UNIVERSITY OF PIRAEUS

DEPARTMENT OF DIGITAL SYSTEMS

POSTGRADUATE PROGRAMME

“INFORMATION SYSTEMS & SERVICES”

BIG DATA & ANALYTICS

Thesis Title: Data Analytics Algorithms for Multi-Dimensional Datasets

AN IMAGE CLASSIFIER TO RECOGNIZE DIFFERENT SPECIES OF FLOWERS

EMMANOUIL ALEXAKIS

ΕΜΜΑΝΟΥΗΛ ΑΛΕΞΑΚΗΣ

ME 1702

ACADEMIC SUPERVISOR: ASSISTANT PROFESSOR DIMOSTHENIS KYRIAZIS

PIRAEUS 2019

Ο επιβλέπων	**1ος συνεξεταστής**	**2ος συνεξεταστής**
Δ. Κυριαζής Επ. Καθηγητής	Μ. Φιλιππάκης Αν. Καθηγητής	Δρ. Α. Μενύχτας Διδάσκων

Για την Γεωργία και τον Αχιλλέα…

Preface

Hundreds of flowers exist on earth, consisting an integral part of all livings not only for the aesthetic aspect but also for human life in many areas such as medical science, industry and environment. It is necessary to set up a database for flower documentation by determining an effective mean to identify the species to which they belong even from a smartphone application. As Artificial Intelligence algorithms (Neural Networks) are more and more incorporated into everyday applications, developing such an image classifier by creating a deep learning model trained on hundreds of thousands of images would drive as part of the overall application architecture. In this work, there is developed and trained an image classifier for recognizing 102 distinct species of flowers utilizing a certain type of machine learning algorithm called Convolutional Neural Networks, resulting in very good performance on a variety of experiments, achieving up to 94% accuracy.

Keywords: Algorithms, Artificial Intelligence, Machine Learning, Deep Learning

Πρόλογος

Εκατοντάδες λουλούδια υπάρχουν στη γη, αποτελώντας αναπόσπαστο μέρος όλων των έμβιων όντων όχι μόνο από αισθητική άποψη αλλά και για την ανθρώπινη ζωή σε πολλούς τομείς όπως η ιατρική επιστήμη, η βιομηχανία και το περιβάλλον. Είναι απαραίτητο να δημιουργηθεί μια βάση δεδομένων για την τεκμηρίωση των λουλουδιών, προσδιορίζοντας έναν αποτελεσματικό τρόπο αναγνώρισης των ειδών στα οποία ανήκουν, ακόμη και από μια εφαρμογή έξυπνου τηλεφώνου. Επειδή οι αλγόριθμοι της Τεχνητής Νοημοσύνης (Νευρωνικά Δίκτυα) ενσωματώνονται ολοένα και περισσότερο στις καθημερινές εφαρμογές, η ανάπτυξη ενός τέτοιου ταξινομητή εικόνας δημιουργώντας ένα μοντέλο βαθιάς μάθησης εκπαιδευμένο σε εκατοντάδες χιλιάδες εικόνες θα οδηγούσε ως μέρος της συνολικής αρχιτεκτονικής εφαρμογών. Στη παρούσα εργασία, αναπτύχθηκε και εκπαιδεύτηκε ένας ταξινομητής εικόνας για την αναγνώριση 102 διακριτών ειδών λουλουδιών χρησιμοποιώντας μιας συγκεκριμένης κατηγορίας αλγορίθμων μηχανικής μάθησης που ονομάζονται Convolutional Neural Networks, με αποτέλεσμα πολύ καλές επιδόσεις σε μια ποικιλία πειραμάτων, επιτυγχάνοντας ακρίβεια έως και 94%.

Λέξεις Κλειδιά: Αλγόριθμοι, Τεχνητή Νοημοσύνη, Μηχανική Μάθηση, Βαθιά Μάθηση

Table of Contents

Ευχαριστίες .... **Error! Bookmark not defined.**

Αφιέρωση .... **Error! Bookmark not defined.**

Preface vi

Πρόλογος vii

Introduction 1

1 From Data Science to Artificial Intelligence 3

1.1 Knowledge Discovery in Data 4

1.2 Machine Learning Algorithms 5

1.2.1 Types of Machine Learning Algorithms 6

1.2.2 Neural Networks and Deep Learning 8

2 Data and Methodology 13

2.1 Data Provenance and Description 13

2.2 Define the Neural Network Architecture 14

2.3 Methodological Approach and Tools 18

3 Results and Discussion 25

3.1 Loading and Preprocessing the image dataset 25

3.2 Selection between Pretrained and Untrained Model 27

3.3 Building the Image Classifier 31

3.4 Model Fine Tuning and Feature Extraction 31

3.5 Build, Train and Validate the Model 36

3.6 Saving and Loading the Model 40

3.7 Inference for Classification and Input Image Preprocessing 40

3.8 Flower Class Prediction 42

4 Conclusions 47

References 49

Introduction

Hundreds of flowers exist on earth, consisting an integral part of all livings not only for the aesthetic aspect but also for human life in many areas such as medical science, industry and environment. It is necessary to set up a database for flower documentation by determining an effective mean to identify the species to which they belong. To do so, the morphologic features that make each flower distinct and differ from the others has to be determined. Generally, morphological feature analysis is not a simple task for non-expert users, as it needs accurate observations of the flower itself and comparisons with other samples of the same species [1]. For this reason, the need to develop an image classifier able to classify flowers and automatically determine the species they belong to, is arising.

Machine Learning, Artificial Intelligence and Deep Learning could assist to the endeavor of developing such a classifier. Artificial Neural Networks (ANN), and more particular Convolution Neural Networks have led to superior performance on a variety of classification problems such as visual and speech recognition. Leveraging on the rapid growth in the amount of the annotated data and the great improvements in the performance of graphics processor units, the research on CNNs has achieved state-of-the-art results on various classification tasks [2]. The purpose of this study is to highlight the importance of machine learning algorithms for processing multidimensional data and provide a guide in order to decide which type of algorithm is optimum to use within the analysis. In particular, in this study, there is developed and trained an image classifier for recognizing 102 distinct species of flowers utilizing a certain type of machine learning algorithm called Convolutional Neural Networks, resulting in very good performance on a variety of experiments, achieving up to 94% accuracy.

1 From Data Science to Artificial Intelligence

On the one hand, Data Science uses computer disciplines such as mathematics and statistics and incorporates techniques such as data mining, cluster analysis, visualization and Machine learning, whereas, on the other hand Artificial Intelligence (AI) is simply a computer capable of imitating or simulating human thought or behavior. Inside that set, there is a subset called machine learning that is now the foundation of the most exciting part of AI. By enabling computers to learn themselves how to solve problems, machine learning has led to a series of breakthroughs that once seemed almost impossible. Machine learning is a branch of AI where a class of data-driven algorithms allows software applications to become very accurate in predicting results without the need for explicit programming. Its fundamental principle is based on the development of algorithms that receive input data and use statistical models to predict outputs while updating them when new data becomes available. The processes involved have a lot in common with predictive modeling and data mining. This is because both approaches require looking in the data to identify patterns and adjust the program accordingly. To this extend, machine learning is a subset of artificial intelligence and data science is an interdisciplinary field for extracting knowledge from data.

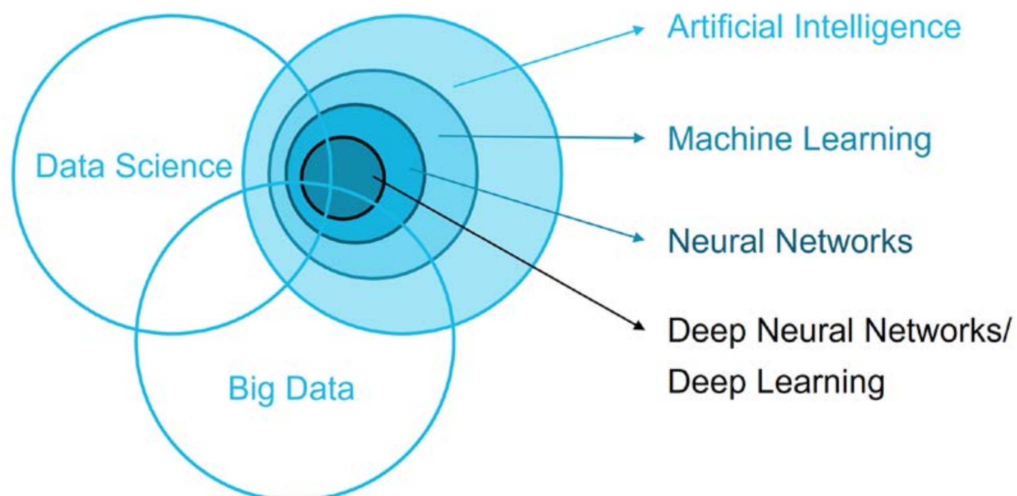

Figure 1 VENN diagram of AI, Big Data and Data Science - Fraunhofer FOKUS

1.1 Knowledge Discovery in Data

The purpose of this chapter is to introduce data mining and the particular data mining techniques to be able to extract the hidden information or relationships within the available datasets. The contents of this chapter will concern an introduction of what data mining is and what purpose generally and particularly in this study fulfills. Then, a thorough description of the calculation steps behind the specific data mining techniques utilized in this study will follow. Within this description the need of data preprocessing will arise making a connection to the coming chapter where the description and the preprocessing of the data takes place. Knowledge Discovery in Databases (KDD) is the computational process of discovering patterns in large data sets involving methods at the intersection of artificial intelligence, machine learning, statistics and database systems [3, 4]. The KDD process is commonly defined within the following stages:

a. Data selection
b. Data pre-processing and transformation
c. Data Mining
d. Validation

a) Before implementing the data mining algorithms, a target dataset is selected. As data mining uncovers patterns present in the data, the target dataset is supposed to be large enough in order to contain these patterns. Additionally, the target dataset has to remain concise in order to be mined within an acceptable time limit. b) Pre-processing is essential to analyze the multivariate datasets before data mining while in case there is a large number of variants a certain transformation of the dataset can take place by classifying them. If necessary, the target dataset is then cleaned by removing the observations containing noise and those with missing data (NaN). The cleaning step should be done with care as even missing data might indicate hidden patterns. c) Data Mining is the advanced step of KDD with the goal to extract information from a data set and transform it into an understandable structure. Data mining involves six classes [5]:

1) Anomaly detection: Identification of unusual data records or data errors
2) Association rule learning: Investigate relationships among variables.
3) Clustering: Discover new groups and structures in the data.
4) Classification: Generalize known structure to apply to new data.
5) Regression: Attempt to find a function which models the data with the least error.

6) Summarization: Provide a more compact representation of the dataset, including visualization and report generation.

d) The final step of KDD process is to identify whether the patterns found by the data mining algorithms are valid or not. The evaluation takes place on a test subset that data mining algorithm was not trained. The learned patterns are applied on the test subset and the resulting output is compared to the desired output. A number of statistical methods may be used to evaluate the algorithm, such as Receiver Operating Characteristic (ROC) curves. In case the learned patterns do not meet the desired standards, re-evaluation and changing the pre-processing and data mining steps is necessary. On the other hand, if the learned patterns do meet the desired standards, the final step is to interpret the learned patterns.

1.2 Machine Learning Algorithms

To realize the aforementioned, Data Scientists and Analysts are concerned in locating and implementing machine learning algorithms in order to address the issues they are interested in, but they are early come up with a certain question of “which algorithm they must use?". The answer to the question depending on many factors, such as: a) The data nature (size, quality etc.) b) The available time and last but not least c) What do they want to do with the data. When it comes to choose an algorithm, parameters like accuracy, training time and ease of use have always to always to be taken into account.

When a data set is available, the first thing to consider is how to get results, no matter what those results are. Once first results have delivered and user have to familiarize with the data by spending more time using more sophisticated algorithms to enhance their understanding of data so as to improve the results. Even an experienced data scientist cannot judge in advance which algorithm will have the optimum performance before they really test the different algorithms. The Machine Learning Algorithm Cheat Sheet presented in Figure 2 is working as a driver/guide in order to find which algorithm suits for specific problems. The propsed algorithms arise from the collection of feedbacks and advice from various data scientists and dedicated learning engineers and developers. Further explanation and how the cheat sheet is utilized is following.

