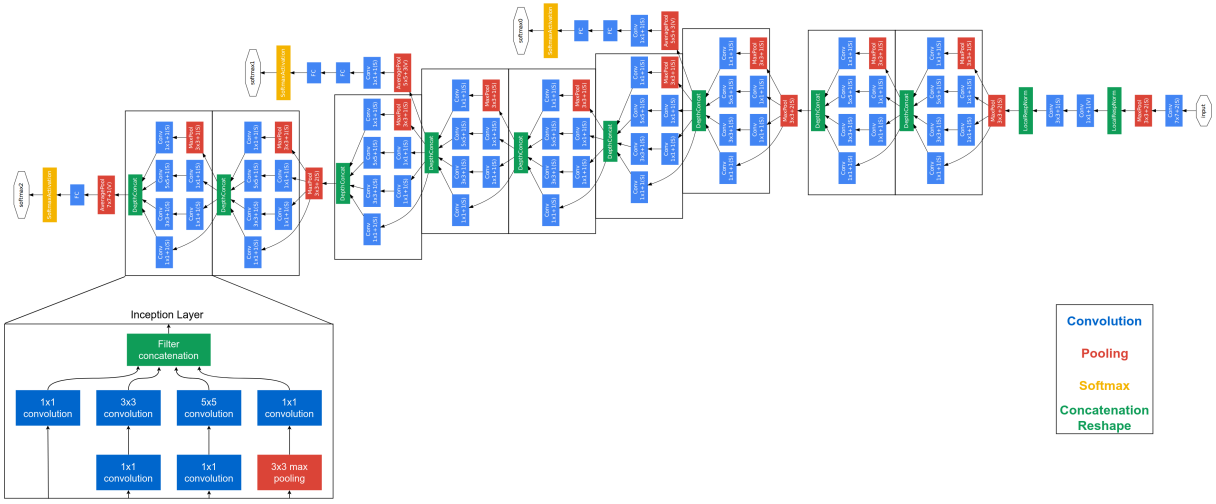


Figure 3.7: Structure of the Convolutional Neural Network GoogLeNet [Szegedy et al., 2015].

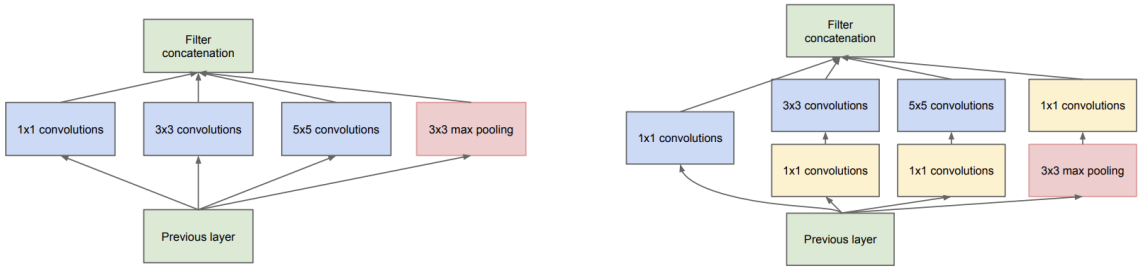
On its overall structure, GoogLeNet used ReLU as the activation function for all Convolutional layers. The network has 9 inception modules throughout its structure and it's last layers are, as follows:

- **Pooling Layer** - Average Pooling, with a 7x7 filter and stride 1;
- **Dropout** with a probability of 0.4;
- **Fully-Connected Layer** - ReLU as its activation function;
- **Output Layer** - with 1000 neurons (equal to the number of labels present in the ILSVRC dataset [Deng et al., 2009]) and Softmax as its activation function.

The inception modules, shown in Figure 3.8(a), were inspired by the series of fixed Gabor filters used by a neuroscience model of the primate visual cortex [Serre et al., 2007]. The purpose of these modules was to make the model capable of processing multiple scales of the same features, through use of a series of 1x1, 3x3 and 5x5 convolution filters and 3x3 max pooling filter, whose values are updated as the model learns from the training samples. As we can observe on Figure 3.8(a), the resulting feature maps are all concatenated together and represent the output from the inception module.

Considering so many different scales of the same features introduced a dimensionality problem, which greatly increased the computational power required to train the model. In order to solve this problem, it was introduced 1x1 convolutions to the inception modules, as shown in Figure 3.8(b). These 1x1 convolutions performed the necessary dimensionality reduction and consequently decreased the number of computations performed by this model.

Another innovation from this network was the use of auxiliary classifiers during the model's training. As shown in Figure 3.7, there are two side-branches in the structure. These small branches are the auxiliary classifiers and are structured as follows:



(a) Naive version. **(b)** Dimensionality Reduction version.

Figure 3.8: Inception Module [Szegedy et al., 2015].

- **Pooling Layer** - Average Pooling, with 5x5 filter and stride 3;
- **Convolutional Layer** - 128 convolution filters of size 1x1 and ReLU as its activation function;
- **Fully-Connected Layer** - with 1024 neurons and ReLU as its activation function;
- **Dropout** with a probability of 0.7;
- **Fully-Connected Layer** - with 1000 neurons (equal to the number of labels present in the ILSVRC dataset [Deng et al., 2009]) and Softmax as its activation function.

These auxiliary classifiers contributed to the training process, by computing the loss rate until the point where they were inserted in the network. The computed loss is added to the total loss of the model, with a weight of 0.3. When classifying/predicting new data samples, these branches are discarded from the model.

The reason behind the auxiliary classifiers was to both avoid the vanishing gradient problem and provide some sort of regularization to the model. They were connected to layers in the middle of the network to help backpropagate the gradient until it reached the first hidden layer.

Although it was much deeper than AlexNet, GoogLeNet was able to train faster and to produce better results, over the same datasets, due to the inception modules.

One of the most relevant drawbacks of GoogLeNet was the need to have large labeled datasets to train the model, due to the depth of its structure.

3.2.4 ResNet

The ResNet (Residual Net) network, created by Microsoft in 2015 [He et al., 2016a], was responsible for the beginning of the depth revolution on CNNs. Microsoft's ResNet won the ILSVRC challenge in 2015 and followed VGG's ideology [Simonyan and Zisserman, 2014] of exploring deeper structures with

simpler layers. This ideology lead the VGG network to lose against GoogLeNet (Section 3.2.3), in 2014, on the ILSVRC challenge.

The innovation that propelled ResNet beyond VGG's ideology, to the point of winning against both GoogLeNet and AlexNet (Section 3.2.2), was the introduction of Residual Blocks (Figure 3.9) to the network's structure. These Residual Blocks allowed the network to become much deeper, without negatively impacting its performance.

Residual Blocks are used to re-route the current input and add it to the input learned from a previously visited layer, during the training process. In other words, as represented in Figure 3.9, a certain layer will receive information from both its adjacent layer, $\mathcal{F}(x)$, and a farther layer in the network, x. The re-route that supports a Residual Block is known as Skip Connection.

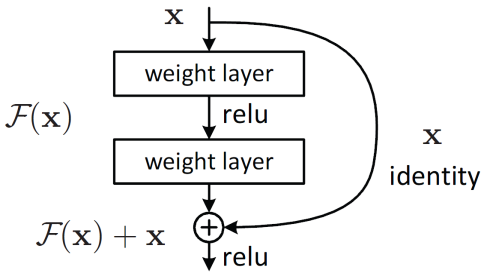


Figure 3.9: Residual Block [He et al., 2016a].

Through the use of these Residual Blocks, the network was able to propagate information smoothly from the input layer to the output layer. They are used to simplify the optimization of the network's weights and to prevent the vanishing gradient problem.

The ResNet structure used in the 2015 ILSVRC challenge had 152 layers, which was ten times deeper than most of the networks at that time. Being a network with simpler layers, most of ResNet's layers were Convolutional. Similarly to the smaller version of the ResNet shown in Figure 3.10, the 152-layer ResNet also had a Pooling Layer, performing Average Pooling, and a Fully-Connected layer, with Softmax and 1000 neurons, as its last layers.

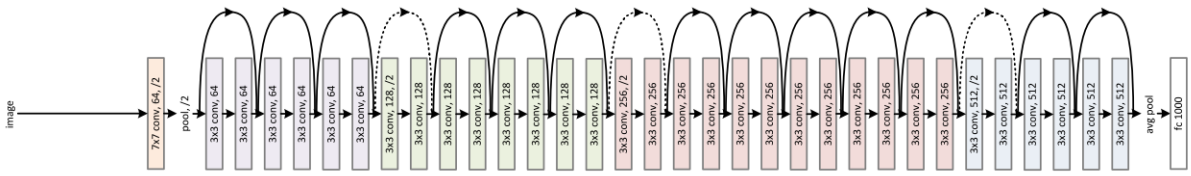


Figure 3.10: Example of a 34-layer ResNet [He et al., 2016a].

Later, it was possible to extend the ResNet used for the ILSVRC dataset [Deng et al., 2009] into a 200-layer network [He et al., 2016b]. The ResNet structure was further deepened to a 1001-layer

network, that could learn from the CIFAR dataset[3] [Krizhevsky et al., 2009] with great success.

In 2016, Andreas Veit, Michael J. Wilber and Serge Belongie explained that ResNet behaves like an ensemble of independent shallow networks, whose structures are less deep than the ResNet where they are integrated into [Veit et al., 2016]. This helped understand why the gradient is able to successfully go through such deep structures without vanishing.

3.2.5 Discussion

Table 3.2: Comparison of the aforementioned CNN milestones.

Model	Year	Number of Layers	Innovation	Dataset
LeNet-5	1998	8	Convolution Operation; Gradient Descent and Backpropagation Algorithms.	MNIST
AlexNet	2012	12	ReLU Activation Function; Max Pooling; Parallelization of the training process between two GPUs.	ImageNet
GoogLeNet	2014	22	Inception Modules; Auxiliary Classifiers.	ImageNet
ResNet	2015	152	Residual Blocks.	ImageNet
ResNet	2016	200	Improvements over the Residual Blocks' Skip Connections.	ImageNet
ResNet	2016	1001	Improvements over the Residual Blocks' Skip Connections.	CIFAR-10 CIFAR-100

Considering the aforementioned milestones, we can easily infer that the tendency in state-of-the-art CNNs has been the increase of the structures depth. Observing Table 3.2, we can see that in less than 20 years we went from an 8-layer network (1998 LeCun's LeNet-5) to a 1001-layer network (2016 Microsoft's ResNet), which represents an increase in depth of over 125 times.

Each new milestone introduced innovations that allowed models to learn more complex datasets. Those innovations made it possible to move from learning a dataset of simple grayscaled (1-channel color space) images, belonging to one of 10 possible labels (MNIST), to a dataset of much more complex colored (3-channel color space) images, belonging to one of 100 different labels (CIFAR-100).

Although, the deeper networks, such as GoogLeNet and ResNet, managed to create solutions to fix the vanishing gradient problem, which is inherently aggravated the deeper the structure of the model is, another problem emerged. That problem consists of the computational power required to train such models, which worsened with each increase of the CNN's depth.

The computational power may not be an urgent problem to the creators of GoogLeNet and ResNet (Google and Microsoft, respectively), because these companies are in the vanguard of computer science and, consequently, have the necessary resources to train these CNN's at their disposal. It is highly probable that smaller companies or even research centers do not have a fraction of those resources, which limits their efficient use of networks with this increasing depth.

[3] CIFAR-10 and CIFAR-100 datasets

A possible solution may be to resort to computer vision techniques in conjunction with simpler models, such as ANNs, in order to extract only certain features from the datasets, making the data samples smaller and easier to process by the model. This thesis intends to ascertain if this is a viable solution to the problem of the computational power required to train deeper CNNs.

3.3 Main Shortcomings

In this section we will explore some of the main shortcomings found in state-of-the-art Convolutional Neural Networks, detailed in the 2019 Annual Review *"Deep Learning: The Good, the Bad, and the Ugly"* [Serre, 2019].

According to the aforementioned annual review, the main problems present in state-of-the-art Convolutional Neural Networks are:

- These structures tend to easily make misclassifications when the images have minor noise perturbations, that are usually unnoticeable to the human eye;
- Although the deeper the network, the more complex features it can learn, this also negatively affects its capability to generalize beyond the samples used to train them;
- State-of-the-art CNNs are not capable of abstraction, which justifies their limited ability to learn abstract relational rules beyond "rote memorization" [Torralba et al., 2011].

From the listed shortcomings, we will focus on the degradation of the generalization capabilities, associated with the enormous depth of modern Convolutional Neural Networks.

In the past few years, networks such as GoogLeNet (Section 3.2.3) and ResNet (Section 3.2.4) began to present deeper and deeper structures. This has been rapidly increasing, with some modern CNNs containing thousands of hidden layers, and consequently tens of millions of parameters [Serre, 2019].

The ever increasing depth is becoming problematic beyond image classification tasks. Recent visual recognition tasks, such as object localization, semantic segmentation and depth estimation, require a greater quantity and complexity of features. Having such demanding requirements to correctly learn from data, modern CNNs need an even greater number of labeled samples, which in most cases is not possible to obtain.

4

Implementation

In this chapter, we will present the solution proposed in this thesis, exploring each step of its implementation.

Considering the problems of generalization and shortage of large labeled datasets, stated on Section 3.3, we propose a solution that aims to learn the features present in a dataset of images, without requiring the use of a deep CNN, such as the models explored in Section 3.2. This solution also aims to solve the problem of the computational power required to train deeper CNNs, mentioned in Section 3.2.5.

The proposed solution, shown in Figure 4.1, consists of a classification architecture with the following three modules:

- **Feature Extraction** - the features (color, texture and shape) are extracted from the images, resorting to computer vision techniques, such as those explored in Section 2.1;
- **Feature Learning** - each feature category (color, texture and shape) will be learned separately, by independent Neural Network models. This module will have three different machine learning models, each responsible of learning one of the modalities;

- **Multimodal Fusion** - the probabilistic outputs, which result from each model of the Feature Learning module, are combined using a Multimodal Fusion technique, in order to predict the final label that should be assigned to each image. Since this module is responsible for the label prediction for the images, it will only be used on the images which belong to the test set.

