GET Lab

Universität Paderborn
Fakultät EIM
Institut für Elektrotechnik und Informationstechnik

Deep Learning-based Scale Estimation for Local Image Features

Master's Thesis
for the Master's Program in Electrical Systems Engineering
submitted by

Dheeraj Rajashekar Poolavaram

supervised by

Dipl.-Ing. Markus Henning

submitted to

Prof. Dr.-Ing. Bärbel Mertsching

Paderborn, in March 2023

Abstract

The objective of this work was to develop a deep learning-based framework to estimate the characteristic scales of local image features. The characteristic scale describes the size of a feature and defines the region used to form the corresponding feature vector. Feature vectors can be used as a basis for feature matching, object detection, object tracking, and other applications. Some of the most important feature detection algorithms use computationally expensive and complex scale-space analysis schemes. The development of the framework involves two main steps: creating datasets with local image features and characteristic scales and selecting, implementing, and testing appropriate Convolutional Neural Network (CNN) architectures. The PASCAL VOC dataset, which contains approximately 17,000 real images, has been used to create the dataset, and SIFT and SURF have been used to extract local features and their characteristic scales. The developed framework estimates the characteristic scales in a scale-invariant manner using a fixed-sized input. Various network architectures have been investigated, and the Mod-ResNet34 architecture has been found to be the most effective, achieving an overall accuracy of 85.89 % and 88.19 % on test datasets created using SIFT and SURF, respectively. Furthermore, the test datasets were subdivided into three subsets based on different characteristic scales for systematic analysis. It was found that the network performs well in general but struggles with small-scaled keypoints. A systematic analysis has been carried out, and the findings have been presented in this work.

Contents

1 Introduction

In computer vision and digital image processing, feature detection and description are fundamental pre-processing steps for many applications (e.g. object recognition, robot navigation, and motion-based segmentation). Image features can be categorized as global or local features. Global features represent an object by a single feature vector, whereas local features represent an object by a set of keypoints and their surrounding regions. Typical local features are corners, blobs, edges, and junctions. In comparison, local features operate better with partial occlusion, deformable objects, and viewpoint changes. This is because small, distinctive regions have a relatively lower chance of being affected [GL11]. The most common method to detect local features is Scale-Invariant Feature Transform (SIFT) [Low04], which uses Difference-of-Gaussians (DoG) to construct a scale-space pyramid. The keypoints are detected by analyzing the DoG images in scale-space for local extrema. Detecting keypoints in scale-space allows SIFT to detect keypoints in a scale and rotation-invariant manner. To speed up the method, several new detectors (e.g. SURF, KAZE, AKAZE, ORB, BRISK) have been developed [TS18]. Assigning the characteristic scale to each keypoint is a key principle of these detectors.

The characteristic scale of a keypoint is a measure of its size relative to the image scale. It represents the scale at which the keypoint is most stable and can be reliably detected and described. An example of using characteristic scales is in object detection, where features at different scales may correspond to different parts of an object, such as a small scale for detecting the edges of an object and a large scale for detecting regions corresponding to an object. Accurate characteristic scale estimation can also be used to improve the robustness of the feature detection and description to changes in image scale, rotation, and viewpoint. In other words, by estimating the characteristic scale of each keypoint, the feature detection algorithm can focus on detecting features at the most stable scale, which improves the robustness of the keypoint detection and description. Since the features in an image exist over specific finite ranges of scale, the characteristic scales are helpful to describe the structure of an image [Lin98]. The scales also adapt to the local image structure in a scale-invariant manner enabling the local structures between different images to be matched

efficiently [GL11]. Overall, characteristic scale information can improve the robustness and accuracy of many computer vision algorithms and helps better to understand the meaning and context of local features [MS04].

1.1 Motivation

Traditional methods for detecting and describing local features, such as SIFT and SURF, use computationally expensive and complex scale-space analysis schemes. These methods typically detect local features by utilizing a computationally intensive approach, for example, determining gradient magnitude and orientation in SIFT at multiple scales and then selecting the scale at which the feature attains a local extremum. These methods are overall efficient in detecting keypoints but inefficient in estimating characteristic scales and can be sensitive to the choice of parameters.