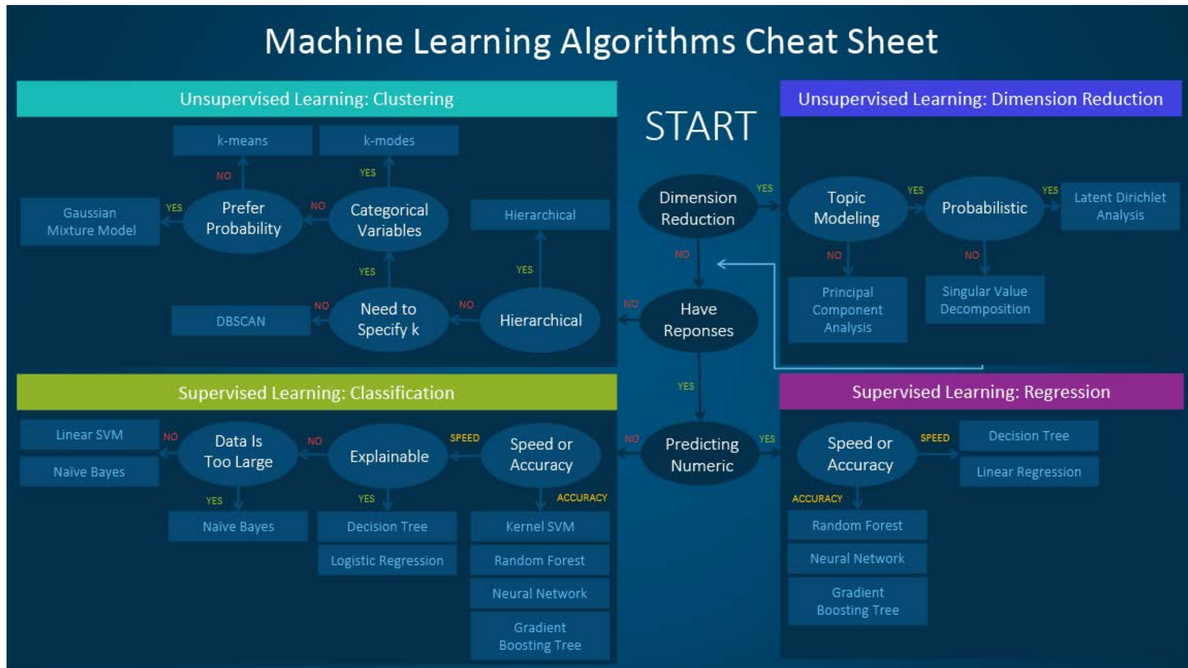

Figure 2 A simplified guide to navigate through the process of choosing what is the best machine learning algorithm [6]

If someone wants to reduce the dimensions of their data they should use Principal Component Analysis, if they need a numerical prediction, use decision trees or linear regression, If they need an hierarchical result, they should use hierarchical clustering, according to Figure 2. As the suggested cheat sheets routes are intended to be rule-by-thumb recommendations, many the times there will be more than one branch algorithm applied and other times none of them will be perfect match, thus, the only sure way to find the best algorithm is to test all of those algorithms.

1.2.1 Types of Machine Learning Algorithms

The most popular division of machine learning algorithms types is: Supervised, Semi-Supervised, Unsupervised and Reinforcement Learning algorithms [7]. Supervised learning algorithms make forecasts based on a set of known examples. For instance, meteorological data can be used to estimate tomorrow's average temperature or wind speed. Within supervised learning, there is available an input variable consisting of labeled training data and the desirable output variable. Then an algorithm can utilize training

data in order to learn the function that maps the input variable to the output one. This function implies mapping of new, unknown examples, generalizing from the training data, to predict the results in unseen situations. When the data is used to predict a categorical variable (i.e. type of flower), supervised learning is called classification. This is the case when assigning a label, either rose or margarita to an image and when there are only two labels, the problem is called binary classification while when there are more than two categories, is called multi-class classification. When it comes to predict constant values, they become a regression problem and when it comes to predict the future based on the past and present data it is called forecasting.

The difficulty regarding the supervised learning is that labeling data can be time-consuming and extremely expensive. In case that labels are limited, the use of non-labeled examples could enhance supervised learning. As the machine learning process is not fully monitored, this particular case of learning is called Semi-Supervised. In Unsupervised learning, the machine learning process is taking place with completely unlabeled data. Usually, the intrinsic patterns hidden within the data, such as a clustering structure, a low-dimensional etc., is expected to be discovered. Grouping a set of data examples so that the examples in a group are more similar (according to some criteria) than in other groups, is called clustering and is included in the Unsupervised machine learning process. This is often used to divide the complete dataset into diverse groups so as to perform analysis in each group to help users find inherent patterns. Diminishing the number of variables under consideration when primary data has very high dimensional characteristics and some features are unnecessary or unrelated to work. Dimensionality Reduction helps to discover the true, latent relationship and it is also included in the Unsupervised machine learning process.

Finally, Reinforcement learning accounts the behavior of an agent based on environmental feedback. In particular, through the process of Trial and Error, rather than telling machines what action they need to take they instead try different scenarios to find out what the most rewarding actions are. The test-and-error and the reward approach distinguish the reinforcement learning from other techniques.

1.2.2 Neural Networks and Deep Learning

1.2.2.1 Historical Background

Neural network simulations appear to be a recent development. The field of Neural Network modeling has been established even before the advent of computers and has survived through several eras. The idea of neurons as structural components of the brain was firstly introduced in the work of Ramón y Cajál, back in 1911, who was struggling to understand how the brain incurs [8]. In fact, the first artificial neural network was developed in 1943, by Warren and Pitts, but the available means of that time were not enough to take any advantage of the them [9]. Despite the fact that their Neural Network had a a fixed set of weights, they suggested innovative ideas like that a neuron has a threshold level which is still remain within the fundamental core of how ANNs operate. In 1949, Hebb created the first learning rule; if two neurons are active at the same time then the strength of their bond should be increased [10]. In the decades to come many researchers (Rosenblatt, Minsky, Papert etc.) dealt with the concept of perceptron [11], [12]. They created the algorithm for pattern recognition [11] but they were also discovered that the basic perceptrons were incapable of processing the exclusive-or circuit meaning that perceptron could not learn those functions which are not linearly separable and the research in the field declined throughout the mid 70's. It was then that Werbos introduced the backpropagation algorithm that made the training of multi-layer networks feasible and efficient by distributing the error term back up through the layers, modifying that way the weights at each node [13].

1.2.2.2 Convolution Neural Network - State of the Art

Convolution Neural Networks (CNN) is the state-of-the-art approach to object recognition and has shown greatly advance on the performance of many compute vision tasks like object recognition and tracking, text recognition or/and detection, pose estimation, action recognition etc. CNNs look like to normal neural networks to the level that both can be arranged as an acyclic graph and visualized as a collection of neurons. Their main difference is that in CNN a hidden layer neuron could be connected only to a subset of neurons of the previous layer thus, they are capable of learning features in an implicit manner. Their architecture can result in hierarchical feature extraction, for example in the 1st convolutional layer, the trained filters can be visualized as set of edges, in the second layer as some

shapes, in the coming layer filters might learn object parts whereas the filters of the final layers can identify the objects (Figure 3).

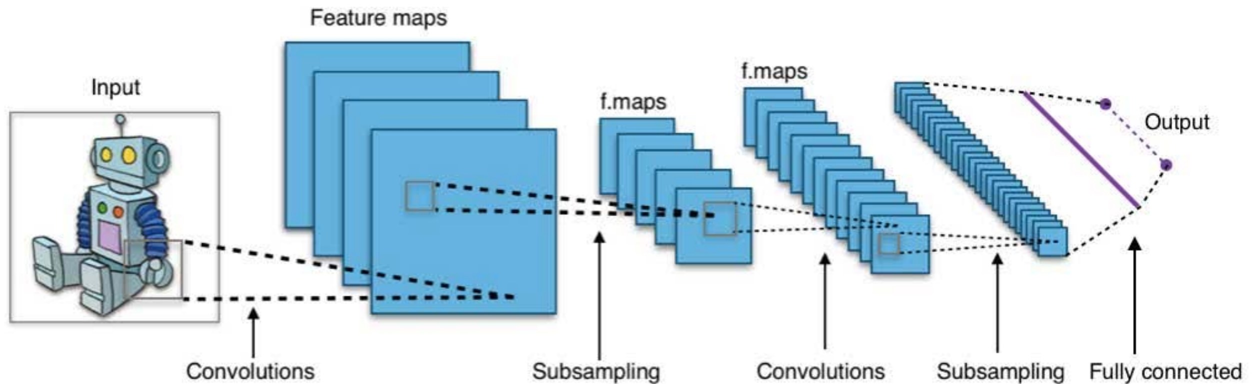

Figure 3 A convolution neural network architecture [14]

The predecessor of CNNs is the so called neocognitron developed in 1980 [15] but LeNet, developed by LeCun et al. in 1990, was in fact the innovate work [16] that highlighted the full potentials of these kind of algorithms as it successfully managed to recognize and classify handwritten digits directly from the input image without any preprocessing. Since then, a wide range of techniques have been developed to enhance the performance or ease the training of CNNs. Some of the most known techniques that have been implemented are published by Krizhevsky and Goodfellow [17], [18]. The network that Krizhevsky (Figure 4) [17] constructed, has eight learned layers – five of them convolutional and three fully-connected. The output of the last layer is a 1000-way softmax which produces a distribution over the 1000 class labels (the ImageNet challenge is to create a classifier that can determine which object is in the image). Because the network is too large to fit in the memory of one GPU, training is split across two GPUs and the kernels of the 2nd, 4th, and 5th convolutional layers are connected only to those kernel maps in the previous layer which reside on the same GPU.

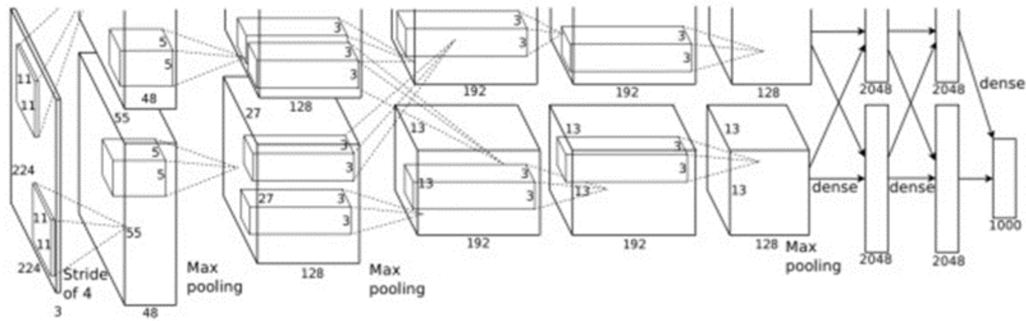

Figure 4 The architecture of CNN, describing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers.

The authors single out four other aspects of the model architecture that they feel are particularly important:

- The use of ReLU activation (instead of tanh). "Deep convolutional neural networks with ReLUs train several times faster than their equivalents with tanh units. Faster learning has a great influence on the performance of large models trained on large datasets ".
- Using multiple GPUs (two) and splitting the kernels between them with cross-GPU communication only in certain layers. The scheme reduces the top-1 and top-5 error rates by 1.7% and 1.2% respectively compared to a net with half as many kernels in each layer and trained on just one GPU.
- Using local response normalization, which "implements a form of lateral inhibition inspired by the type found in real neurons, creating competition for big activities amongst neuron outputs computed using different kernels".
- Using overlapping pooling. Let pooling layers be of size z x z, and spaced s pixels apart. Traditionally pooling was used with s = z, so that there was no overlap between pools. Krizhevsky et al. used s = 2 and z = 3 to give overlapping pooling. This reduced the top-1 and top-5 error rates by 0.4% and 0.3% respectively.

To reduce overfitting dropout and data augmentation (translations, reflections, and principal component manipulation) is used during training.

This convolutional neural network is capable of achieving record-breaking results on a highly challenging dataset using purely supervised learning. It is notable that network's performance degrades if a single convolutional layer is removed. For example, removing any of the middle layers results in a loss of about 2% for the top-1 performance of the network.

Another renowned CNNs are the Maxout Networks [18]. These are designed to work hand-in-glove with dropout. Training with dropout is like training an exponential number of models all sharing the same parameters. In other words, Maxout Networks are standard multilayer perceptron or deep CNNs that use a special activation function called the maxout unit. The output of a maxout unit is simply the maximum of its inputs. Maxout units make a piecewise linear approximation to an arbitrary convex function, as illustrated in Figure 5.

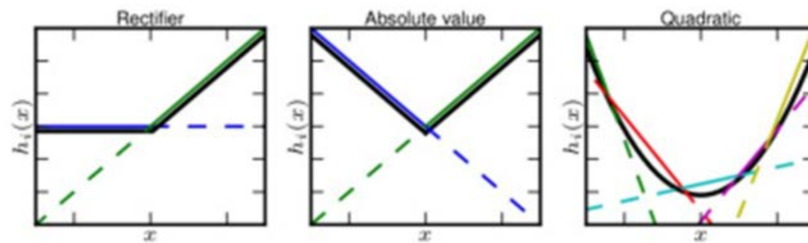

Figure 5 Graphical depiction of how the maxout activation function can implement the rectified linear, absolute value rectifier, and approximate the quadratic activation function.