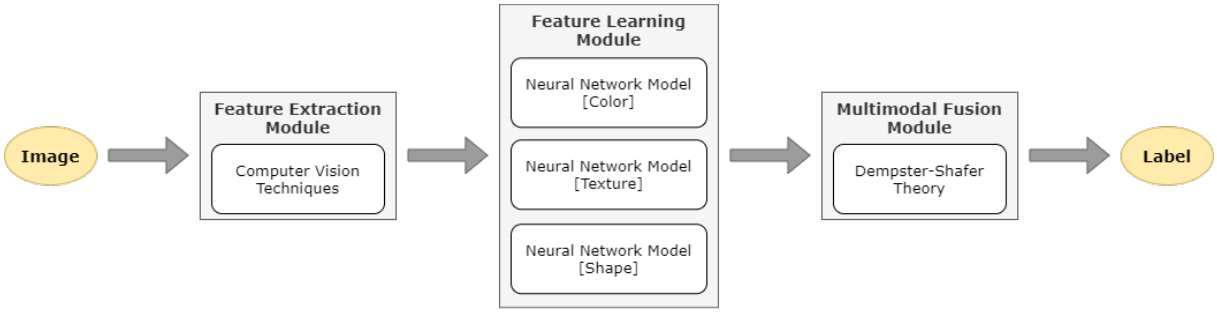


Figure 4.1: Block Diagram of the Proposed Classification Architecture.

This idea of learning different modalities independently was inspired by the cells found in the human eye's retina [Wandell and Thomas, 1997]:

- **Rods** - cells that react to light, detecting different intensities. Intensity variations help us recognize different shapes and even textures, even when we are under dim lights or outside at night;
- **Cones** - cells that react to light, detecting the wavelengths that we interpret as the different colors from the visible spectrum. Our retina has three types of cones, whose purpose is to independently capture each of the colors red (R), green (G) and blue (B).

Using independent models for each feature category, also known as modality, allows us to avoid the problem of certain features negatively impacting the others. This problem usually arises between the light intensity and color, because the former can induce humans to mistakenly identify the color of certain objects, depending on how intense is the light used to illuminate those objects.

Additionally, using this approach we aim to construct models capable of better generalizations, since each one of them will exclusively focus on learning a single modality.

In the Multimodal Fusion module, we will experiment with the omission of one of the modalities, in order to evaluate if the proposed solution is able to correctly classify the images, even in these conditions.

The proposed classification architecture, shown in Figure 4.1, will be compared to Convolutional Neural Networks, which will be trained in the same circumstances: dataset, activation functions, loss function and epochs. These CNNs will serve as a initial baseline.

This chapter is structured in the following sections:

- **CNNs - Creating a Baseline** (Section 4.1) - creation of the baseline CNNs and exploration of the dataset and tools used for the implementation;

- **Features - Extraction and Learning** (Section 4.2) - exploration of the implementation of both Feature Extraction and Feature Learning modules;

- **Multimodal Fusion** (Section 4.3) - exploration of the implementation of the Multimodal Fusion module.

4.1 CNNs - Creating a Baseline

As a starting point, we trained Convolutional Neural Networks with the chosen dataset. These CNNs would serve as a point of comparison in terms of evaluation metrics, which would be used as a baseline for the models trained in the Feature Learning module.

We will start by exploring the dataset and tools used to train these baseline models, following with an explanation of the creation of these CNNs.

4.1.1 Dataset

For the training of the baseline CNNs and the models from the Feature Learning module, we used the Fruits 360 dataset [Mureşan and Oltean, 2017]. This dataset consists of a total of 90,380 images, each with a size of 100x100 pixels, distributed over 131 labels.

The dataset's name is due to the images being mostly of fruits, although there are also some vegetables. The "360" in the name of the dataset represents the process by which the images were captured. Each image was extracted from a 20 second short video, which was recorded with each fruit/vegetable planted on top of a low-speed motor (3 rpm), with a white sheet of paper serving as a background.

In the Figure 4.2, we can observe examples of the samples present on the Fruits 360 dataset.

As we can observe in Figure 4.2, some of the labels are similar, only being differentiated by a number at the end of the label. Contrary to labels such as "Apple Braeburn" and "Apple Pink Lady", which represent different varieties of apples, the labels "Apple Golden 1", "Apple Golden 2" and "Apple Golden 3" represent the same variety of apple. For that reason, we merged labels in those situations, resulting in a decrease from 131 to 113 labels.

The dataset has its images already separated between a train and test sets. The train set has 67,692 images, while the test set has the remaining 22,688 images. This represents a train-test split of approximately 75%-25%. To create a validation set, which will be used to validate the training process of the classification models, we applied a split of 75%-25% to the train set, preserving the same proportions of samples from each label as observed in the original train set. This resulted in a new train set with 50,769 images and a validation set with the remaining 16,923 images.

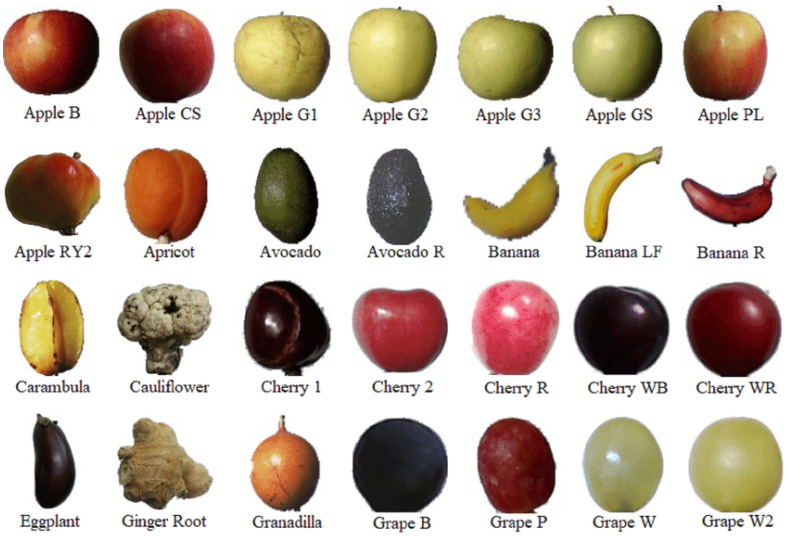


Figure 4.2: Representation of some of the samples from the Fruits 360 dataset [Dandekar et al., 2021].

4.1.2 Frameworks

For the creation and training of the baseline and Feature Learning models, we resorted to the Keras framework. Keras is a high-level framework that works as a wrapper to the TensorFlow framework.

4.1.2.A TensorFlow

TensorFlow is an end-to-end open-source framework used for building, training and deploying machine learning models. This framework includes a set of tools and libraries, among other resources, which provide workflows with high-level APIs, such as the framework Keras.

The main advantages of this framework are:

- TensorFlow offers various levels of abstraction to build and train machine learning models;
- TensorFlow is not bound to a specific programming language or platform;
- TensorFlow is research-friendly, that is, it offers powerful experimentation tools which gives the user flexibility and control over the creation of more complex machine learning structures.

4.1.2.B Keras

Keras is an open-source high-level framework dedicated to Neural Networks and runs on top of frameworks like TensorFlow and Theano. The main purpose of Keras is to allow an easy and fast creation of Deep Learning Networks, which can be trained on a CPU or GPU.

The main advantages of this framework are:

- Keras is user-friendly, which means it has a simple and consistent interface optimized for the user experience. One of the best aspects of its interface is the clear and actionable feedback provided to the user, in the event of errors;
- Keras allows the user to create models by connecting custom building blocks together, which is useful when prototyping or researching new ideas.

The Neural Networks presented on this thesis were built using both Keras' Models and Layers APIs. From the Model API we used the class *Sequential* (Feed-Forward Neural Network) and from the Layers API we used the classes *Dense* (common Hidden Layer), *Conv2D* (Convolutional Layer) and *MaxPooling2D* (Pooling Layer with Max Pooling as the downsampling technique).

We also used resources from Keras' Optimizers and Losses libraries, such as the *SGD* (Stochastic Gradient Descent) and the *CategoricalCrossentropy* (Loss Function used for all models), respectively.

4.1.3 Baseline

We chose two models to serve as a baseline for the proposed solution. Both models are Convolutional Neural Networks which we previously created to learn a subset of the Fruits 360 dataset (Section 4.1.1). This subset consisted of 14,419 images distributed between 4 super-labels: Apples, Lemons, Oranges and Pears. These new super-labels represent groups of labels from the original dataset, such as:

- **Apples** - all original labels associated with types of "Apples": Braeburn, Crimson Snow, Golden, Granny Smith, Pink Lady, Red, Red Delicious and Red Yellow;
- **Lemons** - all original labels associated with types of "Lemons": Normal and Meyer;
- **Oranges** - original label "Orange";
- **Pears** - all original labels associated with types of "Pears": Abate, Forelle, Kaiser, Monster, Red, Stone and Williams.

We named these two models "Baseline Alpha" and "Baseline Beta", and their structures are represented in the Tables A.1 and A.2, on the Appendix "Neural Network Structures".

These two models have some similarities, such as: only having one Convolutional Layer; using Softmax as the activation function on the Output Layer; and using the Cross-Entropy Loss Function on the backpropagation algorithm. The main structural differences between them is that the "Baseline Beta" model has a Pooling Layer, which performs the MaxPooling downsampling, and uses the Dropout regularization technique in two sections of the structure.

The other differences between the baseline models are related to how we trained them over the custom subset of the Fruits 360 dataset. These differences are shown in the Table 4.1.

Table 4.1: Comparison between the Baseline Models, over the Fruits 360 subset.

Model	Dataset Split	Image Resize	Train Configuration	Test Accuracy
Baseline Alpha	Train = 60% Validation = 20% Test = 20%	75%	Epochs = 10 Batch Size = 128	99.62%
Baseline Beta	Train = 56.25% Validation = 18.75% Test = 25%	80%	Epochs = 10 Batch Size = 32	98.78%

Observing the Table 4.1, we can see that, although these models followed different approaches for the resize of the images, before the training process, and for the dataset split into the train, validation and test sets, they achieved similar values of test accuracy, differing in less than 1%.

In order to serve as a baseline, both these models were trained over the complete dataset, without resizing the images and following the train-validation-test split defined in the Section 4.1.1.

The only pre-processing done to the images was the normalization of their values. Considering the images consist of three color channels, each encoded in 8 bits, we normalized each image by dividing its values by 255, which is the maximum value of color intensity represented by that encoding.

We explore this experiment and its results in the Section 5.2.

4.2 Features - Extraction and Learning

The main features we can extract from images are color, texture and shape, as previously explored in Section 2.1. Each of these types of features are considered modalities and will be extracted independently from each other. The extraction will resort to computer vision techniques such as multi-channel color histograms, image gradients and edge detection, depending on the feature to be extracted.

After the feature extraction process, the resulting feature vectors will be used to train the Neural Network models created for each modality.

4.2.1 Tools

For the Feature Extraction module, we used the open source library OpenCV-Python, which works as a Python wrapper for the original OpenCV implemented in C++. This library offers a rich infrastructure of optimized computer vision techniques focused on real-time applications for both image and video.

OpenCV offers tools to perform basic tasks, such as capturing, filtering and storing both images and videos, but it also offers a wide range of options for more complex tasks. These complex tasks can go from object detection and segmentation in images, to car detection or face recognition in a real-time video stream.

When extracting the different types of features, we resorted to OpenCV for some of its filtering and image processing tools, but also for its image analysis tools, which proved to be very useful to easily

visualize the effects of each technique on the processed images.

4.2.2 Color Modality

To represent this modality, we chose to extract the color feature using color histograms. The images of the Fruits 360 dataset are represented in the RGB color space, which means that each image is composed of three channels: Red, Green and Blue.

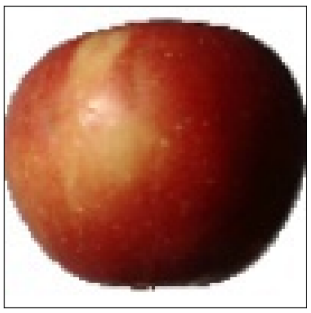
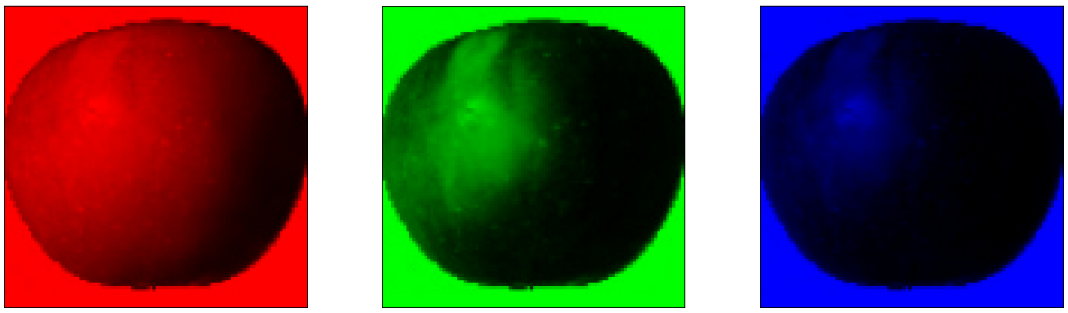


Figure 4.3: Example of an image with the label "Apple Braeburn", from the Fruits 360 dataset.

In order to explain the process used to extract the color information from the images, we will use as an example the sample presented in Figure 4.3.

Firstly, we separated each image into the three respective color channels. This process transformed the images from 3-channel matrices to three individual 1-channel matrices, as shown in Figure 4.4.



(a) Red Channel. **(b)** Green Channel. **(c)** Blue Channel.