In contrast, deep learning-based methods for estimating characteristic scales, such as Learned-Invariant Feature Transform (LIFT), Local Feature Network (LF-Net), and Key.Net, use Convolutional Neural Networks (CNNs). They can automatically learn the characteristic scales of the features in an image by training on a large dataset of images. These methods typically involve convolutional layers and pooling layers, which can learn hierarchical representations of an image and, in turn, learn the scale of the features. These methods are computationally more intensive than traditional methods. However, they can learn to detect robust features invariant to geometric and photometric transformations and generalize well to unseen images [JP20].

This work aims to replace the scale-space schemes with a deep learning-based framework, which can be used for a range of different feature detectors. The implementation as a framework allows systematic tests of different combinations of detectors, scale estimation schemes, and CNN architectures. One example of a complex scale space scheme is Curvature Scale Space (CSS) analysis in [HM20] used to detect contour features. Furthermore, the method requires accurate edge images, which are often difficult to determine, in addition to other challenging steps. This framework could be used as a substitute for the CSS analysis to assign characteristic scales to different local image features in real images.

1.2 Objectives

The objective of this work is to develop a deep learning-based framework to estimate the characteristic scales of local image features. The input to the framework should be real images together with keypoints, and the output should be the characteristic scales of the keypoints.

The first step is to create a dataset consisting of the following information for each keypoint: the position in the image plane (coordinates), characteristic scale, characteristic orientation, and the response specifying the keypoint strength. Information other than the position and the corresponding characteristic scale is only stored for the sake of completeness. The dataset will be split into appropriate training, validation, and testing subsets. The images for the dataset will be obtained from the PASCAL Visual Object Classes [EVGW+], which contains approximately 17,000 real images for vision and machine learning applications.

The second step is selecting, implementing, and testing appropriate CNN architectures from existing deep-learning-based keypoint detection and scale estimation methods.

Another objective of this work is to design and implement an accuracy metric that allows a systematic evaluation of scale estimation.

1.3 Thesis Structure

This work is structured as follows: Chapter 2 describes the relevant fundamentals to comprehend this work. For this purpose, chapter 2.1 gives an overview of methods for object recognition. In chapter 2.2, different scale-invariant feature detectors are discussed. Chapter 2.3 deals with neural networks, particularly the concepts of deep learning and convolutional neural networks investigated in this work. Chapter 3 presents the literature review results. Deep learning-based feature detectors such as LIFT, LF-Net, and Key.Net are introduced and discussed. Chapter 4 discusses the details of development and implementation. Chapter 4.1, describes the dataset created and used in this work. In chapter 4.2, an overview of the implemented framework is discussed. Chapter 4.3 describes the network architectures employed. Chapter 5 elaborates on the experiments conducted and the results obtained. Chapter 6 summarizes this work with an outlook on future work.

2 Fundamentals

This chapter describes the fundamental details required to understand this work. Chapter 2.1 provides an overview of object recognition, specifically the basics of keypoint detection. Chapter 2.2 deals with different traditional scale-invariant feature detectors SIFT [Low04] and SURF [BTVG06]. Chapter 2.3 introduces concepts in deep learning, particularly interesting for this work, such as Artificial Neural Networks (ANNs), Convolutional Neural Networks (CNNs), and different layers of building a CNN.

2.1 Object Recognition

Object recognition is one of the fundamental applications of image processing and computer vision that has been a focus of intensive investigation for several decades. With knowledge of how particular structures appear in an image, it is possible to determine which objects are present and where they are located. For example, suppose it is known what a car looks like. This knowledge can detect cars in images or videos by applying computer vision techniques such as feature detection, feature extraction, and object recognition algorithms. These algorithms can analyze the image and identify specific patterns and structures that match the known features of a particular object. Apart from that, each application has specific requirements and limitations. This characteristic has given rise to a wide variety of algorithms. The advancements in the field of Artificial Intelligence (AI) have enabled a wide range of applications in the field of object recognition. Examples include surveillance systems, autonomous driving, security, medical imaging, and real-time video analytics [CW19].

Researchers studying the field of vision typically divide recognition into two categories: the specific case and the general category case. For example, Albert Einstein's face, the Cologne Cathedral, or a specific book cover are examples of instances of a specific case. At the category level, it is to identify several examples of a generic category belonging to the same class, such as flowers, animals, or automobiles [GL11]. Figure 2.1 shows examples of specific and generic objects.