Under evaluation, the combination of maxout and dropout achieved state of the art classification performance on MNIST, CIFAR-10, CIFAR-100, and SVHN (Street View House Numbers). Dropout does exact model averaging in deeper architectures provided that they are locally linear among the space of inputs to each layer that are visited by applying different dropout masks. In addition, rectifier units that saturate at zero are much more common with dropout training. A zero value stops the gradient from flowing through the unit making it hard to change under training and become active again.

2 Data and Methodology

2.1 Data Provenance and Description

Maria-Elena Nilsback and Andrew Zisserman have created a 102-category dataset, consisting of 102 distinct flower classes [19]. The flowers chosen to be flower commonly occurring in the United Kingdom and each class consists of between 40 and 258 images. The images have large scale, pose and light variations. In addition, there are categories that have large variations within the category and several very similar categories. The dataset is visualized using isomap with shape (Figure 6) and colour features (Figure 7). The images are randomly sampled from the category. The details of the categories and the exact number of images for each class can be found on this statistics page [20]. From the available data non site [19] "*Dataset images*" and "*The image labels*" were used to train the image classifier of the present work.

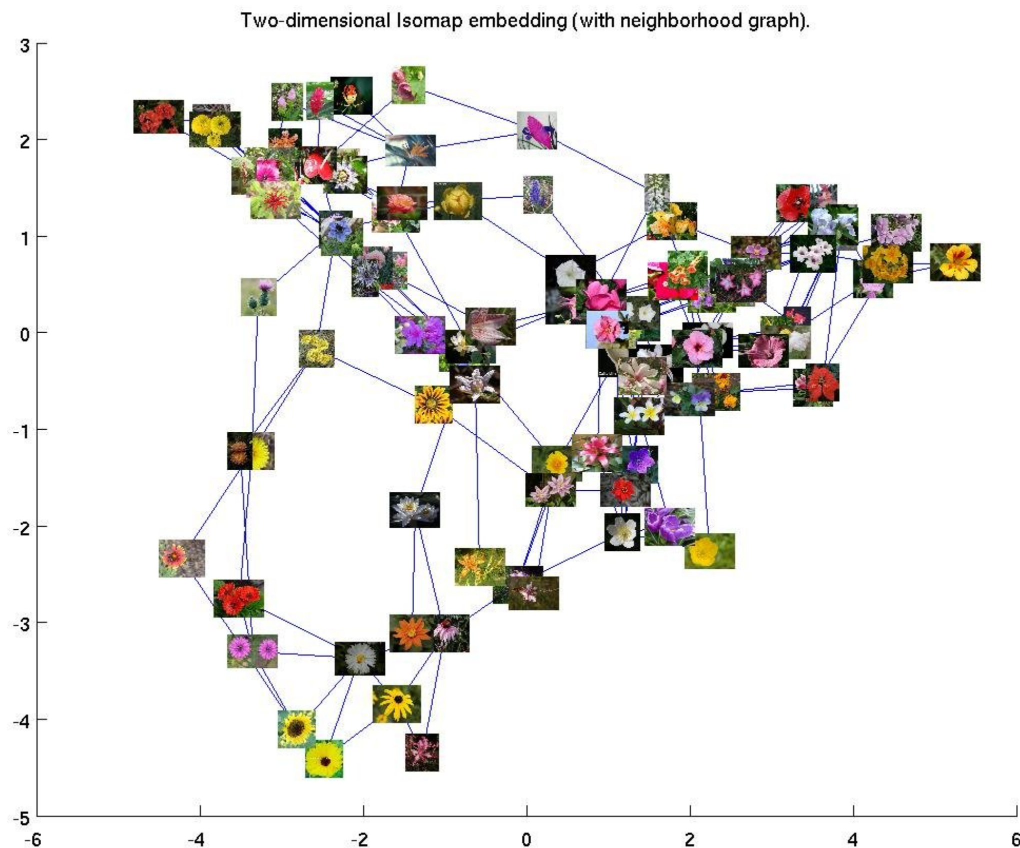

Figure 6 The categories in the dataset using SIFT features as shape descriptors [19]

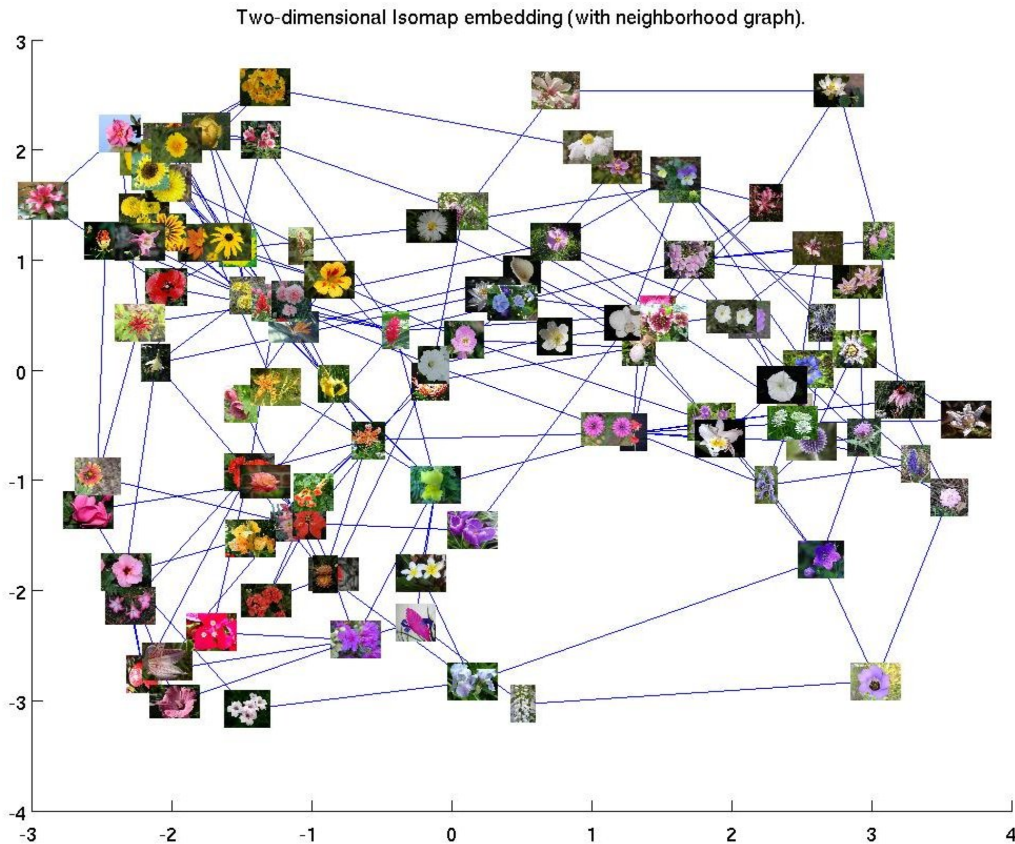

Figure 7 The categories in the dataset using HSV as color descriptors [19]

2.2 Define the Neural Network Architecture

The image classification problem dealt with in this study, is to determine in which of the 102 classes a given image belongs. Now that the data is ready, it's time to build and train the classifier. The VGG19 pretrained Convolution Neural Network from torchvision models is utilized in order to get the image features and a new feed-forward classifier is built and trained using those features. Convolutional Neural Networks is special type of Neural Networks working in the same way as a common neural network, but it includes a convolution layer at its beginning. Thus, rather than feeding the model, as an array of numbers, the entire image, the image is segmented into several squared pieces (m x m pixels) and the model tries to predict what each of these pieces is. Eventually, the model predicts the content of the picture based upon the prediction of all the pieces. In this way, the operations are parallelized, and the detection of the object takes place regardless of its location in the image. The method followed to build the image classifier involves the following steps:

1. Load the VGG-19 pre-trained Neural Network
2. Define a new, untrained feed-forward network as a classifier, using ReLU activations and dropout
3. Train the classifier layers by backpropagation using the pre-trained network to get the features
4. Track the loss and accuracy on the validation set to determine the best hyperparameters (learning rate, units in the classifier, epochs, etc.).
5. Determine the best model and save it with those hyperparameters for inference if model is needed to be rebuilt.

In this study, VGG-19 Convolution Neural Network will be utilized and as it will be presented in the results chapter, the accuracy regarding the image prediction is of the same magnitude whether the model is trained from the scratch or the pretrained one is used in advance. In order to build a single layer Convolutional Neural Network (CNN) there are 5 steps involved: a) Define the Convolution Layer (Initialize a single convolutional layer so that it contains all your created filters) b) Pass the result from an activation function c) Feed output from previous step in a Pooling Layer d) Flatten the vector and e) Feed it to a Fully Connected Linear Layer (Figure 8). Steps from a to c can be repeated as many times as needed in order to achieve the desire depth of the Neural Network.

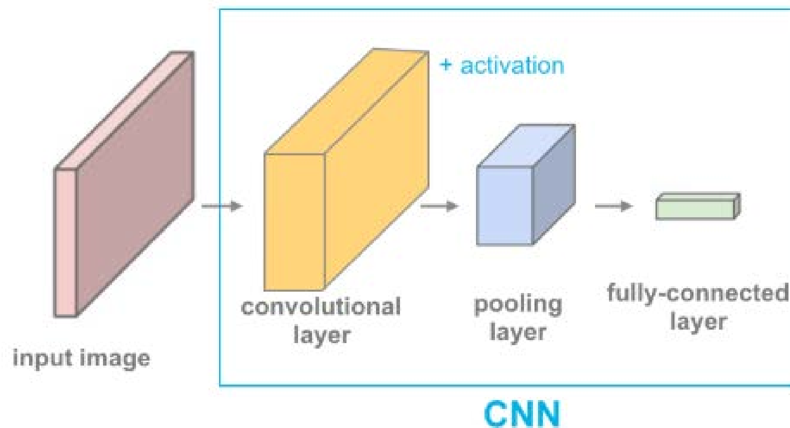

Figure 8 Schematic Representation of a Convolution Neural Network

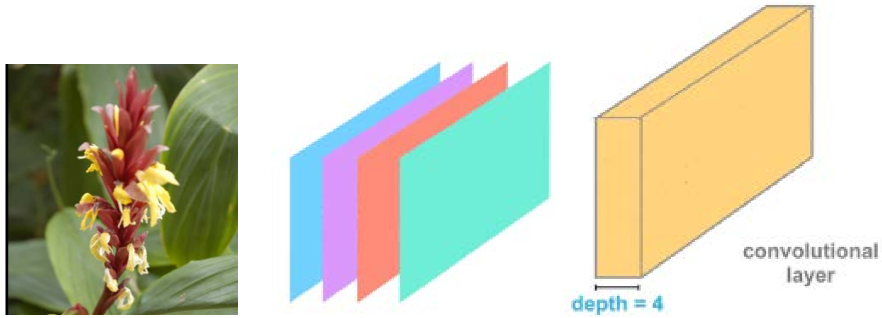

Figure 9 Left: Input Image on which 4 convolutional kernels are applied, Middle: 4 Images one from each kernel Right: The convolution layer of depth 4.

Regarding the activation function, the only requirement is that for a network to approximate a non-linear function, the activation functions must be non-linear. In Figure 10, are illustrated some examples of common activation functions: Sigmoid, Tanh (hyperbolic tangent), and ReLU (rectified linear unit).

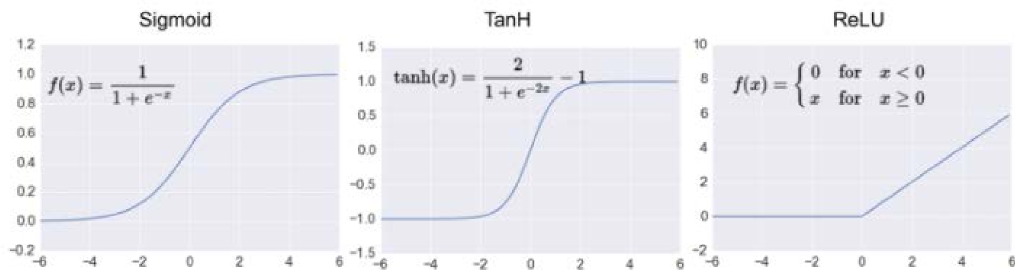

Figure 10 Three common activation functions used applied in the output of the convolution layer transforming its input accordingly

In Figure 11, an example of a 2x2 pooling kernel, with a stride of 2 , is applied to a small patch of grayscale pixel values; reducing the x-y size of the patch by a factor of 2. Only the maximum pixel values in 2x2 remain in the new, pooled output.