Figure 4.4: Representation of the 3 color channels extracted from the image shown in Figure 4.3.

After separating the color channels, we computed a color histogram for each of them. The color histogram represents the quantity of pixels, from a certain image, that are assign to each level of color intensity

Since the images from this dataset are stored using an 24 bit encoding, each channel uses 8 bits to represent the color intensity. This means that our color histograms will use a range of bins from 0 to 255, having 256 (2^8) levels of intensity.

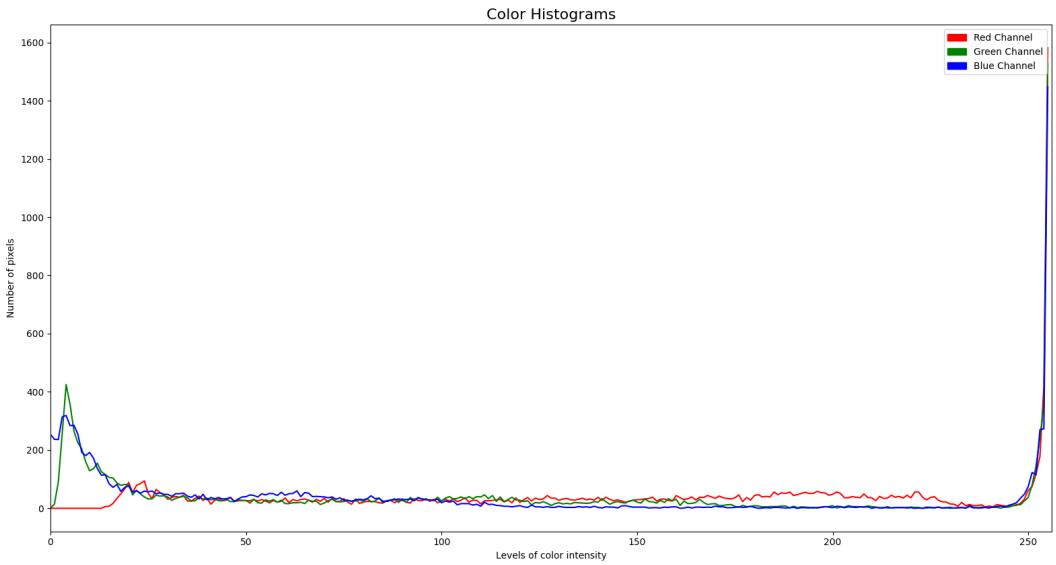


Figure 4.5: Color histograms extracted from the image shown in Figure 4.3.

In the Figure 4.5, we are able to compare the color histograms of each channel. We can see that between the intensities 20 and 250, all three color histograms have less than 200 pixels per level. Another fact we can also observe from Figure 4.5 is that all three color channels present the highest pixel counts for the intensity levels above 250. This is justified by all images presenting a completely white background.

Before using these histograms to learn the color modality, first we had to normalize their values, to better fit the Neural Network models. We considered to normalization techniques: Probabilistic Normalization and Min-Max Normalization. The former transforms the histograms into a probabilistic model, in which its values are divided by the total amount of pixels. On the other hand, the latter technique maps the histogram values into a new range, defined by a given minimum and maximum values.

When comparing these normalization techniques, we concluded that, although both were capable of achieving the objective of normalizing the histogram values into a new range between 0 and 1, the Probabilistic Normalization transformed most of the values into small probabilities, very close to the minimum value (0).

Having decided on the normalization technique, we created the color feature vectors for each data sample. These features vectors are a concatenation of the normalized color histograms and have a size of 768, which resulted from the sum of the previous three color histograms' sizes (256).

In the Figure 4.6, we present the final histogram, which was used as the color feature vector for the example image (Figure 4.3). As previously explained, this is the resulting histogram after concatenating and normalizing the color histograms shown in Figure 4.5.

We built three Neural Networks to train over the extracted color feature vectors. These models have

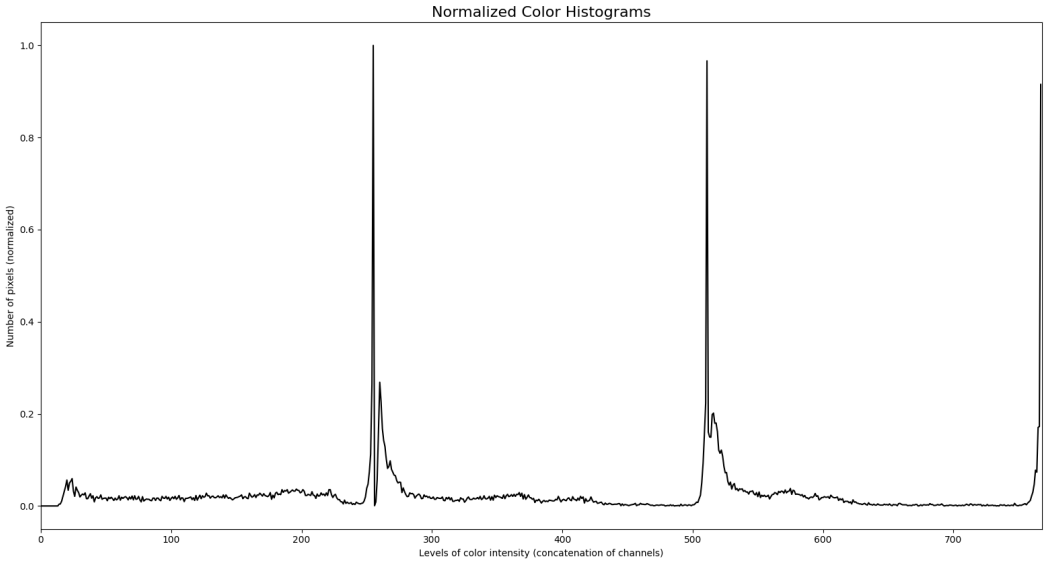


Figure 4.6: Concatenation of the normalized color histograms.

the names “Color Alpha”, “Color Beta” and “Color Gama”, and their structures are represented in the Tables A.3 to A.5, on the Appendix “Neural Network Structures”. Both the “Color Alpha” and “Color Beta” models only have one hidden layer, being differentiated by the number of hidden units. The latter model has double the hidden units (256) of the “Color Alpha” model (128). The “Color Gama” model has two hidden layers: the first with 384 units and the second with 192 units.

We explore the experiment of training these models and its respective results in the Section 5.3.1.

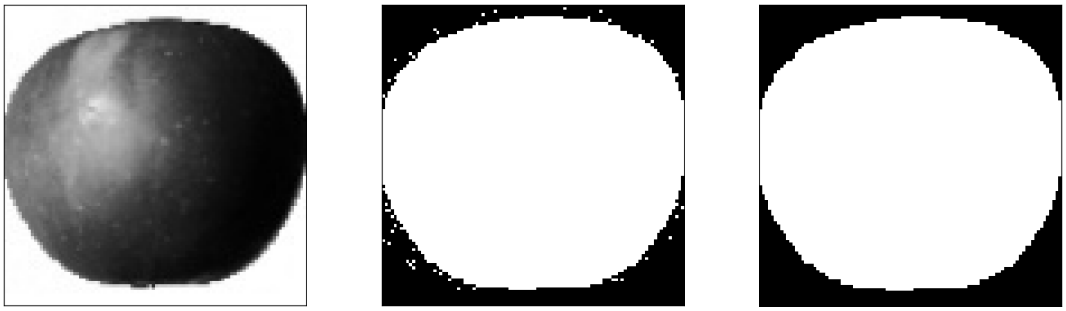
4.2.2.A Improving the Feature Vector

After training the Color Modality models and analyzing the results computed by the loss and accuracy metrics, in Section 5.3.1, we explored the process of extracting the color information, trying to find what could be causing such unsatisfactory results.

When we observed the color histograms for a second time, we started to think if the aforementioned fact that the highest pixel counts were found for intensity levels above 250, could be the what is negatively impacting the metrics. This would make sense, because if the levels between 250 and 255 have by far the highest values, the NNs models will interpret these values as being more important and would associate them with higher weight values during the training process.

We decided to remove the background from the images and repeat the experiment to see if this was truly the problem. The chosen approach to achieve this objective was to create a mask of each image, which would be used to ignore the pixels from the background area of the image, during the computation of the color histograms.

In order to create the masks, these are the steps we followed:



(a) Grayscale Conversion. **(b)** Binary Conversion. **(c)** Mask.

Figure 4.7: Steps for creating a Mask from the image shown in Figure 4.3.

- **Color Space Conversion** - in this step, we converted the images from the RGB to the Grayscale color space. This transformation results in a single channel image, whose values are computed with (4.1). In the Figure 4.7(a), we present an example of the results from this step;
- **Image Thresholding** - in this step, we applied the Binary Inverse Thresholding approach to the grayscale images [Gonzalez et al., 2002]. This thresholding approach converts pixel intensities to 0, if their value is above the defined threshold, or to 1, otherwise. Considering we intended to remove the white background, the value we chose for the threshold was 250. This transformation results in a binary image. In the Figure 4.7(b), we present an example of the results from this step;
- **Morphological Transformation** - observing the results from the Image Thresholding step, we noticed that the images had some noise around the fruits' silhouette. This step aims to clear that noise from the binary images, through use of the Opening Morphological Operator [Comer and Delp III, 1999]. This Morphological Operator uses a structured element (also known as kernel) to remove small areas of activated pixels (with value 1) which are situated around bigger areas of activated pixels, associated with objects present in the image. The kernel is used across the image, computing convolutions which result in the activation or deactivation of the pixels. We used the kernel in (4.2) for the Opening operation. In the Figure 4.7(c), we present the binary mask that results from applying this sequence of steps to the image from Figure 4.3.

$$\text{Grayscale} = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B \tag{4.1}$$

$$Kernel_{Opening} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{4.2}$$

Having created the masks for every image in the dataset, we computed the new color histograms.

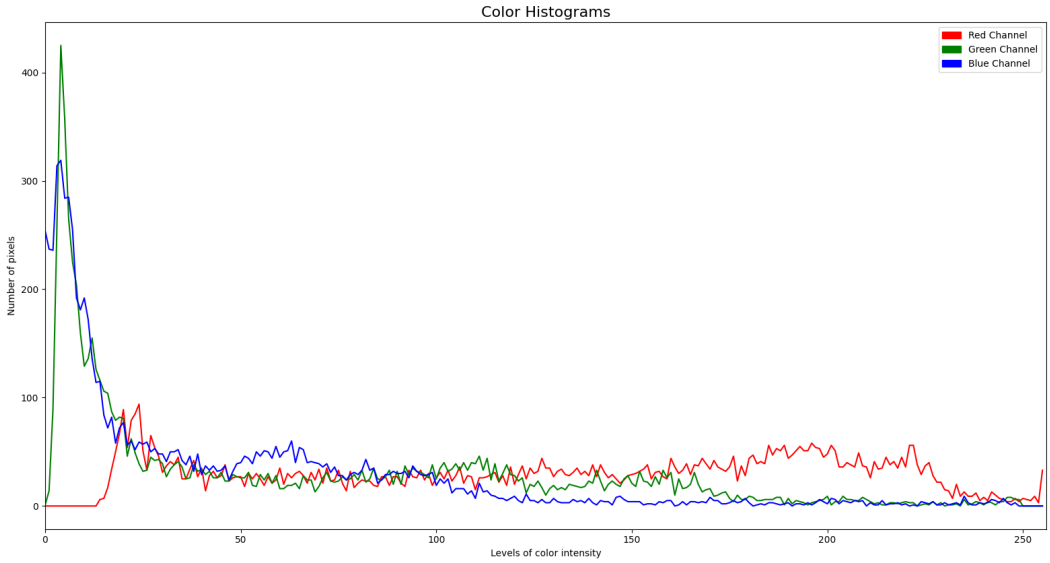


Figure 4.8: Updated color histograms extracted from the image shown in Figure 4.3

These updated color histograms were computed using the masks to decide which pixels should be considered. The decision consisted of if a pixel falls inside the activated region of the mask, then that pixel is used for the histogram calculation, otherwise it will be ignored.

Observing the color histograms, in Figure 4.8, we can clearly notice that the mask was successful in removing the background, since there is a great decrease of pixel counts for the intensity levels above 250, in comparison to the color histograms from Figure 4.5.

The next step was to normalize and concatenate the new color histograms, exactly as we did previously, in order to create the new feature vectors. The updated feature vectors were used in the a new experiment, where we resorted to the same training configurations to evaluate if removing the images' background would improve the results for the same models (Section 5.3.1.A).

4.2.3 Texture Modality

To represent this modality, we chose to extract the texture feature using Histograms of Oriented Gradients (HOG) [Dalal and Triggs, 2005].

In order to explain the process used to extract the texture information from the images, we will use as an example the sample presented in Figure 4.9.

Firstly, we transformed the images from the RGB color space to the Grayscale color space. This is a required pre-processing to compute the image gradient.