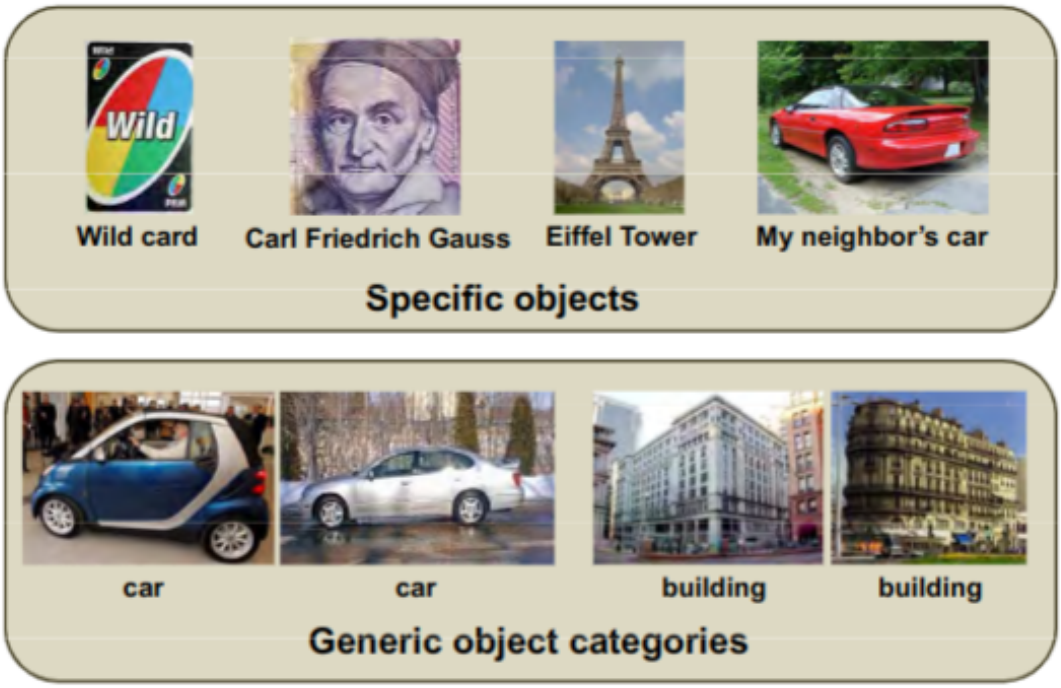


Figure 2.1: Instances of a particular object or scene (top) and instances at a basic level (bottom) from [GL11]

Based on developments in the computer vision literature over the last decade, there are conventional algorithms for both types of recognition. A matching and geometric verification approach is commonly used for specific object recognition. In contrast, a generic case includes a statistical model of appearance or shape learned from a set of examples. Owing to the advancements in deep learning, a CNN can be used to learn visual characteristics for training images to detect or localize objects in images. A CNN can be used in various ways and, in particular, form abstract representations independently from the raw input data without the need for explicit feature engineering or manual extraction of features. Depending on the task, an object recognition system is provided with training data and corresponding target labels using supervised learning methods. Figure 2.2 shows three possible object recognition tasks. Image classification categorizes an image based on the object(s) included in the image. A bounding-box detection further spatially localizes and roughly draws rectangular boxes around the detected objects. For segmentation, a pixel-level map is estimated to separate the foreground and background and localize an object precisely [GL11].

However, there are several challenges when learning visual representations and comparing different images of objects from the same category. They include occlusions, camera viewpoint, varying light conditions, the position of the object, scaling effects, and cluttered backgrounds, as shown in Figure 2.3. Therefore, the recognition methods need to be robust to different