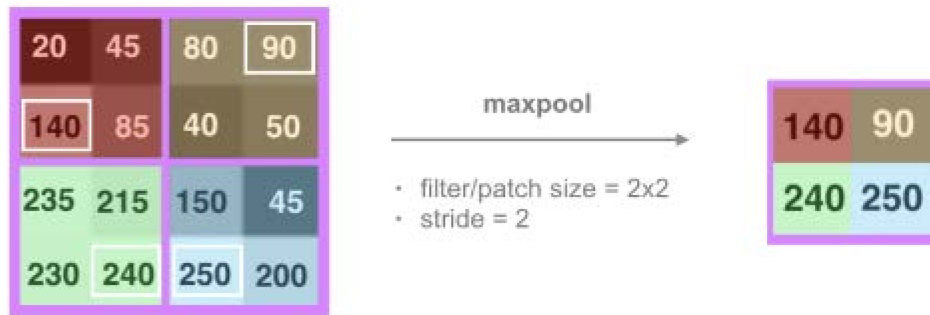

Figure 11 A maxpooling layer reduces the x-y size of an input (output of the activation function) and only keeps the most active pixel values.

VGG-19 is a convolutional neural network that is trained on more than a million images from the ImageNet database [21]. The network is 19 layers deep and is able to classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of 224-by-224 [22]. The neural network architecture is presented in Figure 12.

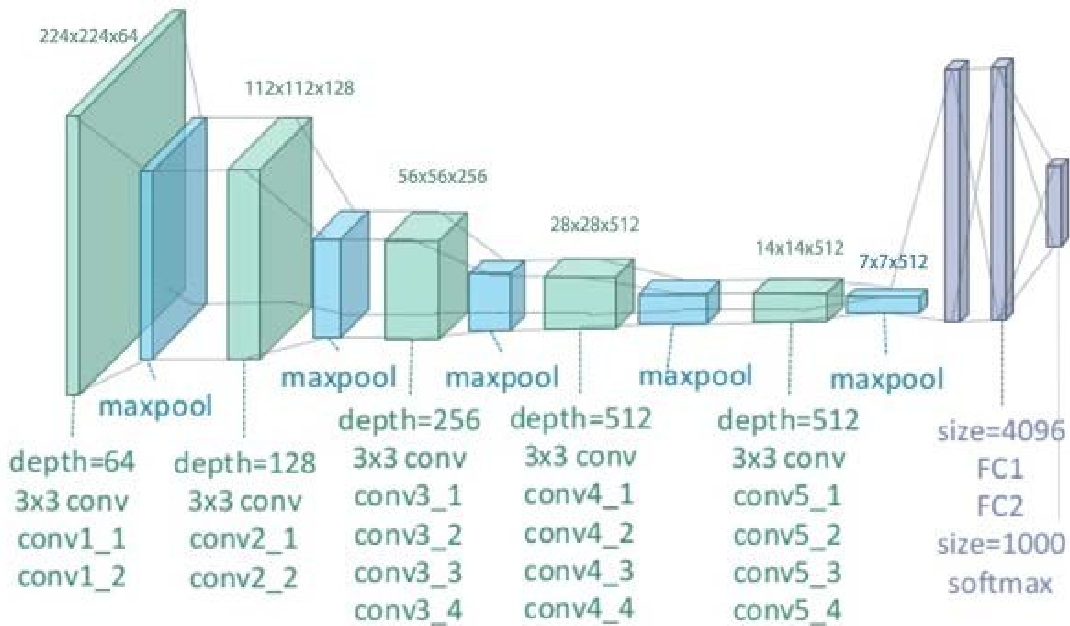

Figure 12 VGG19 Architecture

2.3 Methodological Approach and Tools

The main purpose within this work, is to build and train an image classifier so as to recognize different species of flowers that could be utilized through a phone app that would tell the name of the flower phone camera is looking at. The work done in this project is mainly split down into the following steps:

1) **Loading and Preprocessing the image dataset**
2) **Building, Training and Evaluating the image classifier**
3) **Use the trained classifier to predict image content**

Loading and Preprocessing the image dataset: Before feeding data into the model, they should be transformed into a format the model would "*understand*". Firstly, as the gathered data samples might be in a specific order, any information associated with the ordering of samples would influence the relationship between images and labels. For instance, in this study data are sorted by class corresponding to a flower category, if this data will be split into training/validation sets, these sets will not be representative of the overall distribution of data per each category. Thus, a simple best practice to ensure the model will not be affected from data ordering is shuffling the data and then make the split into training and validation sets assuring that transforming both training and validation data in the same way. In this study, the data for each category were firstly shuffled and then split to use 80% of the samples for training and 20% for validation.

Building, Training and Evaluating the image classifier: The composition and construction of the input layer and the intermediate layers of the model has been previously described. For a multi-class classification, like the one in the present stud, the model should output one probability score per class and the summation of these scores should aggregate to 1. For an example of four classes, outputting {0: 0.4, 1: 0.3, 2: 0.2, 3: 0.1 } means "40% confidence that this sample is in class 0, 30% that it is in class 1, 20% that it is in class 2 and 10% that is in class 3." To output these scores, the activation function of the last layer should be softmax, and the loss function used to train the model should be categorical cross-entropy as presented in Figure 13. This study elaborates the prediction of 102 distinct flower categories.

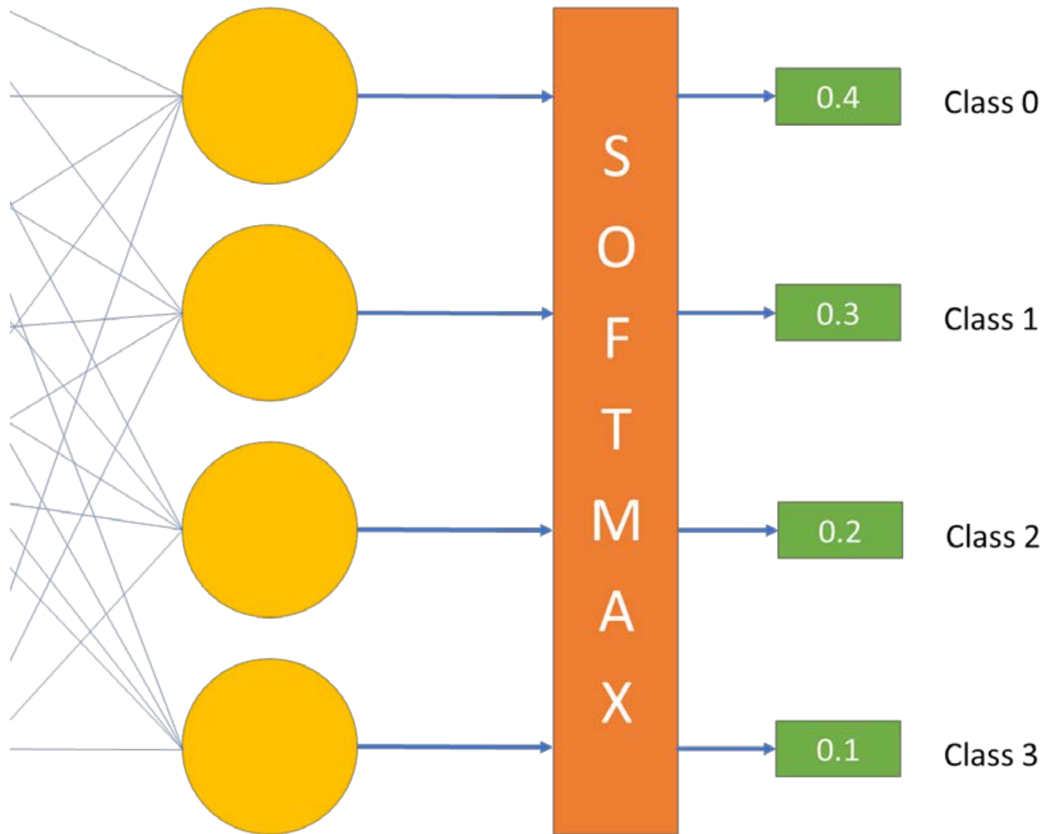

Figure 13 Multi Class Classification Output Layer

Now that the model architecture has defined and all the model layers has constructed, the model training can start taking place. This process involves making a prediction based on the current (untrained) state of the model, calculating how "*incorrect*" the prediction is, and updating the weights or parameters of the neural network so as to minimize this error and eventually make the model to predict more efficiently. This process is repeated until the model has converged and can no longer learn. The key learning parameters taken into account within this self-assessment and optimization process are presented and shortly described in Table 1.

Table 1

Learning Parameter	Description
Metric	How to measure the performance of our model using a metric.
Loss function - multi class classification	A function that is used to calculate a loss value that the training process then attempts to minimize by tuning the network weights.
Optimizer	A function that decides how the network weights will be updated based on the output of the loss function.

The actual training takes place using the fitting method which, depending on the dataset size, is where the most computing cycles will be spent. For each training iteration, a batch size number of images from the training data are used to compute the loss, whereas the weights are updated once, based on this value. The training process completes an epoch once the model has seen the entire training dataset. In the end of each epoch, the validation dataset is utilized in order to evaluate the model performance with regards learning process. The training process is repeated using the dataset for a predetermined number of epochs until the validation accuracy stabilizes between consecutive epochs, suggesting that the model is not training anymore. The hyperparameters involved within this process concern a) the ones of the CNN which will not be tuned in the present study as in the end the architecture of the pretrained VGG19 will be employed for the needs of the project (presented and described in Table 2) and b) the ones of the fully connected neural network which will be fine-tuned in the coming chapter, in order to increase the performance of the prediction model regarding the particular dataset of the study (presented and shortly described in Table 3)

Table 2 CNN Training Hyperparameters

Training hyperparameter	Used in present study	Description
Convolutional and MaxPool Layers	CNN: VGG19 Architecture	Configuration of the Convolutional and MaxPool Layers
Number of units per layer	64	Feature identifiers - The units in a layer must hold the information for the transformation that a layer performs
Kernel size	(3,3) or 2 (MaxPooling)	The size of the convolution window.
Stride	(1,1) or 1 or 2 (MaxPooling)	Controls how the filter convolves around the input volume
Padding	(1,1) or 0 (MaxPooling)	Pads the input volume with zeros around the border so as the output volume to retain its input dimensions

Table 3 Fully Connected Feed Forward Neural Network Hyperparameters

Training hyperparameter	Description
Number of layers in the model	The number of layers in a neural network is an indicator of its complexity
Batch size	Number of images used for each training iteration

Number of units per layer	Units in a layer hold the information for the transformation that a layer performs and they are the number of nodes (neurons) each layer is consisted of
Dropout rate	Dropout layers are used in the model for regularization.
Learning Rate	The rate at which the neural network weights change between iterations.

For defining and training the model several hyperparameters must be chosen, initially based upon intuition, examples or/and best practice recommendations which, however, may not yield the best results but providing a starting point for training. As every problem is different, tuning those hyperparameters will help to refine the model in a manner that will represent better the particularities of the problem itself. Let's take a look at some of the aforementioned hyperparameters and what it means tuning them:

Number of layers in the model: The number of layers in a neural network is an indicator of its complexity. The more layers allow the model to learn more information about the training data, with the precaution of causing overfitting. Less layers could negatively affect the model's learning ability leading to underfitting.

Number of units per layer: The units in a layer hold the information for the transformation that a layer performs. For the first layer, this is driven by the number of features whereas in subsequent layers, the number of units depends on the choice of expanding or contracting the representation from the previous layer. The question here is to successfully manage to minimize the information loss between layers.

Dropout and Learning rate: Dropout layers define the fraction of input to drop as a precaution for overfitting used this way in the model for regularization. The rate at which the neural network weights change between iterations is called learning rate. A large learning rate may cause large fluctuation in the

weights, resulting in difficulty to find their optimal values. A low learning rate would be nice, but the model will need more iterations in order to converge.

Playing around with the values of those hyperparameters in the coming chapter will determine the configuration of which one results in better model performance. Once the best-performing hyperparameters are determined the model is ready to be deployed.

3 Results and Discussion

3.1 Loading and Preprocessing the image dataset

After the completion of the project, an application that can be trained on any set of labeled images will be delivered as the neural network will be learning about flowers and end up as a command line application. As explained in the previous chapter, the implementation is going to take place on the platform jupyter notebook and the respective code snapshots will be demonstrated in figures throughout the present chapter.