Having the images in grayscale, we computed the image gradient for all samples, using the Sobel operator [Sobel and Feldman, 1968]. As explored in Section 2.1.2, the gradient of an image is computed as the first-degree derivative of a 2-dimensional continuous function. The Sobel operator adds to this

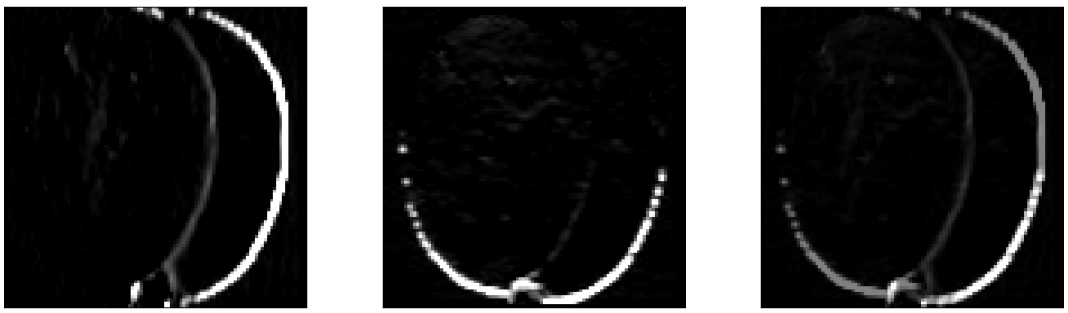
Figure 4.9: Example of an image with the label “Apricot”, from the Fruits 360 dataset.

notion, computing the derivative through a convolution operation between the image and the respective Sobel kernels. The kernels used in the convolution operation depend on the kernel size chosen in the operator. We experimented with the kernel sizes 3 and 5, which were assigned to the kernels (4.3) and (4.4), respectively.

$$M_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad M_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{4.3}$$

$$M_x = \begin{bmatrix} 2 & 1 & 0 & -1 & -2 \\ 2 & 1 & 0 & -1 & -2 \\ 4 & 2 & 0 & -2 & -4 \\ 2 & 1 & 0 & -1 & -2 \\ 2 & 1 & 0 & -1 & -2 \end{bmatrix} \qquad M_y = \begin{bmatrix} 2 & 2 & 4 & 2 & 2 \\ 1 & 1 & 2 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ -1 & -1 & -2 & -1 & -1 \\ -2 & -2 & -2 & -2 & -2 \end{bmatrix} \tag{4.4}$$

From the application of the Sobel operator to an image, we obtain two matrices: the vertical and horizontal image gradients.



(a) Vertical Gradient. **(b)** Horizontal Gradient. **(c)** Gradients combined.

Figure 4.10: Resulting gradients, from applying the Sobel operator (k=3) to the image shown in Figure 4.9.

Observing the image gradients, shown if Figures 4.10 and 4.11, we can clearly see that using a kernel size of 5, associated with the kernels (4.4), detects a lot more noise surrounding the edges found on the image gradients. Using a kernel size of 3 (4.3), we can obtain the same edges with significantly

(a) Vertical Gradient. **(b)** Horizontal Gradient. **(c)** Gradients combined.

Figure 4.11: Resulting gradients, from applying the Sobel operator (k=5) to the image shown in Figure 4.9.

less noise. Although, we can still detect some nuances of textures inside the object. After comparing the results from both kernel sizes, we chose to use the size 3 for the kernels of the Sobel operator.

The next step is to calculate the gradient magnitude and orientation, for each pixel in the image. To calculate the values of magnitude and orientation, we use the following formulas:

$$Magnitude = \sqrt{Grad_x^2 + Grad_y^2} \tag{4.5}$$

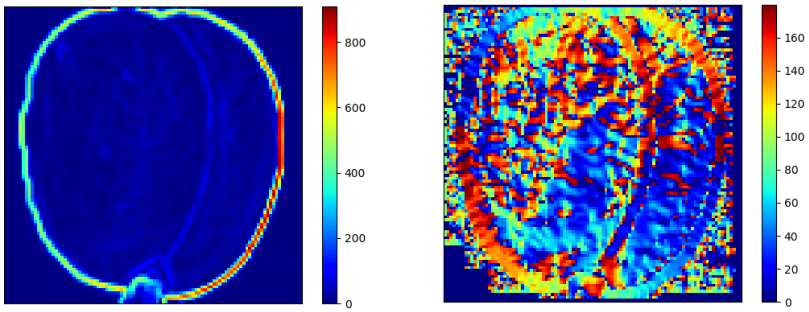
$$Orientation = \left| \tan^{-1} \left(\frac{Grad_y}{Grad_x]} \right) \right| \tag{4.6}$$

Using the vertical and horizontal gradients to compute the magnitudes (4.5) and orientations (4.6) of the gradient, results in two new matrices, with the same size of the original image. From the gradient magnitude matrix, represented in Figure 4.12(a), it is possible to verify that the edges of the object have the highest magnitude values, while the pixels from the background have a zero value magnitude. Observing the gradient orientation matrix, represented in Figure 4.12(b), we can also verify that the highest angle values are distributed across the silhouette of the fruit, considering its roundness.

With the magnitude and orientation matrices, we can finally create the Histogram of Oriented Gradients. The HOG is a computer vision technique used to represent texture information and is created following the steps below:

1. We start by dividing the magnitude and orientation matrices into blocks. These blocks can have a custom size, and for the current example we will consider a size of 10x10, since it is easier to visualize for an image with size 100x100. This results in 100 blocks, since we have a length of 10 blocks both vertically and horizontally;

2. For each block, we create an oriented gradient histogram. The bins for this histogram represent the orientations and the values represent the sum of magnitudes related with those orientations. The



(a) Gradient Magnitudes. **(b)** Gradient Orientations.

Figure 4.12: Representation of the Gradient Magnitude and Orientation matrices.

number of bins for the histogram can have a custom value, which will affect the step size between bins. Considering the bins represent the orientation, which ranges from $0°$ to $180°$, we will use 9 bins for the current example. With a 9-bin histogram, the steps between bins will be of $20°$;

3. Since this is not a common histogram, like the color histograms explored in Section 4.2.2, we have to consider that each bin is selected by the orientation value and the value that is added to the chosen bin is selected from the magnitude matrix. This means that, if the gradient orientation of a certain pixel is not the exact value attributed to one of the bins, the magnitude value associated with that pixel will be proportionally split between two bins. Considering the example of a magnitude value of 60 associated with a orientation of $170°$, that magnitude will be split 50-50% between the bins 160 and 0 (30 for each), since $170°$ is at a distance of $10°$ from both $160°$ and $180°$. Even though the orientation value is between 160 and 180, we are only considering angles from $0°$ to $180°$, which means the angle will wrap around, making $0°$ and $180°$ equivalent;

4. Having computed the histograms for each of the 100 blocks, we will have to normalize the blocks, to prevent possible lighting variations captured by the gradient of the image. To normalize the blocks we use a window of size 2x2 blocks which will slide across the image, concatenating the 4 histograms contained inside the window. The concatenation of the histograms will result in a vector with size 36x1 (4 histograms with 9 bins each). This vector will then be normalized with the L2-norm (4.7);

5. Finally, the HOG is created by concatenating all the normalized vectors, which for the given example would have a dimension of 2916x1. We can visualize the HOG from this example shown in Figure 4.13.

$$\text{L2-norm: } \frac{\overrightarrow{v}}{\sqrt{\|\overrightarrow{v}\|^2] + \varepsilon}}, \text{ where } \varepsilon \text{ is a small value added to prevent zero division} \tag{4.7}$$

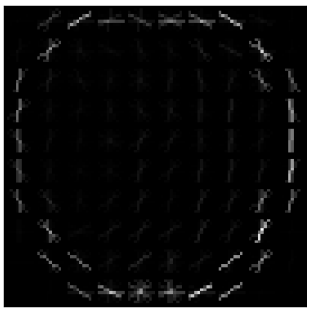


Figure 4.13: Visualization of the example HOG created from the image shown in Figure 4.9

We built three Neural Networks to train over the extracted texture feature vectors. These models have the names “Texture Alpha”, “Texture Beta” and “Texture Gama”, and their structures are represented in the Tables A.6 to A.8, on the Appendix “Neural Network Structures”. Both the “Texture Alpha” and “Texture Beta” models only have one hidden layer, being differentiated by the number of hidden units. The latter model has more 128 hidden units (384) than the “Texture Alpha” model (256). The “Texture Gama” model has two hidden layers: the first with 384 units and the second with 192 units; similarly to the structure of the “Color Gama” model (Table A.3).

We explore the experiment of training these models and its respective results in the Section 5.3.2.

4.2.4 Shape Modality

To represent this modality, we chose to extract the shape feature using a Shape Map. The Shape Map is our proposed solution for representing shape information and consists of an aggregation of multiple features associated with the shape feature. We propose a Shape Map which combines the silhouette of the object, the edges detected with the Canny algorithm [Canny, 1986] and the corners detected with both the Harris [Harris et al., 1988] and Shi-Tomasi [Shi et al., 1994] algorithms.

In order to explain the process used to extract the shape information from the images, we will use as an example the sample presented in Figure 4.14.

Figure 4.14: Example of an image with the label “Banana”, from the Fruits 360 dataset.

Similarly to the texture extraction, we started by transforming the images from the RGB color space

to the Grayscale color space, since this is a required pre-processing for the extraction of the shape features that we chose to explore.

4.2.4.A Silhouette Detection

Figure 4.15: Silhouette of the image shown in Figure 4.14.

In order to detect the objects silhouette, we resorted to the previously implemented approach of removing the image background (Section 4.2.2.A). However, instead of creating the mask to remove the white background from the image, we use the same mask as a representation of the fruits silhouette. The silhouette is composed of active pixels (value 1), which represent the object, and inactive pixels (value 0), which represent the area outside of the detected object.

Creating a background removing mask from the example image, shown in Figure 4.14, results in the silhouette binary image represented in the Figure 4.15.

4.2.4.B Edge Detection

When detecting edges from an image, we have to consider that this process is greatly susceptible to noise in the image. To prevent this problem, we removed possible noise in the images with a 3x3 Gaussian filter. The 3x3 kernel used on this filter is created with the Gaussian function $G(x, y)$ (4.8).

$$G(x,y) = \alpha e^{\frac{-(x-\mu_x)^2}{2\sigma_x^2} + \frac{-(y-\mu_y)^2}{2\sigma_y^2}} \tag{4.8}$$

In this Gaussian function, α is a scale factor chosen to guarantee that (4.9) is true.

$$\sum_{x,y} G(x,y) = 1 \tag{4.9}$$

After filtering the images, we used Canny's Edge Detector to extract the edge information. This edge detection algorithm processes the images through the following steps:

1. **Image Gradient** - the algorithm computes both the horizontal and vertical image gradients with a

Sobel operator, and from those gradient matrices computes the gradient magnitudes and orientations, similarly to what we did to extract the texture information (Section 2.1.2);

2. **Non-maximum Suppression** - the algorithm scans the image removing pixels which do not belong to one of the edges. This is accomplished by checking for each pixel if its magnitude value is a local maximum among the neighbor pixels that follow the same gradient orientation. This step results in a binary image with all the detected edges;

3. **Thresholding through Hysteresis** - in this step, the algorithm filters the edges detected in the previous step, resorting to a low-threshold and a high-threshold. For each detected edge pixel, the algorithm will check where the corresponding magnitude value is situated in comparison to the defined threshold values. If the magnitude value is higher than the high-threshold, that pixel is classified as a true edge. On the other hand, if the magnitude value is lower than the low-threshold, the pixel is discarded. For the last possibility, if the magnitude of an edge pixel lies between the two thresholds, the pixel will only be classified as an edge if it is connected to an already confirmed edge.

(a) Thresholds (20,140). **(b)** Thresholds (30,120).

Figure 4.16: Canny Edge Detection applied to the image shown in Figure 4.14.

After experimenting with the values for the Canny Edge Detector's thresholds, we present some examples, of the application of this algorithm, in Figure 4.16. For the example in Figure 4.16(a), we chose 20 as the low-threshold and 140 as the high-threshold. For the second example, in Figure 4.16(b), we increase the low-threshold to 30 and decreased the high-threshold to 120. Both configurations were able to clearly detect the outside contour of the object. However, the configuration shown in Figure 4.16(b) managed to detect more edges inside of the object, in comparison to the configuration from Figure 4.16(a). We chose the configuration shown in Figure 4.16(b) to be applied in the extraction of the edge information from the images.

4.2.4.C Corner Detection

The Harris Corner Detector detects edges and corners in an image, by scanning the image, with a window function $w(x, y)$, and classifying the region inside that window as a corner, edge or flat. This classification will depend on the value of a score R.

The Harris detector computes the image gradient with the Sobel operator, which will be then used to compute the R score. This score is calculated using the formula in (4.10).

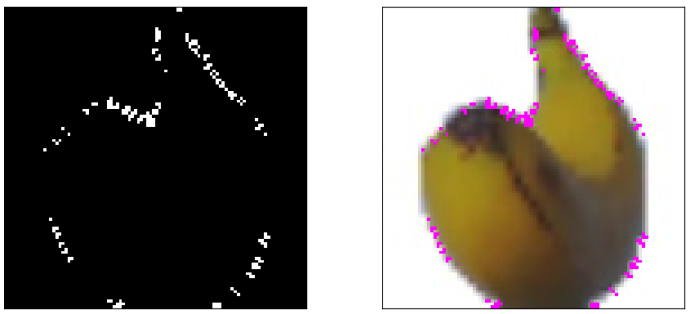
$$R = \lambda_1\lambda_2 - k(\lambda_1 + \lambda_2)^2, \text{where } k \text{ is a scale factor} \tag{4.10}$$

The variables λ_1 and λ_2 in the score R formula represent the eigenvalues of the matrix M, calculated by the equation (4.11).

$$M = \sum_{x,y} w(x, y) \begin{bmatrix} Grad_xGrad_x & Grad_xGrad_y \\ Grad_xGrad_y & Grad_yGrad_y \end{bmatrix} \tag{4.11}$$

According to the rules of the Harris Corner Detector, each region of the image is classified as:

- **Corner** - large R score, which occurs when both λ_1 and λ_2 are large values and $\lambda_1 \sim \lambda_2$;
- **Edge** - $R < 0$, which occurs when $\lambda_1 >> \lambda_2$ or $\lambda_1 << \lambda_2$;
- **Flat** - $|R|$ is small, which occurs when both λ_1 and λ_2 are small values.



(a) Corners Image

(b) Corners on Original Image

Figure 4.17: Harris Corner Detection applied to the image shown in Figure 4.14.

Using a window with size 2 and k factor of 0.025, as the parameters for the Harris Corner Detector algorithm, resulted in the corners represented on the binary image shown in Figure 4.17(a).

For the detection and extraction of corners, we also used the Shi-Tomasi Corner Detection algorithm. In this algorithm, J. Shi and C. Tomasi proposed a small modification to the Harris Corner Detector,

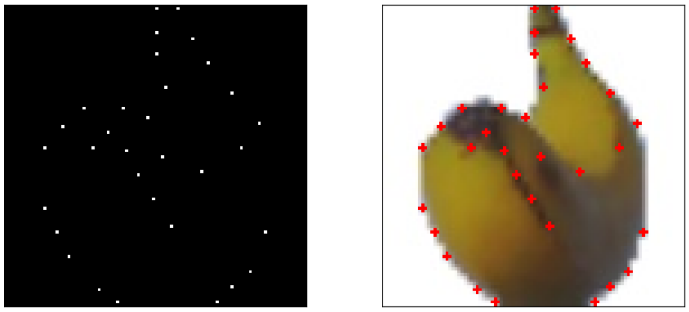
which ended up improving the results achieve by that algorithm. The proposed modification consisted in changing the scoring function R from (4.10) to (4.12).

$$R = \min(\lambda_1, \lambda_2) \tag{4.12}$$

With the presented modification to the scoring function R, this algorithm also adapted the rules used to classify each region of the image. The new rules are as follows:

- **Corner** - both λ_1 and λ_2 are above a specified minimum value (λ_{min});
- **Edge** - only one of the eigenvalues λ_1 and λ_2 is above λ_{min};
- **Edge** - both λ_1 and λ_2 are below λ_{min};

Using a λ_{min} with value 0.01, as the parameter for the Shi-Tomasi Corner Detector algorithm, resulted in the corners represented on the binary image shown in Figure 4.18(a).



(a) Corners Image

(b) Corners on Original Image

Figure 4.18: Shi-Tomasi Corner Detection applied to the image shown in Figure 4.14.

Comparing the results from both Corner Detectors, we noticed that the Harris algorithm finds more corners along the object contour, while the Shi-Tomasi algorithm is able to find corners inside the object's area.

4.2.4.D Shape Map

As aforementioned, we propose a Shape Map, that converges various types of shape information, as the feature vector for the Shape Modality. Our Shape Map will consist of a weighted aggregation of the binary images that resulted from the previously presented methods of extracting shape information.

The weight values were chosen within the range [0, 255], which will result in a grayscale image. After experimenting with multiple variations of these weights, the values chosen to be attributed to each type of shape information are as follows:

- **Silhouette** - 64;
- **Edges** - 128;
- **Harris Corners** - 224;
- **Shi-Tomasi Corners** - 255.

The weights were attributed considering the relevance of each type of shape information. Corners detected by the Shi-Tomasi detector will have a higher weight, in comparison to Harris' corners, since the former is an improvement of the latter algorithm.

The aggregation process will always choose the highest weight possible for each pixel, that is, in a scenario where a pixel is active, for example, in both the Edge and Harris Corners binary images, it will be associated with the weight of the latter. Pixels that are inactive in all four shape binary images will have a weight of 0.

When the aggregation process finishes, we obtain a Shape Map for each image. In the Figure 4.19, we present an example of a Shape Map, which resulted from processing the image shown in Figure 4.14 through all the steps, implemented in the Feature Extraction module, dedicated to the shape information.

Finally, we normalize and resize the Shape Maps. The normalization approach used consists of dividing the values by 255, adjusting their range to [0,1]. We resize the Shape Maps, by a scale of 50%, as a means of dimensionality reduction. The resized maps have a size of 50x50.

Figure 4.19: Shape Map created from the image shown in Figure 4.14.

We built three shallow Convolutional Neural Networks to train over the extracted Shape Maps. These models have the names "Shape Alpha", "Shape Beta" and "Shape Gama", and their structures are represented in the Tables A.9 to A.11, on the Appendix "Neural Network Structures". All three models have one Convolutional layer, with a kernel size (10,10). The difference between the models is the number of defined filters: "Shape Alpha", "Shape Beta" and "Shape Gama" have 8, 16 and 32 filters, respectively.

We explore the experiment of training these models and its respective results in the Section 5.3.3.

4.3 Multimodal Fusion

Multimodal Fusion is the combination of the results obtained from various independent modalities. Being an image classification task, the modalities we defined are associated with the color, texture and shape information, extracted from a dataset of images.

The Multimodal Fusion can be performed through several different methods. For example, we could simply use a weighted-sum, where the weights represent the contribution percentage that each modality has on the final value.

Initially, we thought of training a new Neural Network over an aggregation of the results from each Modality model. But this approach would append another Machine Learning model to our implementation, which would add unnecessary complexity.

We decided to rely on the Dempster-Shafer's theory of evidence to combine the Softmax probabilities, that result from the classification of the test set by each Modality model.

4.3.1 Dempster-Shafer's Theory of Evidence

With the Dempster-Shafer's theory of evidence it is possible to associate measures of uncertainty with sets of hypotheses, when the individual hypothesis are imprecise or unknown [Schocken and Hummel, 1993]. The Dempster-Shafer's rule of combination derives common shared belief between multiple sources and ignores all the conflicting beliefs [Shafer, 1976].

In the context of this thesis, our hypothesis will be if either that is the label associated with the image. All possible mutually exclusive hypothesis are contained in a frame of discernment θ. The possible combinations of hypothesis, including the empty set $\emptyset$, are represented by 2^θ.

The main advantage of using this theory of evidence, is that it supports the specification of a degree of uncertainty, instead of forcing the hypotheses to have probabilities that add to unity, just like a traditional probability theory. However, we will not need this advantage, since the results from our modalities are probabilities calculated by the Softmax Activation Function.

The Dempster-Shafer's theory of evidence allows the definition of a belief mass function m. This belief mass function is applied to each element contained in θ, following these three requirements:

- $m : 2^\theta \rightarrow [0, 1]$;
- $m(\emptyset) = 0$;
- $\sum_{A \epsilon 2^\theta} m(A) = 1$.

The probability of a label A being perceived as correct, for the image processed by a modality, is given by the confidence interval [$Belief(A)$, $Plausibility(A)$].

Belief represents the lower bound of the confidence interval and is defined as being the total evidence that supports the hypothesis. The belief is calculated as the sum of all the masses of the subsets associated with the set A (4.13).

$$Belief(A) = \sum_{B|B \subseteq A} m(B) \tag{4.13}$$

Similarly, Plausibility is the upper bound of the confidence interval and is defined as being the sum of all the masses of the set B, that intersects the set of interest A (4.14).

$$Plausibility(A) = \sum_{B|B \cap A \neq \emptyset} m(B) \tag{4.14}$$

The Dempster-Shafer theory provides a combination rule (4.16) to combine evidences detected by two different modalities S_1 and S_2.

$$m_{S_1,S_2}(\emptyset) = 0 \tag{4.15}$$

$$(m_{S_1} \otimes m_{S_2})(A) = \frac{1}{1-K} \sum_{B \cap C = A \neq \emptyset} m_{S_1}(B) m_{S_2}(C) \tag{4.16}$$

The K in the formula 4.16 measures the conflict between the two modalities and is calculated by the equation 4.17.

$$K = \sum_{B \cap C = \emptyset} m_{S_1}(B) m_{S_2}(C) \tag{4.17}$$

In the context of our thesis, the mass functions for each modality represent the probability of each label being correct for a certain image. In other words, the mass functions will be the probability values given by each Modality model.

Considering we have three modalities, one for each type of features extracted (color, texture and shape), we had to apply the Dempster-Shafer's combination rule in two iterations. On the first iteration, we simply used the mass functions of two modalities. For the second iteration, applied the combination rule between the mass functions from the remaining modality and the results from the first iteration. This results in combined mass functions that represent the fusion of the Color, Texture and Shape Modalities.

Lastly, the Multimodal Fusion module will predict the final labels for the test set. This prediction is done by choosing, for each image, the label assigned to the highest value from the combined mass functions.

In the Section 5.4, we explore the results from using the Dempster-Shafer's theory of evidence as the Multimodal Fusion approach that combines the results produced by our Modality models.

5

Experimental Evaluation

In this chapter, we will explore the experiments performed during the implementation of this thesis. For each experiment, we will present the resulting values for the chosen evaluation metrics, accompanied by a critical analysis and discussion.

This chapter is structured in the following sections:

- **Evaluation Metrics** (Section 5.1) - exploration of the metrics used to evaluate the performance of each experiment;
- **Training the Baseline CNNs** (Section 5.2) - presentation of the experiments performed during the train process of the baseline models and analysis of the results;
- **Learning the Extracted Features** (Section 5.3) - presentation of the experiments performed for the extraction and learning of each modality. Discussion about the results and justification of the chosen models for each modality;
- **Multimodal Fusion** (Section 5.4) - presentation of the experiments performed for the fusion of modalities and analysis of the results;

- **Results Analysis** (Section 5.5) - Deep analysis of the final results. Comparison between the obtained results and state-of-the-art models trained over the same dataset.

5.1 Evaluation Metrics

In order to evaluate the performance of our models during the training process, we will rely on both the loss and accuracy metrics. However, when evaluating the final results of the proposed solution, we will rely on both precision and recall, as label focused metrics, and on accuracy, as the overall performance metric.

5.1.1 Loss Metric

The loss metric represents the total cost of mapping the predicted labels onto the expected labels, for a complete train cycle (epoch) over a set of data samples. The farther the loss metric is from zero, the worst a model is performing during the training process. We compute this metric with the Cross-Entropy Loss Function used in the backpropagation algorithm of all the built models. As mentioned in Section 2.2.1, this loss function conditions the NNs computed outputs to be values between 0 and 1, which is achieved using the Softmax Activation Function. The Cross-Entropy Loss Function computes the loss value following the formula (2.13), presented at the end of Section 2.2.1.

5.1.2 Accuracy on Binary Classification Tasks

The accuracy metric represents the percentage of correctly classified samples and can be extracted from a confusion matrix. The confusion matrix represents the distribution of each sample accordingly to how it was classified by the model, considering the original labels. For a binary classification task, one of the labels is considered as being positive, while the other as being negative.

The accuracy for a binary classification task is calculated as:

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{5.1}$$

The categories shown in Figure 5.1 and used in (5.1) to calculate the accuracy metric, are associated to the following definitions:

- **TP** (True Positives) - Number of positive samples, which the model correctly classified as being positive;
- **TN** (True Negatives) - Number of negative samples, which the model correctly classified as being negative;

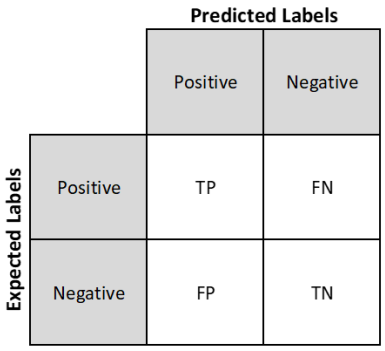


Figure 5.1: Confusion Matrix for a Binary Classification Task.

- **FP** (False Positives) - Number of negative samples, which the model wrongly classified as being positive;
- **FN** (False Negatives) - Number of positive samples, which the model wrongly classified as being negative.

From the confusion matrix, we can also compute the precision and recall metrics. The precision metric (5.2) represents the percentage of actual positive samples among all the samples predicted as being positive. This metric is useful to know how many samples are being incorrectly classified as positive. The recall metric (5.3) represents the percentage of positive samples correctly classified as such. This metric is useful to know the amount of samples that are being incorrectly classified as negative.

$$Precision = \frac{TP}{TP + FP} \tag{5.2}$$

$$Recall = \frac{TP}{TP + FN} \tag{5.3}$$

5.1.3 Accuracy on Multi-label Classification Tasks

When considering a multi-label classification task, there are some changes to the accuracy, precision and recall metrics. Observing the confusion matrix in Figure 5.2, we can find a value of TP from the perspective of each label.

The accuracy metric for a multi-label classification task can be calculated as:

$$Accuracy = \frac{\sum_{i=0}^{n} TP[label_i]}{\text{Sum of values in Confusion Matrix}} \tag{5.4}$$

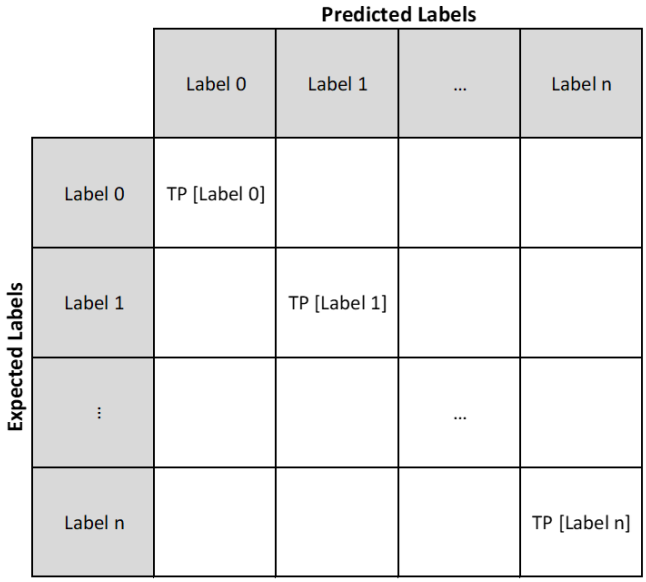


Figure 5.2: Confusion Matrix for a Multi-label Classification Task.

Both the precision and recall metrics will be computed per label. For a given label, we will only take into consideration the values present in the row or column correspondent to that label.

For the precision metric, we only consider the values present in the column of the label in focus. This metric is calculated as:

$$Precision[label_i] = \frac{TP[label_i]}{\text{Sum of } label_i \text{ column}} \tag{5.5}$$

On the other hand, for the recall metric we only consider the values present in the row of that label. This metric is calculated as:

$$Recall[label_i] = \frac{TP[label_i]}{\text{Sum of } label_i \text{ row}} \tag{5.6}$$

5.2 Training the Baseline CNNs

Considering the size of the Fruits 360 dataset, with 90,380 images, and the computational power required to process that amount of data, we resorted to batching when training our models. For each experiment, we will train the models using batch sizes of 32, 64, 128, 512, 1024 and 4096. Then, we will compare the effect of changing the batch size on the evaluation metrics.