In the beginning the needed pytorch libraries and packages are imported as presented in Figure 14. Then the data is downloaded, loaded and the dataset is randomly split into two subsets: training and validation (Figure 15). The training and validation datasets are comprised 80% and 20% out of the initial dataset, respectively. In order to improve the neural network generalization ability that leads to higher performance, different transformations such as random scaling, cropping, and flipping are applied over the training dataset. Due to the fact that a pre-trained neural network is going to be used, the input data is resized to 256x256 and cropped to 224x224 pixels as it is pre-requisite for the particular utilized model, followed by data Normalization, as presented in Figure 16. As the validation set is used to measure the model's performance on data it hasn't seen yet, there should not applied any scaling or rotation transformations, despite resizing and then cropping the images to the appropriate size. The implementation code that explains the aforementioned, is presented in Figure 16.

```
In [1]: # Imports here
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import datasets, models
import torchvision.transforms as transforms
import numpy as np
from PIL import Image
import torch.optim as optim
import matplotlib.pyplot as plt
from collections import OrderedDict
import json
import os, random
from torch.autograd import Variable
```

Figure 14 Importing the needed libraries

```
In [2]: ######################### Split to Train and Test ################################

def split_data_to_train_test_subsets(data_directory, directory_training_data, directory_testing_data, percentage_testing_data):
    if directory_testing_data.count('/') > 1:
        rmtree(directory_testing_data, ignore_errors=False)
        makedirs(directory_testing_data)

    if directory_training_data.count('/') > 1:
        rmtree(directory_training_data, ignore_errors=False)
        makedirs(directory_training_data)

    num_training_files = 0
    num_testing_files = 0

    for subdir, dirs, files in walk(data_directory):
        category_name = path.basename(subdir)

        if category_name == path.basename(data_directory):
            continue

        training_data_category_dir = directory_training_data + '/' + category_name
        testing_data_category_dir = directory_testing_data + '/' + category_name

        if not path.exists(training_data_category_dir):
            mkdir(training_data_category_dir)

        if not path.exists(testing_data_category_dir):
            mkdir(testing_data_category_dir)

        for file in files:
            input_file = path.join(subdir, file)
            if rand(1) < percentage_testing_data:
                copy(input_file, directory_testing_data + '/' + category_name + '/' + file)
                num_testing_files += 1
            else:
                copy(input_file, directory_training_data + '/' + category_name + '/' + file)
                num_training_files += 1
```

Figure 15 Splitting Training and Validation (labeled testing) datasets

```
In [4]: # Define your transforms for the training and validation sets

# define dataloader parameters
batch_size = 32
# convert data to a normalized torch.FloatTensor
train_transforms = transforms.Compose([transforms.RandomRotation(30),
                                       transforms.RandomResizedCrop(224),
                                       transforms.RandomHorizontalFlip(),
                                       transforms.ToTensor(),
                                       transforms.Normalize([0.485, 0.456, 0.406],
                                                            [0.229, 0.224, 0.225])])

valid_transforms = transforms.Compose([transforms.Resize(256),
                                      transforms.CenterCrop(224),
                                      transforms.ToTensor(),
                                      transforms.Normalize([0.485, 0.456, 0.406],
                                                           [0.229, 0.224, 0.225])])

#test_transforms = transforms.Compose([transforms.Resize(256),
#                                      transforms.CenterCrop(224),
#                                      transforms.ToTensor(),
#                                      transforms.Normalize([0.485, 0.456, 0.406],
#                                                           [0.229, 0.224, 0.225])
#                                      ])

# Load the datasets with ImageFolder
train_data = datasets.ImageFolder(data_dir + '/train', transform=train_transforms)
valid_data = datasets.ImageFolder(data_dir + '/valid', transform=valid_transforms)
#test_data  = datasets.ImageFolder(test_dir, transform=test_transforms)


# Using the image datasets and the trainforms, define the dataloaders
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=True)
valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=batch_size)
#test_loader  = torch.utils.data.DataLoader(test_data, batch_size=batch_size)
loaders = [train_loader, valid_loader]#, test_loader]
```

Figure 16 Define dataloader parameters and transformations for training and validation datasets

The available torchvision pre-trained neural networks trained on the ImageNet dataset, have each color channel normalized separately. This is the reason why both subsets have to have the means and standard deviations of the images normalized because this is what the trained neural network expects. For the means, it's [0.485, 0.456, 0.406] and for the standard deviations [0.229, 0.224, 0.225], calculated from the ImageNet images. These values will shift each color channel to be centered at 0 and range from -1 to 1. Another important step before proceeding to building and training the classifier is the data label mapping; load in a mapping from category label to category name. Accompanied with the data there is a file (JSON object that can be read in with the json module) named cat_to_name.json which in fact is a dictionary mapping the integer encoded categories to the actual names of the flowers (Figure 17).

```
In [5]: with open('cat_to_name.json', 'r') as f:
            cat_to_name = json.load(f)

        print(cat_to_name)
        print("\n Length:", len(cat_to_name))

        {'21': 'fire lily', '3': 'canterbury bells', '45': 'bolero deep blue', '1': 'pink primrose', '34': 'mexican aster', '27': 'prin
        ce of wales feathers', '7': 'moon orchid', '16': 'globe-flower', '25': 'grape hyacinth', '26': 'corn poppy', '79': 'toad lily',
        '39': 'siam tulip', '24': 'red ginger', '67': 'spring crocus', '35': 'alpine sea holly', '32': 'garden phlox', '10': 'globe thi
        stle', '6': 'tiger lily', '93': 'ball moss', '33': 'love in the mist', '9': 'monkshood', '102': 'blackberry lily', '14': 'spear
        thistle', '19': 'balloon flower', '100': 'blanket flower', '13': 'king protea', '49': 'oxeye daisy', '15': 'yellow iris', '61':
        'cautleya spicata', '31': 'carnation', '64': 'silverbush', '68': 'bearded iris', '63': 'black-eyed susan', '69': 'windflower',
        '62': 'japanese anemone', '20': 'giant white arum lily', '38': 'great masterwort', '4': 'sweet pea', '86': 'tree mallow', '10
        1': 'trumpet creeper', '42': 'daffodil', '22': 'pincushion flower', '2': 'hard-leaved pocket orchid', '54': 'sunflower', '66':
        'osteospermum', '70': 'tree poppy', '85': 'desert-rose', '99': 'bromelia', '87': 'magnolia', '5': 'english marigold', '92': 'be
        e balm', '28': 'stemless gentian', '97': 'mallow', '57': 'gaura', '40': 'lenten rose', '47': 'marigold', '59': 'orange dahlia',
        '48': 'buttercup', '55': 'pelargonium', '36': 'ruby-lipped cattleya', '91': 'hippeastrum', '29': 'artichoke', '71': 'gazania',
        '90': 'canna lily', '18': 'peruvian lily', '98': 'mexican petunia', '8': 'bird of paradise', '30': 'sweet william', '17': 'purp
        le coneflower', '52': 'wild pansy', '84': 'columbine', '12': "colt's foot", '11': 'snapdragon', '96': 'camellia', '23': 'fritil
        lary', '50': 'common dandelion', '44': 'poinsettia', '53': 'primula', '72': 'azalea', '65': 'californian poppy', '80': 'anthuri
        um', '76': 'morning glory', '37': 'cape flower', '56': 'bishop of llandaff', '60': 'pink-yellow dahlia', '82': 'clematis', '5
        8': 'geranium', '75': 'thorn apple', '41': 'barbeton daisy', '95': 'bougainvillea', '43': 'sword lily', '83': 'hibiscus', '78':
        'lotus lotus', '88': 'cyclamen', '94': 'foxglove', '81': 'frangipani', '74': 'rose', '89': 'watercress', '73': 'water lily', '4
        6': 'wallflower', '77': 'passion flower', '51': 'petunia'}

         Length: 102
```

Figure 17 Mapping the integer encoded categories to the actual names of the flowers

3.2 Selection between Pretrained and Untrained Model

Now that the data is ready to be used, the decision whether to choose an untrained or pretrained model has to be taken. Figure 18 (lower) shows the pretrained VGG19 model to be employed while the training processes has been frozen within all the features layers of the neural network. For demonstration purposes, the same architecture (Figure 19) but untrained neural network model will be trained from scratch (Figure 18 upper) resulting in respective validation loss but after a longer period of training

(greater number of epochs), as it can be compared from Figure 20 and Figure 21. Thus, the pretrained VGG19 model has been chosen to proceed from now on.

```
In [6]: # Load the pretrained model from pytorch
        myModel = models.vgg19(pretrained=False)

        # Freeze training for all "features" layers
        for param in myModel.features.parameters():
            param.requires_grad = True
```

```
In [6]: # Load the pretrained model from pytorch
        myModel = models.vgg19(pretrained=True)

        # Freeze training for all "features" layers
        for param in myModel.features.parameters():
            param.requires_grad = False
```

Figure 18 Upper: Loading the untrained model and allow training for all features layers, Lower: Loading the pretrained model and freeze training for all features layers

```
In [9]: myModel

Out[9]: VGG(
          (features): Sequential(
            (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (1): ReLU(inplace)
            (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (3): ReLU(inplace)
            (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
            (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (6): ReLU(inplace)
            (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (8): ReLU(inplace)
            (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
            (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (11): ReLU(inplace)
            (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (13): ReLU(inplace)
            (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (15): ReLU(inplace)
            (16): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (17): ReLU(inplace)
            (18): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
            (19): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (20): ReLU(inplace)
            (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (22): ReLU(inplace)
            (23): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (24): ReLU(inplace)
            (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (26): ReLU(inplace)
            (27): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
            (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (29): ReLU(inplace)
            (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (31): ReLU(inplace)
            (32): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (33): ReLU(inplace)
            (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (35): ReLU(inplace)
            (36): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
          )
          (classifier): Sequential(
            (0): Linear(in_features=25088, out_features=4096, bias=True)
            (1): ReLU(inplace)
            (2): Dropout(p=0.5)
            (3): Linear(in_features=4096, out_features=4096, bias=True)
            (4): ReLU(inplace)
            (5): Dropout(p=0.5)
            (6): Linear(in_features=4096, out_features=1000, bias=True)
          )
        )
```

Figure 19 Default Layers and Features of VGG19 - Model Architecture

In [14]:

```python
# number of epochs to train the model
n_epochs = 200

valid_loss_min = np.Inf # track change in validation loss

for epoch in range(1, n_epochs+1):

    # keep track of training and validation loss
    train_loss = 0.0
    valid_loss = 0.0

    ###################
    # train the model #
    ###################
    myModel.cuda().train()
    for data, target in train_loader:
        # move tensors to GPU if CUDA is available
        if train_on_gpu:
            data, target = data.cuda(), target.cuda()
        # clear the gradients of all optimized variables
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = myModel(data)
        # calculate the batch loss
        loss = criterion(output, target)
        # backward pass: compute gradient of the loss with respect to model parameters
        loss.backward()
        # perform a single optimization step (parameter update)
        optimizer.step()
        # update training loss
        train_loss += loss.item()*data.size(0)

    ######################
    # validate the model #
    ######################
    myModel.eval()
    for data, target in valid_loader:
        # move tensors to GPU if CUDA is available
        if train_on_gpu:
            data, target = data.cuda(), target.cuda()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = myModel(data)
        # calculate the batch loss
        loss = criterion(output, target)
        # update average validation loss
        valid_loss += loss.item()*data.size(0)

    # calculate average losses
    train_loss = train_loss/len(train_loader.dataset)
    valid_loss = valid_loss/len(valid_loader.dataset)

    # print training/validation statistics
    print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
        epoch, train_loss, valid_loss))

    # save model if validation loss has decreased
    if valid_loss <= valid_loss_min:
        print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.format(
        valid_loss_min,
        valid_loss))
        torch.save(myModel.state_dict(), 'checkpoint_vgg19_001lr_200ep_32ba_25088_4096_4096_untrained.pt')
        valid_loss_min = valid_loss
```

```
Epoch: 182      Training Loss: 2.839024         Validation Loss: 2.578535
Epoch: 183      Training Loss: 2.840415         Validation Loss: 2.568139
Epoch: 184      Training Loss: 2.831100         Validation Loss: 2.619749
Epoch: 185      Training Loss: 2.851480         Validation Loss: 2.597519
Epoch: 186      Training Loss: 2.845908         Validation Loss: 2.651638
Epoch: 187      Training Loss: 2.845529         Validation Loss: 2.514817
Validation loss decreased (2.563189 --> 2.514817).  Saving model ...
Epoch: 188      Training Loss: 2.829862         Validation Loss: 2.540378
Epoch: 189      Training Loss: 2.842853         Validation Loss: 2.530801
Epoch: 190      Training Loss: 2.829562         Validation Loss: 2.622636
Epoch: 191      Training Loss: 2.807395         Validation Loss: 2.593098
Epoch: 192      Training Loss: 2.837705         Validation Loss: 2.543316
```