Both baseline models were trained over 10 epochs and used Cross-Entropy as the loss function, as previously defined in Section 5.1.1. These models learned the normalized images: matrices with values between 0 and 1.

Analyzing Tables 5.1 and 5.2, we can notice that increasing the batch size negatively affected both

Table 5.1: Evaluation Metrics for the training of the "Baseline Alpha" Model.

Model	Process	Metric	Batch Size					
			32	64	128	512	1024	4096
Baseline Alpha	Train	Loss	0.0009	0.0018	0.0043	0.0417	0.1707	1.1597
		Accuracy	1.0000	1.0000	0.9999	0.9951	0.9772	0.7774
	Validation	Loss	0.2882	0.3372	0.3167	0.4380	0.5356	1.3191
		Accuracy	0.9346	0.9214	0.9225	0.8884	0.8521	0.6676

Table 5.2: Evaluation Metrics for the training of the "Baseline Beta" Model.

Model	Process	Metric	Batch Size					
			32	64	128	512	1024	4096
Baseline Beta	Train	Loss	0.2317	0.6640	1.2896	1.4712	2.3565	4.0354
		Accuracy	0.9215	0.7927	0.5945	0.6013	0.3970	0.1302
	Validation	Loss	0.2583	0.5623	0.7259	1.1381	1.9477	3.8857
		Accuracy	0.9334	0.8547	0.8162	0.7452	0.5730	0.1564

models. However, the "Baseline Beta" model seems to be affected the most, having its validation accuracy drastically decreased from 93.34% to 15.64%, between batch sizes 32 and 4096.

When comparing the results for the two models, the "Baseline Alpha" model has overall better results for both loss and accuracy metrics. In the best scenario (batch size 32) this model achieves an accuracy of 93.46% and a loss value of 0.2882, for the validation set. Even in the worst scenario (batch size 4096), this model still manages to achieve acceptable results, with accuracy of 77.74% and 66.76% for the train and validation sets, respectively, and a loss value slightly above 1 for both sets.

From these results, we can conclude that, even though both models were originally created to learn a subset of this dataset, the "Baseline Alpha" model managed to achieve great results when learning the full dataset. Comparing for the same batch size originally used (128), the "Baseline Alpha" model went from a train accuracy of 99.62%, for the subset of the Fruits 360 dataset (Section 4.1.3), to an accuracy of 92.25%, for the complete version of the dataset.

5.3 Learning the Extracted Features

For the experiments of learning the features extracted from the images, we will consider the same batch sizes defined on the training of the baseline CNNs. Similarly to the experiments performed for the baseline models, we will train models, this time over each independent modality instead of simply using the dataset images without any pre-processing, and evaluate the performance of the training process using the aforementioned metrics (Section 5.1).

All the models referred to in these experiments will follow the training configurations defined on the baseline experiments: train over 10 epochs with the Cross-Entropy Loss Function.

5.3.1 Color Modality

After training the Color Modality models, we present the resulting values for the accuracy and loss metrics, organized on the Tables 5.3 to 5.5.

The model "Color Gama" achieved acceptable results when trained with a batch size 32: accuracy values of 78.37% and 72.99%, for the train and validation sets, respectively, these were outliers when looking at the overall results.

Analyzing the metrics, we can conclude that none of the trained models achieved results that could be considered as good. Even if we consider the fact that increasing the batch size for the training process tends to negatively impact both loss and accuracy metrics, for the scenario with batch size 32 the models "Color Alpha" and "Color Beta" only achieved accuracy values of 41.48% and 45.04%, respectively, for the validation set.

Table 5.3: Evaluation Metrics for the training of the "Color Alpha" Model.

Model	Process	Metric	Batch Size					
			32	64	128	512	1024	4096
Color Alpha	Train	Loss	2.7048	3.7583	4.3628	4.6109	4.6666	4.7127
		Accuracy	0.4129	0.1901	0.0922	0.0388	0.0532	0.0213
	Validation	Loss	2.7577	3.7524	4.3617	4.6084	4.6648	4.7115
		Accuracy	0.4148	0.1965	0.0922	0.0385	0.0570	0.0202

Table 5.4: Evaluation Metrics for the training of the "Color Beta" Model.

Model	Process	Metric	Batch Size					
			32	64	128	512	1024	4096
Color Beta	Train	Loss	2.5478	3.6475	4.2750	4.5793	4.6508	4.7015
		Accuracy	0.4564	0.2275	0.1146	0.0389	0.0391	0.0458
	Validation	Loss	2.6234	3.6425	4.2777	4.5799	4.6504	4.7009
		Accuracy	0.4504	0.2337	0.1163	0.0388	0.0389	0.0548

Table 5.5: Evaluation Metrics for the training of the "Color Gama" Model.

Model	Process	Metric	Batch Size					
			32	64	128	512	1024	4096
Color Gama	Train	Loss	1.0156	2.4134	3.8948	4.5780	4.6507	4.7081
		Accuracy	0.7837	0.4524	0.1577	0.0389	0.0391	0.0624
	Validation	Loss	1.3011	2.4840	3.8628	4.5778	4.6488	4.7066
		Accuracy	0.7299	0.4388	0.1698	0.0385	0.0386	0.0650

Considering the low results, we started searching for what could be causing this problem. We noticed the fact that the background of these images is white could be having a bigger impact, than we initially thought, on the color histograms. We explore this problem and present a solution in the Section 4.2.2.A.

5.3.1.A Improving the Results

After the updating the color feature vectors, removing the pixels associated with the image background from the color histograms, we repeated the previous experiment. We present the results for the accuracy and loss metrics, for the new iteration of this experiment, organized on the Tables 5.6 to 5.8.

Table 5.6: Evaluation Metrics for the training of the "Color Alpha" Model (Improved Feature Vector).

Model	Process	Metric	Batch Size					
			32	64	128	512	1024	4096
Color Alpha	Train	Loss	0.2749	0.5547	1.2573	3.5924	4.1794	4.5603
		Accuracy	0.9554	0.8998	0.7747	0.2839	0.1535	0.0434
	Validation	Loss	0.4026	0.6497	1.2790	3.5480	4.1621	4.5516
		Accuracy	0.9181	0.8771	0.7659	0.3070	0.1605	0.0428

Analyzing the overall results for the accuracy and loss metrics, it is noticeable that removing the white background from the images proved to be a successful solution to the problem encountered in the previous experiment. All three models achieved loss values lower than 1, in both train and validation sets, for the scenarios with batch sizes 32 and 64. For the accuracy metric, we can observe that the models achieved values above 70% for the batch sizes 32, 64 and 128.

The "Color Gama" model continues to be have the best performance, among the models created for the Color Modality. When comparing this model's performance between the current and the previous iteration for this experiment, we can see a great improvement over the evaluation metrics. If we compare the scenario with batch size 32, we can see an increase of over 20% for the accuracy of both the training and validation sets. Additionally, the train loss went from 1.0156 (previous iteration) to 0.0882 (current iteration).

Table 5.7: Evaluation Metrics for the training of the "Color Beta" Model (Improved Feature Vector).

Model	Process	Metric	Batch Size					
			32	64	128	512	1024	4096
Color Beta	Train	Loss	0.2482	0.5059	1.0852	3.3213	4.0036	4.5215
		Accuracy	0.9619	0.9115	0.8222	0.3307	0.1708	0.0394
	Validation	Loss	0.3681	0.5989	1.1292	3.2913	3.9933	4.5161
		Accuracy	0.9260	0.8903	0.8041	0.3455	0.1798	0.0402

Table 5.8: Evaluation Metrics for the training of the "Color Gama" Model (Improved Feature Vector).

Model	Process	Metric	Batch Size					
			32	64	128	512	1024	4096
Color Gama	Train	Loss	0.0882	0.2064	0.5075	2.8768	4.0401	4.5718
		Accuracy	0.9837	0.9619	0.8979	0.4107	0.1903	0.0641
	Validation	Loss	0.2543	0.3372	0.6392	2.8479	4.0176	4.5664
		Accuracy	0.9423	0.9294	0.8642	0.4320	0.2014	0.0768

After this analysis, we decided to choose the model "Color Gama", trained with the background

removing mask and a batch size 32, to classify the test set and compute the respective Softmax probabilities, which then will be used as the Color Modality on the Multimodal Fusion.

5.3.2 Texture Modality

As mentioned in Section 4.2.3, the block size and bin step are configurable parameters on the creation of the Histogram of Oriented Gradients. For the training of the Texture Modality models, we chose 25x25 for the block size, which for a 100x100 image results in 16 blocks, and 10° for the bin step, which results in a total of 18 bins. With this configuration, the HOG feature vector will have a dimension of 648x1.

After training the Texture Modality models, we present the resulting values for the accuracy and loss metrics, organized on the Tables 5.9 to 5.11.

Table 5.9: Evaluation Metrics for the training of the "Texture Alpha" Model (block size 25x25, bin step 10).

Model	Process	Metric	Batch Size					
			32	64	128	512	1024	4096
Texture Alpha	Train	Loss	1.4513	2.6477	3.6474	4.4744	4.5773	4.6944
		Accuracy	0.7166	0.4061	0.1875	0.0475	0.0387	0.0433
	Validation	Loss	1.8544	2.8263	3.6953	4.4848	4.5830	4.6920
		Accuracy	0.5970	0.3521	0.1842	0.0473	0.0384	0.0494

Once more, it is clear the negative impact that increasing the batch size has on both metrics. For a batch size of 32, both "Texture Alpha" and "Texture Beta" models have average results. The former achieved train accuracy of 71.66%, which decreased to 59.7% for the validation set, and the latter achieved train accuracy of 74.20%, which, similarly, decrease to 61.98% for the validation set.

Table 5.10: Evaluation Metrics for the training of the "Texture Beta" Model (block size 25x25, bin step 10).

Model	Process	Metric	Batch Size					
			32	64	128	512	1024	4096
Texture Beta	Train	Loss	1.3635	2.5244	3.5713	4.4571	4.5643	4.6821
		Accuracy	0.7420	0.4543	0.2110	0.0405	0.0438	0.0378
	Validation	Loss	1.7920	2.7421	3.6240	4.4699	4.5713	4.6809
		Accuracy	0.6198	0.3883	0.1972	0.0413	0.0419	0.0389

Meanwhile, the "Texture Gama" model performed superiorly when compared with the other two Texture Modality models. Analyzing the loss metric, this model achieved a train loss lower than 1 (0.5357) and a validation loss slightly above 1 (1.0867), for the batch size 32. The accuracy metric had also good results for this batch size: 86.83% and 72.01% for the train and validation sets, respectively.

From the presented results for this experiment, we can conclude that the "Texture Gama" model is the clear choice to represent the Texture Modality. However, we propose a new experiment that intends to analyze if changing the HOG parameters would have a significantly impact on the evaluation metrics.

Table 5.11: Evaluation Metrics for the training of the "Texture Gama" Model (block size 25x25, bin step 10).

Model	Process	Metric	Batch Size					
			32	64	128	512	1024	4096
Texture Gama	Train	Loss	0.5357	1.4185	2.9870	4.4623	4.6074	4.6794
		Accuracy	0.8683	0.6687	0.2688	0.0472	0.0389	0.0312
	Validation	Loss	1.0867	1.8503	3.1084	4.4655	4.6078	4.6804
		Accuracy	0.7201	0.5628	0.2551	0.0480	0.0387	0.0399

5.3.2.A HOG Parametrization

Table 5.12: Evaluation Metrics for the training of the "Texture Gama" Model (block size 20x20, bin step 15).

Model	Process	Metric	Batch Size					
			32	64	128	512	1024	4096
Texture Gama	Train	Loss	0.3420	1.0340	2.3922	4.3280	4.5178	4.6801
		Accuracy	0.9244	0.7606	0.4447	0.0964	0.0393	0.0445
	Validation	Loss	0.9632	1.5638	2.6141	4.3387	4.5277	4.6797
		Accuracy	0.7564	0.6165	0.3807	0.0879	0.0383	0.0441

In the Table 5.12, we present the results of training the "Texture Gama" model with the following HOG configuration:

- Block size of 20x20, which for a 100x100 image results in 25 blocks;
- Bin step of $15°$, which results in a total of 12 bins;
- The resulting HOG feature vector will have a dimension of 768x1.

Table 5.13: Evaluation Metrics for the training of the "Texture Gama" Model (block size 20x20, bin step 20).

Model	Process	Metric	Batch Size					
			32	64	128	512	1024	4096
Texture Gama	Train	Loss	0.3645	1.0335	2.2355	4.3120	4.5279	4.6838
		Accuracy	0.9171	0.7556	0.4907	0.0813	0.0581	0.0314
	Validation	Loss	0.9986	1.5655	2.4487	4.3296	4.5394	4.6851
		Accuracy	0.7392	0.6183	0.4328	0.0776	0.0577	0.0324

In the Table 5.13, we present the results of training the "Texture Gama" model with the following HOG configuration:

- Block size of 20x20, which for a 100x100 image results in 25 blocks;
- Bin step of $20°$, which results in a total of 9 bins;
- The resulting HOG feature vector will have a dimension of 576x1.

The "Texture Gama" model achieved slightly better results in both the new HOG configurations, in comparison with the previous experiment. Focusing on the scenario with batch size 32, we notice

that for the new configurations, the model achieved a loss value lower than 1 for both the train and validation sets. In the previous experiment, this model achieved a validation accuracy of 72.01%. For the experiment with block size 20x20 and bin step 15° (Table 5.12), the validation accuracy (75.64%) increased over 3.5% in comparison to the first experiment. The experiment with block size 20x20 and bin step 20° (Table 5.13) also managed to increase over the initial validation accuracy, achieving 73.92%, which is an increase of almost 2%.