Figure 20 Training and Validating the untrained VGG19 Pytorch Model with the flower dataset of the present study

In [11]:

```python
# number of epochs to train the model
n_epochs = 30

valid_loss_min = np.Inf # track change in validation loss

for epoch in range(1, n_epochs+1):

    # keep track of training and validation loss
    train_loss = 0.0
    valid_loss = 0.0

    ###################
    # train the model #
    ###################
    myModel.cuda().train()
    for data, target in train_loader:
        # move tensors to GPU if CUDA is available
        if train_on_gpu:
            data, target = data.cuda(), target.cuda()
        # clear the gradients of all optimized variables
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = myModel(data)
        # calculate the batch loss
        loss = criterion(output, target)
        # backward pass: compute gradient of the loss with respect to model parameters
        loss.backward()
        # perform a single optimization step (parameter update)
        optimizer.step()
        # update training loss
        train_loss += loss.item()*data.size(0)

    ######################
    # validate the model #
    ######################
    myModel.eval()
    for data, target in valid_loader:
        # move tensors to GPU if CUDA is available
        if train_on_gpu:
            data, target = data.cuda(), target.cuda()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = myModel(data)
        # calculate the batch loss
        loss = criterion(output, target)
        # update average validation loss
        valid_loss += loss.item()*data.size(0)

    # calculate average losses
    train_loss = train_loss/len(train_loader.dataset)
    valid_loss = valid_loss/len(valid_loader.dataset)

    # print training/validation statistics
    print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
        epoch, train_loss, valid_loss))

    # save model if validation loss has decreased
    if valid_loss <= valid_loss_min:
        print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.format(
        valid_loss_min,
        valid_loss))
        torch.save(myModel.state_dict(), 'checkpoint_vgg19_001lr_200ep_32ba_25088_4096_4096_pretrained_GPU.pt')
        valid_loss_min = valid_loss
```

```
Epoch: 1 	Training Loss: 4.255613 	Validation Loss: 3.445646
Validation loss decreased (inf --> 3.445646).  Saving model ...
Epoch: 2 	Training Loss: 3.245429 	Validation Loss: 2.147629
Validation loss decreased (3.445646 --> 2.147629).  Saving model ...
Epoch: 3 	Training Loss: 2.327855 	Validation Loss: 1.463941
Validation loss decreased (2.147629 --> 1.463941).  Saving model ...
Epoch: 4 	Training Loss: 1.810564 	Validation Loss: 1.053890
Validation loss decreased (1.463941 --> 1.053890).  Saving model ...
Epoch: 5 	Training Loss: 1.498170 	Validation Loss: 0.820607
Validation loss decreased (1.053890 --> 0.820607).  Saving model ...
Epoch: 6 	Training Loss: 1.298648 	Validation Loss: 0.738670
Validation loss decreased (0.820607 --> 0.738670).  Saving model ...
Epoch: 7 	Training Loss: 1.172519 	Validation Loss: 0.619789
Validation loss decreased (0.738670 --> 0.619789).  Saving model ...
Epoch: 8 	Training Loss: 1.041657 	Validation Loss: 0.564265
Validation loss decreased (0.619789 --> 0.564265).  Saving model ...
Epoch: 9 	Training Loss: 0.988476 	Validation Loss: 0.548490
Validation loss decreased (0.564265 --> 0.548490).  Saving model ...
Epoch: 10 	Training Loss: 0.942167 	Validation Loss: 0.482100
Validation loss decreased (0.548490 --> 0.482100).  Saving model ...
Epoch: 11 	Training Loss: 0.900232 	Validation Loss: 0.501467
Epoch: 12 	Training Loss: 0.821344 	Validation Loss: 0.446393
Validation loss decreased (0.482100 --> 0.446393).  Saving model ...
Epoch: 13 	Training Loss: 0.791335 	Validation Loss: 0.448768
Epoch: 14 	Training Loss: 0.772125 	Validation Loss: 0.416626
Validation loss decreased (0.446393 --> 0.416626).  Saving model ...
Epoch: 15 	Training Loss: 0.736453 	Validation Loss: 0.382080
Validation loss decreased (0.416626 --> 0.382080).  Saving model ...
Epoch: 16 	Training Loss: 0.695318 	Validation Loss: 0.383362
Epoch: 17 	Training Loss: 0.698901 	Validation Loss: 0.373015
Validation loss decreased (0.382080 --> 0.373015).  Saving model ...
Epoch: 18 	Training Loss: 0.668468 	Validation Loss: 0.373300
Epoch: 19 	Training Loss: 0.647225 	Validation Loss: 0.390713
Epoch: 20 	Training Loss: 0.620957 	Validation Loss: 0.323891
Validation loss decreased (0.373015 --> 0.323891).  Saving model ...
Epoch: 21 	Training Loss: 0.623595 	Validation Loss: 0.349870
Epoch: 22 	Training Loss: 0.594082 	Validation Loss: 0.334403
Epoch: 23 	Training Loss: 0.564929 	Validation Loss: 0.326531
Epoch: 24 	Training Loss: 0.555890 	Validation Loss: 0.351933
Epoch: 25 	Training Loss: 0.562260 	Validation Loss: 0.310228
Validation loss decreased (0.323891 --> 0.310228).  Saving model ...
Epoch: 26 	Training Loss: 0.550379 	Validation Loss: 0.309827
Validation loss decreased (0.310228 --> 0.309827).  Saving model ...
Epoch: 27 	Training Loss: 0.537390 	Validation Loss: 0.312703
Epoch: 28 	Training Loss: 0.509479 	Validation Loss: 0.303337
Validation loss decreased (0.309827 --> 0.303337).  Saving model ...
Epoch: 29 	Training Loss: 0.525954 	Validation Loss: 0.325178
Epoch: 30 	Training Loss: 0.519544 	Validation Loss: 0.288626
Validation loss decreased (0.303337 --> 0.288626).  Saving model ...
```

Figure 21 Training and Validating the pretrained VGG19 Pytorch Model with the flower dataset of the present study

3.3 Building the Image Classifier

Now that the data is ready and the pretrained convolutional neural network has been selected, it's time to build and train the classifier. The pretrained model from torchvision models will be utilized to get the image features and the new feed-forward classifier will be developed and trained using those features. As illustrated in the code of Figure 22, the untrained feed-forward network is defined as a classifier, using ReLU as the activation function for each layer and dropout (p=0.5) but the number of output features of the last layer is reduced from 1000 (classes that VGG19 has been pretrained) to 102 (classes of flowers regarding the present study) and the activation function is the LogSoftmax due to the multi class classification. This time, the classifier is trained by backpropagation using the pre-trained network to get the features and track the loss and accuracy on the validation set, so as to ensure that only the weights of the feed-forward network are updated. The analytic results of the optimization process for the different hyperparameters (learning rate, batch size, loss function, epochs, etc.) in order to find the best model, are following in the coming subchapter. The optimized hyperparameters will be saved with the trained model and used as the default values in the coming part of the project.

```
In [9]: ### Define a new, untrained feed-forward network as a classifier, using ReLU activations and dropout

        classifier = nn.Sequential(nn.Linear(in_features=25088, out_features=4096, bias=True),
                                   nn.ReLU(inplace=True),
                                   nn.Dropout(p = 0.5),
                                   nn.Linear(in_features=4096, out_features=4096, bias=True),
                                   nn.ReLU(inplace=True),
                                   nn.Dropout(p = 0.5),
                                   nn.Linear(in_features=4096, out_features=len(cat_to_name), bias=True),
                                   nn.LogSoftmax(dim=1))
```

Figure 22 Definition of the untrained feed-forward network as the classifier

3.4 Model Fine Tuning and Feature Extraction

There are two types of transfer learning used to follow in the optimization process of a prediction model: a) finetuning and b) feature extraction. In finetuning, all of the pretrained model's parameters are updated within the framework of the particular new task, which in other words means to retrain the whole model whereas, feature extraction only the model's final layer weights, from which predictions is

derived, are updated. It is called feature extraction because the pretrained CNN is utilized as a fixed feature-extractor, and only the output layer is changed. Both transfer learning methods follow the same steps:

- Initialize the pretrained model
- Reshape the final layer(s) to have the same number of outputs as the number of classes in the new dataset
- Define for the optimization algorithm which parameters we want to update during training
- Run the training step

Running some preliminary training steps for the learning parameters presented in Table 4, it was decided to assess the performance of the model accounting the metrics of validation loss and validation accuracy. Due to the fact that there is a multiclass classification problem, the combination of the loss function and optimizer that resulted in better model performance (according to the aforementioned metrics) was the Cross – Entropy Loss and the Stochastic Gradient Descent, respectively. The implementation code is presented in Figure 23.

Table 4 Learning Parameters for Model

Learning Parameters	Used in this study
Metric	Validation Loss / Accuracy
Loss function - multi class classification	Cross-entropy loss
Optimizer	Stochastic Gradient Descent

```
In [11]: # specify loss function (categorical cross-entropy)
criterion = nn.CrossEntropyLoss()

# specify optimizer (stochastic gradient descent) and learning rate = 0.01
optimizer = optim.SGD(myModel.classifier.parameters(), lr=0.01)
```

Figure 23 Defining the model hyperparameter (loss function, optimizer and learing rate)

The batch size was another parameter over which the model assessed by keeping the rest of the VGG19 default parameters stable, resulting in best model performance (minimum validation loss) when the batch size is of 32 images as presented in Table 5. A similar approach was followed for the learning rate (Table 6), the number of inputs per layer (Table 7) and the drop out (Table 8) with the difference that the determined optimum learning parameter from the iterative process was used as input, keeping the rest VGG19 default parameters stable. Eventually, all the determined optimal parameters are summarized in Table 9 which in the coming chapter will be combined in a series of model simulations in order to determine the parameters configuration resulting in the model's best performance.

Table 5 Assessing model performance with respect to the batch size parameter

Description	Batch Size	Minimum Validation Loss
Pretrained VGG19, Default Hyperparameters, epochs = 30	8	0.76
Pretrained VGG19, Default Hyperparameters, epochs = 30	16	0.65
Pretrained VGG19, Default Hyperparameters, epochs = 30	**32**	**0.28**
Pretrained VGG19, Default Hyperparameters, epochs = 30	64	CUDA Out of Memory

Table 6 Assessing model performance with respect to the learning rate parameter

Description	Learning Rate	Minimum Validation Loss
Pretrained VGG19, Default Hyperparameters, batch size = 32, epochs = 160	**0.001**	**0.28**

Pretrained VGG19, Default Hyperparameters, batch size = 32, epochs = 30	**0.01**	**0.28**
Pretrained VGG19, Default Hyperparameters, batch size = 32, epochs = 30	0.1	1.17
Pretrained VGG19, Default Hyperparameters, batch size = 32, epochs = 30	1	NaN

Table 7 Assessing model performance with respect to the number of inputs per layer

Description	Number of inputs per layer	Minimum Validation Loss
Pretrained VGG19, batch size = 32, Drop out = 0.5, epochs = 30	**25088 (IN) -> 12544 (OUT) -> 12544 (IN) -> 6272 (OUT) -> 6272 (IN) -> 102 (OUT)**	**0.28**
Pretrained VGG19, batch size = 32, Drop out = 0.5, epochs = 30	**25088 (IN) -> 4096 (OUT) -> 4096 (IN) -> 4096 (OUT) -> 4096 (IN) -> 102 (OUT)**	**0.28**
Pretrained VGG19, batch size = 32, Drop out = 0.5, epochs = 30	25088 (IN) -> 1632 (OUT) -> 1632 (IN) -> 408 (OUT) -> 408 (IN) -> 102 (OUT)	0.31
Pretrained VGG19, batch size = 32, Drop out = 0.5, epochs = 30	25088 (IN) -> 408 (OUT) -> 408 (IN) -> 408 (OUT) -> 408 (IN) -> 102 (OUT)	0.37

Table 8 Assessing model performance with respect to the drop out parameter

Description	Drop Out	Minimum Validation Loss
Pretrained VGG19, batch size = 32, epochs = 30	**0.1**	**0.27**
Pretrained VGG19, batch size = 32, epochs = 30	0.2	0.30
Pretrained VGG19, batch size = 32, epochs = 30	0.3	0.29
Pretrained VGG19, batch size = 32, epochs = 30	0.4	0.28
Pretrained VGG19, batch size = 32, epochs = 30	0.5	0.28

Table 9

Training hyperparameter	Value used in this study after Fine Tuning
Number of layers in the model	Fully Connected Feed Forward Network: 3 layers
Batch size	32[1]
Number of inputs per layer	25088 (IN) -> 4096 (OUT) -> 4096 (IN) -> 4096 (OUT) -> 4096 (IN) -> 102 (OUT) And

[1] Maximum number due to limitation to VGA available memory

	25088 (IN) -> 12544 (OUT) -> 12544 (IN) -> 6272 (OUT) -> 6272 (IN) -> 102 (OUT)
Dropout rate	0.1
Learning Rate	0.001 and 0.01

3.5 Build, Train and Validate the Model

A series of model simulations with various combinations of optimal hyperparameters as they were determined in the previous chapter, is illustrated in Table 10, providing the insight of the optimal hyperparameter configuration that will result in the model's best performance. From the same table, the optimal hyperparameters configuration achieved a 0.26 model validation loss and 94% model validation accuracy and the implementation code for this is illustrated in Figure 24 while in Figure 26 the implementation code for the calculation of the validation accuracy.