After this analysis, we decided to choose the model "Texture Gama", trained with block size 20x20, bin step 15° and a batch size 32, to classify the test set and compute the respective Softmax probabilities, which then will be used as the Texture Modality on the Multimodal Fusion.

5.3.3 Shape Modality

Table 5.14: Evaluation Metrics for the training of the "Shape Alpha" Model.

Model	Process	Metric	Batch Size					
			32	64	128	512	1024	4096
Shape Alpha	Train	Loss	0.0296	0.1022	0.2820	1.5757	2.7942	4.0913
		Accuracy	0.9970	0.9808	0.9368	0.6392	0.3858	0.0953
	Validation	Loss	1.9141	1.9193	1.9955	2.6814	3.4675	4.1597
		Accuracy	0.6800	0.6548	0.5958	0.4334	0.2863	0.0957

After training the Shape Modality models, we present the resulting values for the accuracy and loss metrics, organized on the Tables 5.14 to 5.16.

As expected, increasing the batch size also deteriorated the evaluation metrics for this experiment. Observing the results on Tables 5.14 to 5.16, we can see that for batch sizes greater than 128 will have a noticeable negative impact on the loss and accuracy metrics.

Table 5.15: Evaluation Metrics for the training of the "Shape Beta" Model.

Model	Process	Metric	Batch Size					
			32	64	128	512	1024	4096
Shape Beta	Train	Loss	0.0281	0.0862	0.2668	1.5654	2.6068	4.0452
		Accuracy	0.9971	0.9855	0.9411	0.6463	0.4325	0.0964
	Validation	Loss	1.8289	1.8788	1.9446	2.6346	3.1578	4.1286
		Accuracy	0.6882	0.6547	0.6081	0.4260	0.3045	0.0624

Table 5.16: Evaluation Metrics for the training of the "Shape Gama" Model.

Model	Process	Metric	Batch Size					
			32	64	128	512	1024	4096
Shape Gama	Train	Loss	0.0259	0.0839	0.2644	1.4344	2.6222	4.0579
		Accuracy	0.9975	0.9859	0.9420	0.6802	0.4303	0.0882
	Validation	Loss	1.8523	1.8059	1.9405	2.4947	3.2703	4.2330
		Accuracy	0.6887	0.6606	0.6002	0.4488	0.2767	0.0973

Overall, the three models had a similar performance for this experiment. If we focus on the metrics for the batch size 32, the models achieved almost the same validation loss value. The accuracy for the validation set, follows the same tendency, with all models achieving values between 68% and 69%.

Comparing the evaluation metrics, the "Shape Gama" model had the best performance for practically all batch sizes. For that reason, we decided to choose this model, trained with a batch size 32, to classify the test set and compute the respective Softmax probabilities, which the will be used as the Shape Modality on the Multimodal Fusion.

5.3.4 Testing the Modality Models

Having decided which model will represent each modality, we used these models to classify the test set. Testing the models serves for more than simply obtaining the test accuracy metric for these models. The main reason to process the test set is so the models generate the Softmax probabilities for the data samples.

When using the Softmax Activation Function, the models prediction consists of a probability vector. Each position of this vector represents the predicted probability of a sample being assigned to the corresponding label.

As previously defined, the models chosen to represent each modality are:

- **Color Modality** - "Color Gama" model, trained with a batch size of 32 and applying a background removing mask to the images before computing the color histograms;
- **Texture Modality** - "Texture Gama" model, trained with a batch size of 32 and with the HOG parameters: block size 20x20 and bin step 15°;
- **Shape Modality** - "Shape Gama" model, trained with a batch size of 32 and with the resized Shape Maps.

Table 5.17: Summary of the accuracy values for each Modality model.

Model	Accuracy		
	Train	Validation	Test
Color Modality	98.37%	94.23%	95.05%
Texture Modality	92.44%	75.64%	73.81%
Shape Modality	99.75%	68.87%	69.96%

The Table 5.17 summarizes the accuracy values that each Modality model achieve for the train, validation and test sets. The test accuracy values are in line with the validation accuracies. Both the Color and Shape Modality models had a slight increase from the validation accuracy, while the Texture Modality model had a decrease of less that 2% in for the test set.

5.4 Multimodal Fusion

Using the Dempster-Shafer's rule of combination on the Softmax prediction probabilities, produced from our modalities' classification of the test set, resulted in a combined accuracy value of 94.80%.

The Multimodal Fusion achieved a significantly better accuracy than both the Texture and Shape Modalities. In comparison with the Color Modality's test accuracy, the Multimodal Fusion achieved almost the same accuracy, with a small difference of 0.25%.

Analyzing from a label perspective, we extracted the following statistics:

- 61 labels, out of 113, have Precision = 100%;
- 1 label, out of 113, has Precision $<$ 60%;
- 66 labels, out of 113, have Recall = 100%;
- 2 labels, out of 113, have Recall $<$ 60%.

From these statistics we can conclude that more than 50% of the labels are being perfectly assigned, while only 3 out of the 113 labels present lower precision or recall values.

Although we achieved a good accuracy (94.80%) using the Dempster-Shafer's theory of evidence to combine our modalities, we decided to experiment using the same approach but only considering two out of the three modalities at a time.

Table 5.18: Summary of the Multimodal Fusion experiments.

Modalities			Accuracy
Color	Texture	Shape	
X	X	X	94.80%
X	X		97.40%
X		X	94.20%
	X	X	81.58%

The Table 5.18 summarizes the resulting accuracies obtained for each possible combination of the Color, Texture and Shape Modalities.

From these results, it is clear that, for the Fruits 360 dataset, the most significant modality is the Color, since the Multimodal Fusion module achieves the lowest accuracy when combining only the Texture and Shape Modalities. Consequently, the least relevant modality for this dataset seems to be the Shape.

The combination of the Color and Texture Modalities achieved an accuracy of 97.40%, showing an increase of slightly over 2.5% in comparison with the initial experiment of fusing all three modalities. Using the Color and Texture Modalities also improved the precision and recall metrics:

- 85 labels, out of 113, have Precision = 100%;
- All of the labels have Precision $>$ 60%;

- 85 labels, out of 113, have Recall = 100%;
- 2 labels, out of 113, have Recall $<$ 60%.

These statistics show that over 75% of the labels are being perfectly assigned, having both perfect precision and recall, which represent an improvement of 25% over the precision and classification results for the first experiment. This combination also improved the overall values for precision metric, since none of the labels achieved a precision lower than 60%.

Analyzing the results, we can conclude that for the Fruits 360 dataset the best approach would be to use the proposed classification architecture extracting only the color and texture information from the images.

5.5 Results Analysis

Finally, having concluded all the experiments and obtained the final values for the evaluation metrics, we compared the performance of our implementation with the two baseline models, whose train was explored in Section 5.2. In the Table 5.19, we present the accuracies for the baseline models and for the best modality combinations from the application of the Dempster-Shafer's combination rule. All the models being compared were trained in the same scenario: batch size 32 and 10 epochs.

Table 5.19: Comparison between the final results and the baseline models.

Model	Accuracy
Baseline Alpha	93.46%
Baseline Beta	93.34%
Dempster-Shafer [Color, Texture, Shape]	94.80%
Dempster-Shafer [Color, Texture]	97.40%

Observing Table 5.19, we can see that both baseline CNNs underperformed, in comparison with the results for the Multimodal Fusion. Both the baseline models and the implemented classification architecture were trained in the same circumstances, as previously mentioned, and used shallow Neural Networks (even the CNNs).

One aspect that is important to refer is that our implementation achieved more 4% accuracy using only two out of the three modalities extracted from the images. The Color and Texture Modalities combined have a dimensionality of 1,536 (768 from the color histograms and 768 from the HOG), while using the images represents a dimensionality of 30,000 (size 100x100 in a 3-channel color space).

Considering all this, we can conclude that in the same train circumstances, the implemented solution achieved better results, while resorting to smaller samples and simpler NNs than the baselines.

We decided to compare our results with another model trained with the same dataset (Fruits 360): Muresan-Oltean's model [Mureşan and Oltean, 2017].

Muresan and Oltean are the creators of the Fruits 360 dataset. They presented the dataset and consequently created a model to learn from those images. Their model consists of a 12-layers Convolutional Neural Network, with 4 pairs of Convolutional and MaxPooling layers. They trained this model for 25 epochs with a batch size of 50 samples, on 3 different scenarios, obtaining the following test accuracies for each:

- **Grayscale color space** - 95.25%;
- **RGB color space** - 98.66%;
- **HSV color space** - 96.09%.

Comparing the results from both our implementation and the Muresan-Oltean's model, its noticeable that our Multimodal Fusion classifier achieved a better accuracy than their Grayscale and HSV experiments. However, their RGB experiment achieved an accuracy 1.26% higher than our best experiment (Multimodal Fusion of the Color and Texture Modalities).

It seems that our solution slightly underperformed in comparison. Although, if we consider the complexity of Muresan-Oltean's model and the fact that they trained their model for 25 epochs, while all models from our solution were trained only for 10 epochs, we can claim that a difference of 1.26% is an acceptable trade-off for having both simpler models and data samples, two factors which have a great impact in reducing the computational power required to train Deep Learning models.

6

Conclusion

A new branch of Machine Learning models, designated as connectionist models, began with Rosenblatt's proposal of the Perceptron algorithm, which was inspired by the artificial neuron model previously presented by McCulloch and Pitts. Following important innovations, inspired by the Perceptron and its simulation of a neuron, models known as Artificial Neural Networks were created.

Some years later, Hubel and Wiesel discovered that the mammal brain has receptive fields inside its visual cortex. This was another important breakthrough, which launched models, such as Fukushima's Neocognitron, that envisioned the perception and processing of optical stimuli similarly to the human eye.

The notions presented by the those models and the introduction of the backpropagation algorithm, as the main learning approach used by neural networks, led to the creation of the first Convolutional Neural Network, proposed by LeCun.

Following several improvements over the first proposed model, CNNs have become one of the most widely used approaches to solve image recognition tasks.

Modern CNNs take advantage of many recently proposed innovations. Due to the ImageNet challenge, networks such as AlexNet, GoogLeNet and ResNet have proven that it is possible to have ever

deeper structures capable of training on enormous datasets of images and learning their most complex features autonomously.

Analyzing those models, we realized that modern CNNs have some shortcomings associated with their increasing depth. Some of the major problems found are the amount of labeled samples and computational power required by these networks to successfully learn the data. Additionally to these problems, recent studies have showed a certain degradation of the generalization ability of such networks, caused by the excessive increase of their depth.

As a solution to these shortcomings, we implemented a classification architecture with the goal of performing image recognition tasks using approaches that differ from the state-of-the-art deep CNNs. The main idea behind our solution was to extract the image's features and to have independent Neural Networks learning different feature types (modalities).

The implemented classification architecture consists of three sequential modules: Feature Extraction, Feature Learning and Multimodal Fusion.

On the Feature Extraction module, we used computer vision techniques to extract the features and organize them in feature vectors, which would represent each modality. The color information was extracted using color histograms, which were improved with the addition of a background removing mask. For the Texture Modality, we used the Sobel operator to compute the gradient of the images. From the gradient, we were able to create Histograms of Oriented Gradients. At last, for the shape information, we created our own approach, which consists of a weighted combination of: the silhouette of the object present in the image; the contours obtained with the Canny Edge Detector; and the corners extracted with both the Harris and Shi-Tomasi Corner Detectors.

Having extracted the features and created the feature vectors for each modality, the next step is learning the modalities on the Feature Learning module. Here, we created three Neural Networks for each modality and trained them using their respective feature vectors. At the end, we decided on a model to represent each modality, based on their performance with the validation set. We used the loss and accuracy metrics to evaluate the models performance.

With a trained model chosen to represent the Color, Texture and Shape modalities, we processed the test set with the Feature Extraction and Feature Learning modules, in order to obtain the Softmax probabilities for each test sample. These probability vectors were used as mass functions on the Dempster-Shafer's theory of evidence, on the Multimodal Fusion module. Resorting to the Dempster-Shafer's combination rule, we combined the Softmax probabilities produce by each Modality model and obtained the final labels for the test set.

To support the implementation of the proposed solution and experiment with the models from the Feature Learning module, we chose the Fruits 360 dataset. This dataset has a total of 90,380 images, assign to one of 113 labels. We used 75% of the dataset to train and validate the modules and 25% to

test the Multimodal Fusion module.

The implemented solution yielded good results, having achieved an accuracy of 94.80% for the fusion of all three modalities and 97.40% for the fusion of the Color and Texture modalities. Our Multimodal classifier performed better than both the baseline CNNs trained over the same dataset, in the same circumstances: 10 epochs with a batch size 32.

We compared our results with the model used by the creators of the Fruits 360 dataset. Their best experiment achieved an accuracy 1.26% higher than our top result. However, because our solution uses shallow NNs trained over 10 epochs with samples with considerable low dimensionality, instead of a 12-layer CNN trained over 25 epochs with 100x100x3 images, we conclude that the difference of accuracy is an acceptable trade-off.

From the implementation the proposed solution and the results obtained with the experiments, the main conclusion we retain is that sometimes it makes sense to analyze the modalities independently and even combining some of them, in order to obtain similar, or even better, results than using a deep CNN. Considering as an example the Fruits 360 dataset, we were able to achieve better results than the shallow baseline CNNs, and very close results to the 12-layer CNN used by Muresan and Oltean, combining only the information of color and texture extracted from the images, through an approach that is less demanding in terms of computational power.