Table 10

Description	Minimum Validation Loss	Maximum Validation Accuracy
Pretrained VGG19, batch size = 32, Drop out = 0.1, epochs = 30, FFC (12544->6272), lr = 0.01	0.27	0.93
Pretrained VGG19, batch size = 32, Drop out = 0.1, epochs = 30, FFC (1632 -> 408), lr = 0.01	0.3	0.88
Pretrained VGG19, batch size = 32, Drop out = 0.1, epochs =	**0.26**	**0.94**

160, FFC (12544->6272), lr = 0.001		
Pretrained VGG19, batch size = 32, Drop out = 0.1, epochs = 160, FFC (1632 -> 408), lr = 0.001	0.29	0.89

```
In [11]: # number of epochs to train the model
n_epochs = 160

valid_loss_min = np.Inf # track change in validation loss

for epoch in range(1, n_epochs+1):

    # keep track of training and validation loss
    train_loss = 0.0
    valid_loss = 0.0

    ###################
    # train the model #
    ###################
    myModel.cuda().train()
    for data, target in train_loader:
        # move tensors to GPU if CUDA is available
        if train_on_gpu:
            data, target = data.cuda(), target.cuda()
        # clear the gradients of all optimized variables
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = myModel(data)
        # calculate the batch loss
        loss = criterion(output, target)
        # backward pass: compute gradient of the loss with respect to model parameters
        loss.backward()
        # perform a single optimization step (parameter update)
        optimizer.step()
        # update training loss
        train_loss += loss.item()*data.size(0)

    ######################
    # validate the model #
    ######################
    myModel.eval()
    for data, target in valid_loader:
        # move tensors to GPU if CUDA is available
        if train_on_gpu:
            data, target = data.cuda(), target.cuda()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = myModel(data)
        # calculate the batch loss
        loss = criterion(output, target)
        # update average validation loss
        valid_loss += loss.item()*data.size(0)

    # calculate average losses
    train_loss = train_loss/len(train_loader.dataset)
    valid_loss = valid_loss/len(valid_loader.dataset)

    # print training/validation statistics
    print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
        epoch, train_loss, valid_loss))

    # save model if validation loss has decreased
    if valid_loss <= valid_loss_min:
        print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.format(
        valid_loss_min,
        valid_loss))
        torch.save(myModel.state_dict(), 'checkpoint_vgg19_0001lr_30ep_32ba_01drout_25088_12544_6272.pt')
        valid_loss_min = valid_loss
```

```
Epoch: 145      Training Loss: 0.426343         Validation Loss: 0.281120
Epoch: 146      Training Loss: 0.413029         Validation Loss: 0.276643
Validation loss decreased (0.277599 --> 0.276643).  Saving model ...
Epoch: 147      Training Loss: 0.409134         Validation Loss: 0.284604
Epoch: 148      Training Loss: 0.401059         Validation Loss: 0.271806
Validation loss decreased (0.276643 --> 0.271806).  Saving model ...
Epoch: 149      Training Loss: 0.417049         Validation Loss: 0.272237
Epoch: 150      Training Loss: 0.423867         Validation Loss: 0.282101
Epoch: 151      Training Loss: 0.404585         Validation Loss: 0.283478
Epoch: 152      Training Loss: 0.398536         Validation Loss: 0.283590
Epoch: 153      Training Loss: 0.393251         Validation Loss: 0.272153
Epoch: 154      Training Loss: 0.419675         Validation Loss: 0.283599
Epoch: 155      Training Loss: 0.396002         Validation Loss: 0.265703
Validation loss decreased (0.271806 --> 0.265703).  Saving model ...

Epoch: 156      Training Loss: 0.406657         Validation Loss: 0.275358
Epoch: 157      Training Loss: 0.395286         Validation Loss: 0.269022
Epoch: 158      Training Loss: 0.411410         Validation Loss: 0.263258
Validation loss decreased (0.265703 --> 0.263258).  Saving model ...
Epoch: 159      Training Loss: 0.374210         Validation Loss: 0.266356
Epoch: 160      Training Loss: 0.393551         Validation Loss: 0.268827
```

Figure 24 Implementation code for training and validating the model and determine the optimal hyperparameters configuration

```python
In [*]: ###Train and Validate the Model
n_epochs = 20
steps = 0

running_loss = 0
accuracy = 0

cuda = torch.cuda.is_available()

if cuda:
    myModel.cuda()
else:
    myModel.cpu()

for e in range(n_epochs):

    train_mode = 0
    valid_mode = 1

    for mode in [train_mode, valid_mode]:
        if mode == train_mode:
            myModel.train()
        else:
            myModel.eval()

        pass_count = 0

        for data in loaders[mode]:
            pass_count += 1
            inputs, labels = data
            if cuda == True:
                inputs, labels = Variable(inputs.cuda()), Variable(labels.cuda())
            else:
                inputs, labels = Variable(inputs), Variable(labels)

            optimizer.zero_grad()
            # Forward
            output = myModel.forward(inputs)
            loss = criterion(output, labels)
            # Backward
            if mode == train_mode:
                loss.backward()
                optimizer.step()

            running_loss += loss.item()
            ps = torch.exp(output).data
            equality = (labels.data == ps.max(1)[1])
            accuracy = equality.type_as(torch.cuda.FloatTensor()).mean()

        if mode == train_mode:
            print("\nEpoch: {}/{} ".format(e+1, n_epochs),
                  "\nTraining Loss: {:.4f}  ".format(running_loss/pass_count))
        else:
            print("Validation Loss: {:.4f}  ".format(running_loss/pass_count),
              "Accuracy: {:.4f}".format(accuracy))

        running_loss = 0
```

```
Epoch: 5/20
Training Loss: 0.3878
Validation Loss: 0.2613   Accuracy: 0.9444

Epoch: 6/20
Training Loss: 0.3832
Validation Loss: 0.2601   Accuracy: 0.9444

Epoch: 7/20
Training Loss: 0.3823
Validation Loss: 0.2659   Accuracy: 0.9444

Epoch: 8/20
Training Loss: 0.3834
Validation Loss: 0.2686   Accuracy: 0.9444

Epoch: 9/20
Training Loss: 0.3927
Validation Loss: 0.2751   Accuracy: 0.9444

Epoch: 10/20
Training Loss: 0.3715
Validation Loss: 0.2628   Accuracy: 0.9444

Epoch: 11/20
Training Loss: 0.3867
Validation Loss: 0.2659   Accuracy: 0.9444

Epoch: 12/20
Training Loss: 0.3686
Validation Loss: 0.2578   Accuracy: 0.9444

Epoch: 13/20
Training Loss: 0.3594
Validation Loss: 0.2633   Accuracy: 0.9444
```

Figure 25 Implementation code for determining validation loss and validation accuracy

3.6 Saving and Loading the Model

Now that the neural network has been trained, the model should be saved so as it could be loaded in future for making predictions. In order to completely rebuild the model later on and use it for inference and be able to keep training it, there are other things that eventually have to be saved such as the mapping of classes to indices, the number of epochs, the optimizer state etc. The code for saving and loading the trained models is presented in Figure 26 and Figure 27, respectively.

```
In [15]: # change the model_name, for saving multiple files

         model_name = 'checkpoint_vgg19_Normed_100ep_20ba_25088_4096_4096.pt'

         myModel.class_to_idx = train_data.class_to_idx

         checkpoint = {'classifier' : classifier,
                       'epochs': epochs,
                       'batch_size': batch_size,
                       'opt_state': optimizer.state_dict(),
                       'class_to_idx': myModel.class_to_idx,
                       'state_dict': myModel.state_dict()}

         with open(model_name, 'wb') as f:
             torch.save(checkpoint, f)
```

Figure 26 Code for saving trained model for inference

```
In [18]: def load_checkpoint(filename):
             checkpoint = torch.load(filename)
             model = models.vgg19(pretrained=True)
             model.classifier = checkpoint['classifier']
             model.epochs = checkpoint['epochs']
             model.batch_size = checkpoint['batch_size']
             model.load_state_dict(checkpoint['state_dict'])
             model.class_to_idx = checkpoint['class_to_idx']
             optimizer.load_state_dict(checkpoint['opt_state'])

             return model, optimizer
```

Figure 27 Code for loading saved model for inference

3.7 Inference for Classification and Input Image Preprocessing

Now that the model can be saved and loaded, a function that would use the trained network for inference and pass an image into the network so as to predict the class of the flower in the image, should be developed. Before doing so, the input image should be preprocessed such that it can be used by the

network. As presented in Figure 28, library PIL is initially employed within process_image function to load and convert to 'RGB' the image and so as to subsequently preprocess the image in the same manner used for training so it can be used as input for the model.

Firstly, the image is resized (using resize method) making the shortest side 255 pixels and keeping the aspect ratio and then is cropped out the center 224x224 portion of the image. Like before training process, the network expects the images to be normalized in a specific way ([0.485, 0.456, 0.406] and for the standard deviations [0.229, 0.224, 0.225]) which, in other words, is subtraction of the means from each color channel, and division by the standard deviation. Due to the fact that the color channels of images are typically encoded as integers 0-255, these values are converted to floats from 0-1 like the way the model expects as input, which is done by utilizing Numpy Array. The code and the results of the image processing are illustrated in Figure 29. The resulted image is surpassed into another function (developed in the coming subchapter), called predict, that would take as input parameters an image and a trained model, and would returns the top K most likely classes along with the probabilities.

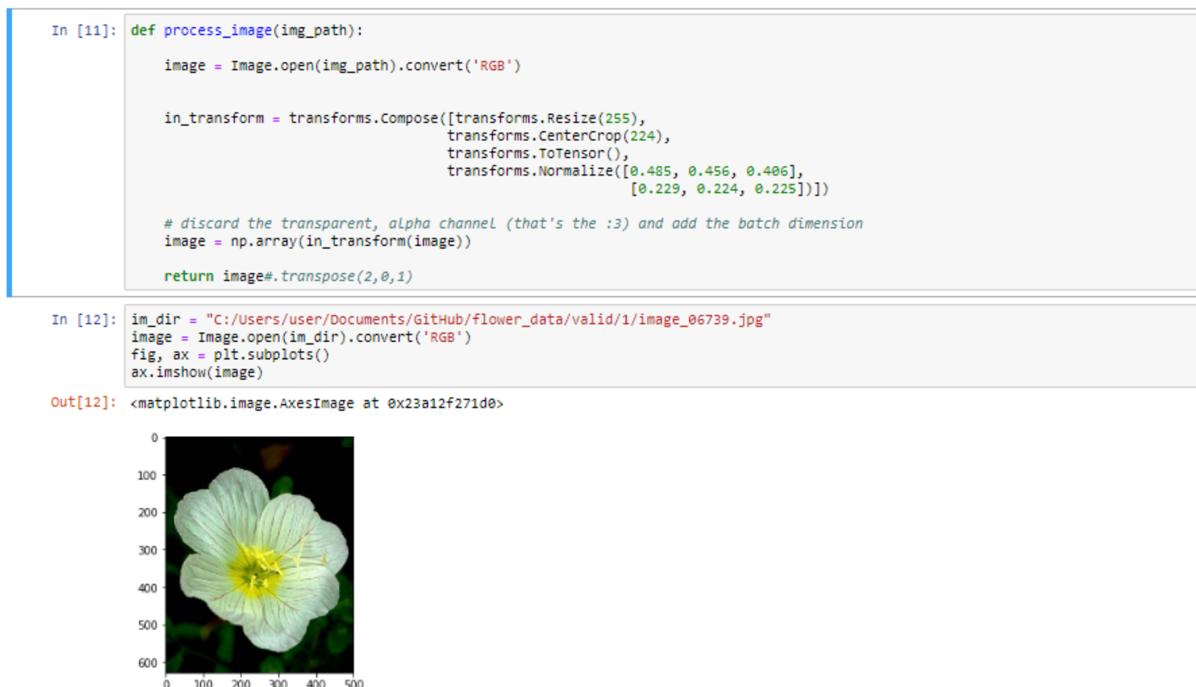

Figure 28 Image processing for input to the prediction function

```
In [168]: tensor_image = process_image(im_dir)
```

```
In [169]: def imshow(image, ax=None, title=None):
              """Imshow for Tensor."""
              if ax is None:
                  fig, ax = plt.subplots()

              # PyTorch tensors assume the color channel is the first dimension
              # but matplotlib assumes is the third dimension
              image = image.transpose((1, 2, 0))

              # Undo preprocessing
              mean = np.array([0.485, 0.456, 0.406])
              std = np.array([0.229, 0.224, 0.225])
              image = std * image + mean

              # Image needs to be clipped between 0 and 1 or it looks like noise when displayed
              image = np.clip(image, 0, 1)

              ax.imshow(image)

              return ax
```

```
In [170]: imshow(tensor_image)
Out[170]: <matplotlib.axes._subplots.AxesSubplot at 0x2b38051b898>
```

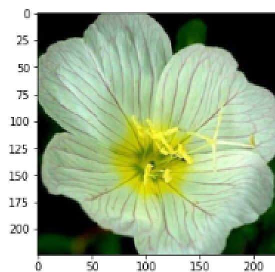

Figure 29 Resulted image for input in the prediction function, after processing

3.8 Flower Class Prediction

Now that the images have been converted to the correct format, it's time to write a function for making predictions with the model. A common practice is to predict the top 7 or so (usually called top-K) most probable classes. To get the top K largest values in a tensor the method x.topk(k) is used which returns both the highest k probabilities and the indices of those probabilities corresponding to the classes. Then those indices are converted to the actual class labels using class_to_idx assuring the dictionary inversion so as to get a mapping from index to class as well. The implementation code of the prediction function and the results is presented in Figure 30 and Figure 31, respectively.

```python
In [20]: def predict(image_path, model, topk=7):
             ''' Predict the class (or classes) of an image using a trained deep learning model.
             '''

             # Predict the class from an image file
             # move the model to cuda
             cuda = torch.cuda.is_available()
             if cuda:
                 # Move model parameters to the GPU
                 model.cuda()
                 print("Number of GPUs:", torch.cuda.device_count())
                 print("Device name:", torch.cuda.get_device_name(torch.cuda.device_count()-1))
             else:
                 model.cpu()
                 print("We go for CPU")

             # turn off dropout
             model.eval()

             # The image
             image = process_image(image_path)

             # tranfer to tensor
             image = torch.from_numpy(np.array([image])).float()

             # The image becomes the input
             image = Variable(image)
             if cuda:
                 image = image.cuda()

             output = model.forward(image)

             probabilities = torch.exp(output).data

             # getting the topk (=7) probabilites and indexes
             # 0 -> probabilities
             # 1 -> index
             prob = torch.topk(probabilities, topk)[0].tolist()[0] # probabilities
             index = torch.topk(probabilities, topk)[1].tolist()[0] # index

             ind = []
             for i in range(len(model.class_to_idx.items())):
                 ind.append(list(model.class_to_idx.items())[i][0])

             # transfer index to label
             label = []
             for i in range(7):
                 label.append(ind[index[i]])

             return prob, label
```

Figure 30 Implementation code of the prediction function

```
In [21]: img = random.choice(os.listdir(data_dir + '/valid/3/'))
         img_path = data_dir + '/valid/3/' + img

         with  Image.open(img_path) as image:
             plt.imshow(image)

         prob, classes = predict(img_path, myModel)
         print(prob)
         print(classes)
         print([cat_to_name[x] for x in classes])
```

```
Number of GPUs: 1
Device name: GeForce GTX 1070 Ti
[0.537497341632843, 0.1812904179096222, 0.0789879634976387, 0.04332459345459938, 0.04283355176448822, 0.023261073976755142, 0.0
18938207998871803]
['3', '4', '94', '11', '36', '32', '43']
['canterbury bells', 'sweet pea', 'foxglove', 'snapdragon', 'ruby-lipped cattleya', 'garden phlox', 'sword lily']
```

Figure 31 Prediction results

In order to check whether the predictions of the trained model make sense, despite the fact that the validation loss and accuracy is respectively low and high, it's always good to check that there are not any rational mistakes. To do so, the probabilities for the top 7 classes are plotted, with the use of matplotlib library, as a bar graph along with the input image, as presented in Figure 32. The conversion from the class integer encoding to the actual flower names took place with the cat_to_name.json file while the imshow function defined before is utilized to illustrate a PyTorch tensor as an image.

In [22]:
```
# Display an image along with the top 7 classes
prob, classes = predict(img_path, myModel)
max_index = np.argmax(prob)
max_probability = prob[max_index]
label = classes[max_index]

fig = plt.figure(figsize=(6,6))
ax1 = plt.subplot2grid((15,9), (0,0), colspan=9, rowspan=9)
ax2 = plt.subplot2grid((15,9), (9,2), colspan=5, rowspan=5)

image = Image.open(img_path)
ax1.axis('off')
ax1.set_title(cat_to_name[label])
ax1.imshow(image)

labels = []
for cl in classes:
    labels.append(cat_to_name[cl])

y_pos = np.arange(7)
ax2.set_yticks(y_pos)
ax2.set_yticklabels(labels)
ax2.set_xlabel('Probability')
ax2.invert_yaxis()
ax2.barh(y_pos, prob, xerr=0, align='center', color='blue')

plt.show()
```

```
Number of GPUs: 1
Device name: GeForce GTX 1070 Ti
```

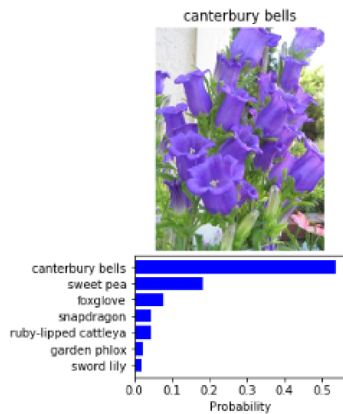

Figure 32 The probabilities for the top 7 classes and the corresponding image for prediction

4 Conclusions

The present study comprises an endeavor of how to process multidimensional data, in particular images, and build an image classifier able to classify flowers and automatically determine the species they belong. Within this framework, it is highlighted the importance of machine learning algorithms for processing multidimensional data and provide a guide in order to decide which type of algorithm is optimum to use for the data analysis. The employed techniques and the methodology followed in this study have led to promising prediction results. In particular, there has been developed and trained an image classifier for recognizing 102 distinct species of flowers utilizing a certain type of machine learning algorithm called Convolutional Neural Networks, resulting in very good performance on a variety of experiments, achieving up to 94% validation accuracy.

The overall methodology followed is summarized in choosing and loading a pretrained CNN model, train and update by back propagation only the weights of the last part of the Neural Network (Fully Connected Layer – Classifier) and validate the overall model. Then, the model fine tuning and feature extraction took place in order to optimize model prediction performance. Based on that, despite the high validation accuracy results there are potentials of improving prediction model performance. Limitations of GPU did not allow to further evaluate certain model learning hyperparameters (batch size) while more learning rate, drop out and layer inputs values could have been tested. Additionally, training the VGG19 model from scratch could have led to better performance but it would need very long times of training and probably broadening of the available dataset. Moreover, the modification of the model architecture itself could have an impact on model's performance. For instance, modify the number of the fully connected layers within classifier or add/remove convolutional layers. Last but not least, fine tuning the hyperparameters of the VGG19 (number of feature filters, kernel size, stride, padding, max pooling etc.) would have a positive impact on model performance.

References

[1] C. D. Ruberto and L. Putzu, "A Fast Leaf Recognition Algorithm based on SVM Classifier and High Dimensional Feature Vector," in *International Conference on Computer Vision Theory and Applications (VISAPP)*, 2015.

[2] J. Gu, Z. Wang, J. Kuen, L. Ma, A. Shahroudy, B. Shuai, T. Liu, X. Wang, G. Wang, J. Cai and T. Chen, , "Recent advances in convolutional neural networks," *Pattern Recognition,* pp. 354-377, 2018.

[3] Clifton Christopher, *Definition of Data Mining,* 2010.

[4] Hastie Trevor, Tibshirani Robert, Friedman Jerome, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction,* 2009.

[5] Fayyad Usama, Piatetsky-Shapiro Gregory, Padhraic Smyth, *From Data Mining to Knowledge Discovery in Databases,* American Association for Artificial Intelligence, 1996.

[6] Hui Li, "The SAS Data Science Blog - Which machine learning algorithm should I use?," SAS THE POWER TO KNOW , 12 April 2017. [Online]. Available: https://blogs.sas.com/content/subconsciousmusings/2017/04/12/machine-learning-algorithm-use/.

[7] Shai Shalev-Shwartz, UNDERSTANDING MACHINE LEARNING - From Theory to Algorithms, Cambridge University Press, 2014.

[8] M. Hajek, NEURAL NETWORKS, 2005.

[9] McCulloch Warren, Walter Pitts, "A Logical Calculus of Ideas Immanent in Nervous Activity," *Bulletin of Mathematical Biophysics,* vol. 5, no. 4, pp. 115-133, 1943.

[10] Hebb, Donald, The Organization of Behavior., New York: Wiley, 1949.

[11] Rosenblatt, F., "The Perceptron: A Probabilistic Model For Information Storage And Organization In The Brain," *Psychological Review,* vol. 65, no. 6, p. 386–408, 1958.

[12] Minsky, Marvin; Papert, Seymour, Perceptrons: An Introduction to Computational Geometry., MIT Press., 1969.

[13] Werbos, P.J., Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences., 1975.

[14] Aphex34, "Typical CNN architecture," 16 12 2015. [Online]. Available: https://commons.wikimedia.org/wiki/File:Typical_cnn.png .

[15] K. Fukushima, Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position, Biological cybernetics, 1980.

[16] B. B. Le Cun, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard,and L. D. Jackel, Handwritten digit recognition with a backpropagation network, NIPS. Citeseer, 1990.

[17] A. Krizhevsky, I. Sutskever and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *NIPS,* pp. 1106-1114, 2012.

[18] I. J. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville and Y. Bengio, "Maxout Networks," *JMLR WCP,* vol. 28, no. 3, pp. 1319-1327, 2013.

[19] Nilsback, M-E. and Zisserman, A., "Automated flower classification over a large number of classes," in *Proceedings of the Indian Conference on Computer Vision, Graphics and Image Processing (2008)*, 2018.

[20] University of Oxford - Department of Engineering Science, University of Oxford, "102 Category Flower Dataset," Visual Geometry Group, 20 02 2009. [Online].

[21] ImageNet., "ImageNet," [Online]. Available: http://www.image-net.org.

[22] Russakovsky, O., Deng, J., Su, H., et al. , "ImageNet Large Scale Visual Recognition Challenge.," *International Journal of Computer Vision (IJCV) ,* vol. 115, no. 3, p. 211–252, 2015.

[23] Peterson Leif E., "Scholarpedia," Center for Biostatistics, The Methodist Hospital Research Institute, 2009. [Online]. Available: http://www.scholarpedia.org/article/K-nearest_neighbor.

[24] Thirumuruganathan Saravanan, "A Detailed Introduction to k-Nearest Neighbor Algorithm," WordPress, 17 May 2010. [Online]. Available: http://saravananthirumuruganathan.wordpress.com/2010/05/17/a-detailed-introduction-to-k-nearest-neighbor-knn-algorithm/.

[25] Andrews W K Donald, *Asymptotic Optimality of Generalized CL, Cross-validation, and Generalized Cross-validation in Regression with Heteroskedastic Errors,* vol. 47, 1991.

[26] Agrawal Rakesh, Ramakrishnan Srikant, *Fast Algorithms for Mining Association Rules,* 650 Harry Road, San Jose, CA 95120: IBM Almaden Research Center, 1994.

[27] Belur V Dasarathy, *Nearest Neighbor (NN) Norms: NN Pattern Classification Techniques.,* 1991.

[28] Li K C, Cheng Ching-Shui, *Optimality criteria in survey sampling,* vol. 74, 1987.

[29] P. T. (2009), Multidimensional Modeling. In: LIU L., ÖZSU M.T. (eds) Encyclopedia of Database Systems., Boston, MA: Springer, 2009.

[30] P. Pai, "Data Augmentation Techniques in CNN using Tensorflow," 25 10 2017. [Online]. Available: https://medium.com/ymedialabs-innovation/data-augmentation-techniques-in-cnn-using-tensorflow-371ae43d5be9.