6.1 Future Work

In terms of future work for our classification architecture, the following objectives stand out:

- In the field of feature extraction, we would like to experiment with other feature types, such as wavelets. Considering that this feature type is different in nature from the ones we are already using, it would be interesting to analyze the results of using the time-frequency information obtained from the wavelets;
- Obtain better hardware resources in order to experiment with training over more epochs to analyze if it would improve the already good results we achieved;
- Train our Multimodal classification model with bigger datasets, such as the ImageNet dataset, which is the dataset used in the ImageNet Large Scale Visual Recognition Challenge. This dataset was used by some of the models explored in Section 3.2: AlexNet, GoogLeNet and ResNet, and consists of 14 million images distributed over 20,000 labels. Similarly with the previous objective, this one also requires better hardware resources.

In general, we would like to increase the modalities our solution is able to extract and to experiment with larger datasets, that are currently being used for training the state-of-the-art CNNs.

Bibliography

[Barnard and Casasent, 1990] Barnard, E. and Casasent, D. (1990). Shift invariance and the neocognitron. *Neural Networks*, 3(4):403–410.

[Bishop, 2006] Bishop, C. M. (2006). *Pattern recognition and machine learning (Information Science and Statistics)*. Springer.

[Bishop et al., 1995] Bishop, C. M. et al. (1995). *Neural networks for pattern recognition*. Oxford university press.

[Canny, 1986] Canny, J. (1986). A computational approach to edge detection. *IEEE Transactions on pattern analysis and machine intelligence*, 8(6):679–698.

[Cardoso and Wichert, 2010] Cardoso, Â. and Wichert, A. (2010). Neocognitron and the map transformation cascade. *Neural Networks*, 23(1):74–88.

[Cardoso and Wichert, 2013] Cardoso, A. and Wichert, A. (2013). Handwritten digit recognition using biologically inspired features. *Neurocomputing*, 99:575–580.

[Cardoso and Wichert, 2014] Cardoso, Â. and Wichert, A. (2014). Noise tolerance in a neocognitron-like network. *Neural Networks*, 49:32–38.

[Comer and Delp III, 1999] Comer, M. L. and Delp III, E. J. (1999). Morphological operations for color image processing. *Journal of electronic imaging*, 8(3):279–289.

[Dalal and Triggs, 2005] Dalal, N. and Triggs, B. (2005). Histograms of oriented gradients for human detection. In *2005 IEEE computer society conference on computer vision and pattern recognition (CVPR'05)*, volume 1, pages 886–893. Ieee.

[Dandekar et al., 2021] Dandekar, M., Punn, N. S., Sonbhadra, S. K., Agarwal, S., and Kiran, R. U. (2021). Fruit classification using deep feature maps in the presence of deceptive similar classes. In *2021 International Joint Conference on Neural Networks (IJCNN)*, pages 1–6. IEEE.

[Debao, 1993] Debao, C. (1993). Degree of approximation by superpositions of a sigmoidal function. *Approximation Theory and its Applications*, 9(3):17–28.

[Deng et al., 2009] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee.

[Denker et al., 1989] Denker, J. S., Gardner, W., Graf, H. P., Henderson, D., Howard, R. E., Hubbard, W., Jackel, L. D., Baird, H. S., and Guyon, I. (1989). Neural network recognizer for hand-written zip code digits. In *Advances in neural information processing systems*, pages 323–331.

[Domingos, 2015] Domingos, P. (2015). *The master algorithm: How the quest for the ultimate learning machine will remake our world*. Basic Books.

[Fukushima, 1975] Fukushima, K. (1975). Cognitron: A self-organizing multilayered neural network. *Biological cybernetics*, 20(3-4):121–136.

[Fukushima, 1980] Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193–202.

[Fukushima, 1988] Fukushima, K. (1988). Neocognitron: A hierarchical neural network capable of visual pattern recognition. *Neural networks*, 1(2):119–130.

[Fukushima, 1989] Fukushima, K. (1989). Analysis of the process of visual pattern recognition by the neocognitron. *Neural Networks*, 2(6):413–420.

[Fukushima, 2003] Fukushima, K. (2003). Neocognitron for handwritten digit recognition. *Neurocomputing*, 51:161–180.

[Gonzalez et al., 2002] Gonzalez, R. C., Woods, R. E., et al. (2002). Digital image processing.

[Harris et al., 1988] Harris, C., Stephens, M., et al. (1988). A combined corner and edge detector. In *Alvey vision conference*, pages 10–5244. Citeseer.

[He et al., 2016a] He, K., Zhang, X., Ren, S., and Sun, J. (2016a). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

[He et al., 2016b] He, K., Zhang, X., Ren, S., and Sun, J. (2016b). Identity mappings in deep residual networks. In *European conference on computer vision*, pages 630–645. Springer.

[Hinton, 2012] Hinton, G. E. (2012). A practical guide to training restricted boltzmann machines. In *Neural networks: Tricks of the trade*, pages 599–619. Springer.

[Hubel, 1995] Hubel, D. H. (1995). *Eye, brain, and vision.* Scientific American Library/Scientific American Books.

[Hubel and Wiesel, 1962] Hubel, D. H. and Wiesel, T. N. (1962). Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology*, 160(1):106–154.

[Hubel and Wiesel, 1968] Hubel, D. H. and Wiesel, T. N. (1968). Receptive fields and functional architecture of monkey striate cortex. *The Journal of physiology*, 195(1):215–243.

[Kawaguchi, 2016] Kawaguchi, K. (2016). Deep learning without poor local minima. In *Advances in neural information processing systems*, pages 586–594.

[Krizhevsky, 2014] Krizhevsky, A. (2014). One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*.

[Krizhevsky et al., 2009] Krizhevsky, A., Hinton, G., et al. (2009). Learning multiple layers of features from tiny images. Technical report, Citeseer.

[Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.

[Kröse and van der Smagt, 1993] Kröse, B. and van der Smagt, P. (1993). An introduction to neural networks. *Journal of Computer Science (J Comput Sci)*.

[Laws, 1980] Laws, K. (1980). Textured image segmentation [dissertation]. *Los Angeles: University of Southern California*.

[LeCun, 1998] LeCun, Y. (1998). The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*.

[LeCun et al., 2015a] LeCun, Y., Bengio, Y., and Hinton, G. (2015a). Deep learning. *nature*, 521(7553):436–444.

[LeCun et al., 1989] LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551.

[LeCun et al., 1998] LeCun, Y., Bottou, L., Bengio, Y., Haffner, P., et al. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

[LeCun et al., 2015b] LeCun, Y. et al. (2015b). Lenet-5, convolutional neural networks. *URL: http://yann. lecun. com/exdb/lenet*, 20:5.

[Li and Wu, 1993] Li, C. and Wu, C.-H. J. (1993). Introducing rotation invariance into the neocognitron model for target recognition. *Pattern recognition letters*, 14(12):985–995.

[McCulloch and Pitts, 1943] McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133.

[Minsky and Papert, 1969] Minsky, M. and Papert, S. (1969). *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, MA, USA.

[Mureşan and Oltean, 2017] Mureşan, H. and Oltean, M. (2017). Fruit recognition from images using deep learning. *arXiv preprint arXiv:1712.00580*.

[Paulus and Hornegger, 2003] Paulus, D. and Hornegger, J. (2003). *Applied pattern recognition: algorithms and implementation in C++*. Springer Science & Business Media.

[Prewitt, 1970] Prewitt, J. M. (1970). Object enhancement and extraction. *Picture processing and Psychopictorics*, 10(1):15–19.

[Rosenblatt, 1958] Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386.

[Rumelhart et al., 1986] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *nature*, 323(6088):533–536.

[Satoh et al., 1999] Satoh, S., Kuroiwa, J., Aso, H., and Miyake, S. (1999). A rotation-invariant neocognitron. *Systems and computers in Japan*, 30(4):31–40.

[Schocken and Hummel, 1993] Schocken, S. and Hummel, R. A. (1993). On the use of the dempster shafer model in information indexing and retrieval applications. *International Journal of Man-Machine Studies*, 39(5):843–879.

[Serre, 2019] Serre, T. (2019). Deep learning: the good, the bad, and the ugly. *Annual Review of Vision Science*, 5:399–426.

[Serre et al., 2007] Serre, T., Wolf, L., Bileschi, S., Riesenhuber, M., and Poggio, T. (2007). Robust object recognition with cortex-like mechanisms. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, 29(3):411–426.

[Shafer, 1976] Shafer, G. (1976). *A mathematical theory of evidence*. Princeton university press.

[Shapiro, 2001] Shapiro, L. (2001). Computer vision/linda g. *Shapiro, George C. Stockman–Pearson*.

[Shi et al., 1994] Shi, J. et al. (1994). Good features to track. In *1994 Proceedings of IEEE conference on computer vision and pattern recognition*, pages 593–600. IEEE.

[Simard et al., 2003] Simard, P. Y., Steinkraus, D., Platt, J. C., et al. (2003). Best practices for convolutional neural networks applied to visual document analysis. In *Icdar*, volume 3.

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

[Sobel and Feldman, 1968] Sobel, I. and Feldman, G. (1968). A 3x3 isotropic gradient operator for image processing. *a talk at the Stanford Artificial Project in*, pages 271–272.

[Srivastava et al., 2014] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958.

[Sun and Su, 2015] Sun, W. and Su, F. (2015). Regularization of deep neural networks using a novel companion objective function. In *2015 IEEE International Conference on Image Processing (ICIP)*, pages 2865–2869. IEEE.

[Szegedy et al., 2015] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2015). Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9.

[Torralba et al., 2011] Torralba, A., Efros, A. A., et al. (2011). Unbiased look at dataset bias. In *CVPR*, volume 1, page 7. Citeseer.

[Veit et al., 2016] Veit, A., Wilber, M. J., and Belongie, S. (2016). Residual networks behave like ensembles of relatively shallow networks. In *Advances in neural information processing systems*, pages 550–558.

[Wandell and Thomas, 1997] Wandell, B. and Thomas, S. (1997). Foundations of vision. *Psyccritiques*, 42(7).

[Werbos, 1974] Werbos, P. (1974). Beyond regression:" new tools for prediction and analysis in the behavioral sciences. *Ph. D. dissertation, Harvard University*.

[Wichert, 2015] Wichert, A. M. (2015). *Intelligent Big Multimedia Databases*, chapter Chapter 5 - Feature Extraction. World Scientific.

A

Neural Network Structures

A.1 Baseline Models

The models represented in the Tables A.1 and A.2 were created with different input shapes from the ones used for this thesis. For the custom subset of the Fruits 360 dataset, the “Baseline Alpha” and the “Baseline Beta” had input shapes of (25, 25, 3) and (20, 20, 3), respectively.

Table A.1: Structure of the “Baseline Alpha” Model.

Layer Type	Configuration	Activation Function
Input	Input Shape = (100, 100, 3)	-
Convolutional	Filters = 32 Kernel Size = (5, 5) Strides = (1, 1) Padding = None	ReLU
Output	Units = 113	Softmax

Table A.2: Structure of the “Baseline Beta” Model.

Layer Type	Configuration	Activation Function
Input	Input Shape = (100, 100, 3)	-
Convolutional	Filters = 32 Kernel Size = (3, 3) Strides = (1, 1) Padding = None	ReLU
Pooling	Downsampling = MaxPooling Kernel Size = (2, 2)	-
Dropout	p = 0.25	-
Fully-Connected	Units = 128	ReLU
Dropout	p = 0.5	-
Output	Units = 113	Softmax

A.2 Feature Learning Models

The models represented in the Tables A.3, A.4 and A.5, were used on the experimental evaluations of the color modality.

The models represented in the Tables A.6, A.7 and A.8, were used on the experimental evaluations of the texture modality. The input shape use for these models will depend on the parameters used to compute the Histograms of Oriented Gradients.

The models represented in the Tables A.9, A.10 and A.11, were used on the experimental evaluations of the shape modality.

Table A.3: Structure of the “Color Alpha” Model.

Layer Type	Configuration	Activation Function
Input	Input Shape = (768,1)	-
Fully-Connected	Units = 128	ReLU
Output	Units = 113	Softmax

Table A.4: Structure of the “Color Beta” Model.

Layer Type	Configuration	Activation Function
Input	Input Shape = (768,1)	-
Fully-Connected	Units = 256	ReLU
Output	Units = 113	Softmax

Table A.5: Structure of the “Color Gama” Model.

Layer Type	Configuration	Activation Function
Input	Input Shape = (768,1)	-
Fully-Connected	Units = 384	ReLU
Fully-Connected	Units = 192	ReLU
Output	Units = 113	Softmax

Table A.6: Structure of the “Texture Alpha” Model.

Layer Type	Configuration	Activation Function
Input	Input Shape = (768,1) or (648,1)	-
Fully-Connected	Units = 256	ReLU
Output	Units = 113	Softmax

Table A.7: Structure of the “Texture Beta” Model.

Layer Type	Configuration	Activation Function
Input	Input Shape = (768,1) or (648,1)	-
Fully-Connected	Units = 384	ReLU
Output	Units = 113	Softmax

Table A.8: Structure of the “Texture Gama” Model.

Layer Type	Configuration	Activation Function
Input	Input Shape = (768,1) or (648,1)	-
Fully-Connected	Units = 384	ReLU
Fully-Connected	Units = 192	ReLU
Output	Units = 113	Softmax

Table A.9: Structure of the “Shape Alpha” Model.

Layer Type	Configuration	Activation Function
Input	Input Shape = (50, 50, 1)	-
Convolutional	Filters = 8 Kernel Size = (10, 10) Strides = (1, 1) Padding = None	ReLU
Output	Units = 113	Softmax

Table A.10: Structure of the “Shape Beta” Model.

Layer Type	Configuration	Activation Function
Input	Input Shape = (50, 50, 1)	-
Convolutional	Filters = 16 Kernel Size = (10, 10) Strides = (1, 1) Padding = None	ReLU
Output	Units = 113	Softmax

Table A.11: Structure of the “Shape Gama” Model.

Layer Type	Configuration	Activation Function
Input	Input Shape = (50, 50, 1)	-
Convolutional	Filters = 32 Kernel Size = (10, 10) Strides = (1, 1) Padding = None	ReLU
Output	Units = 113	Softmax