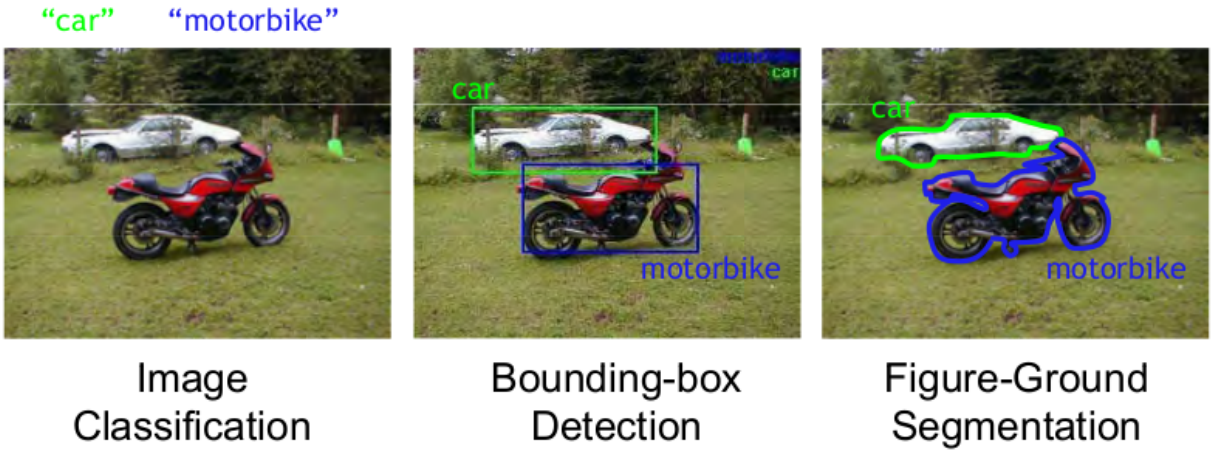


Figure 2.2: Different object recognition tasks from [GL11]

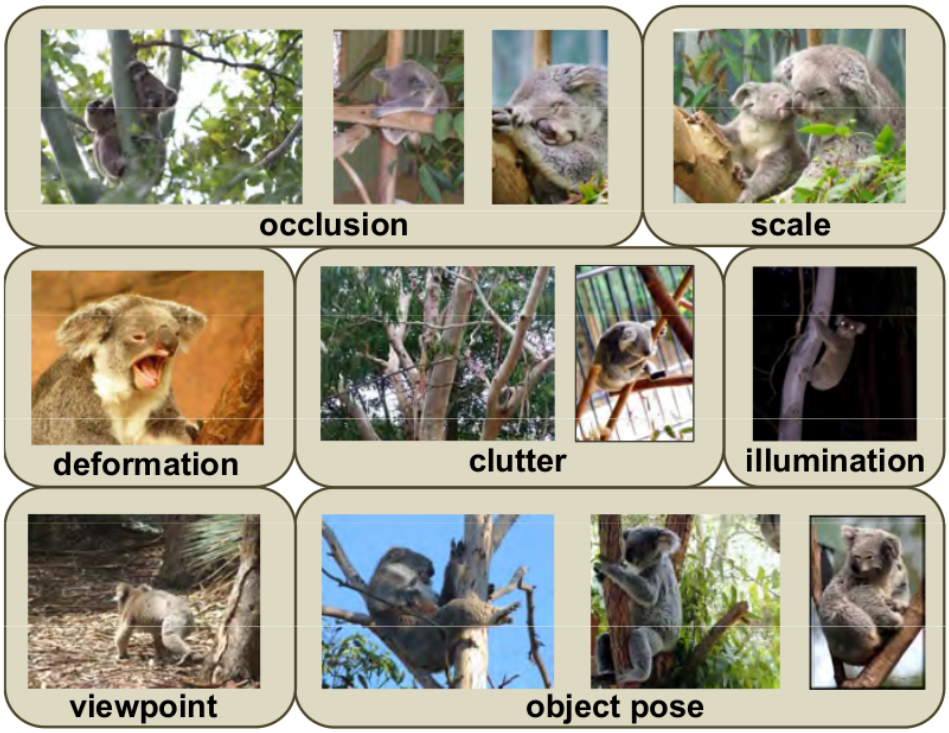


Figure 2.3: Examples of different challenges in object recognition from [GL11]

image conditions. In addition to robustness, computational complexity and scalability are other factors that object recognition systems must consider for real-time applications. Using highly efficient algorithms is essential to address complex and high-dimensional image representations, review large image libraries, or expand recognition to thousands of categories. When creating data required for training a recognition system, scalability issues arise since well-labeled, most informative image examples are expensive to collect. Therefore, current

techniques must assess the advantages provided to the learning process against the amount of expensive manual supervision an algorithm requires [GL11]. The following sections discuss commonly used methodologies for detecting objects (or interest regions) as keypoints.

2.1.1 Image Features

Regions of interest in an image can be categorized as global or local features. Global features represent an object by a single feature vector, whereas local features represent an object by a set of keypoints and their surrounding regions. Global feature techniques rely on comparing whole images or image windows. Figure 2.4 shows global descriptions as ordered and unordered sets of intensities. Depending on the object, the distribution of intensities generally follows a specific pattern [GL11]. Such techniques work well for learning global object structure but struggle with partial occlusion, strong viewpoint changes, and deformable objects.

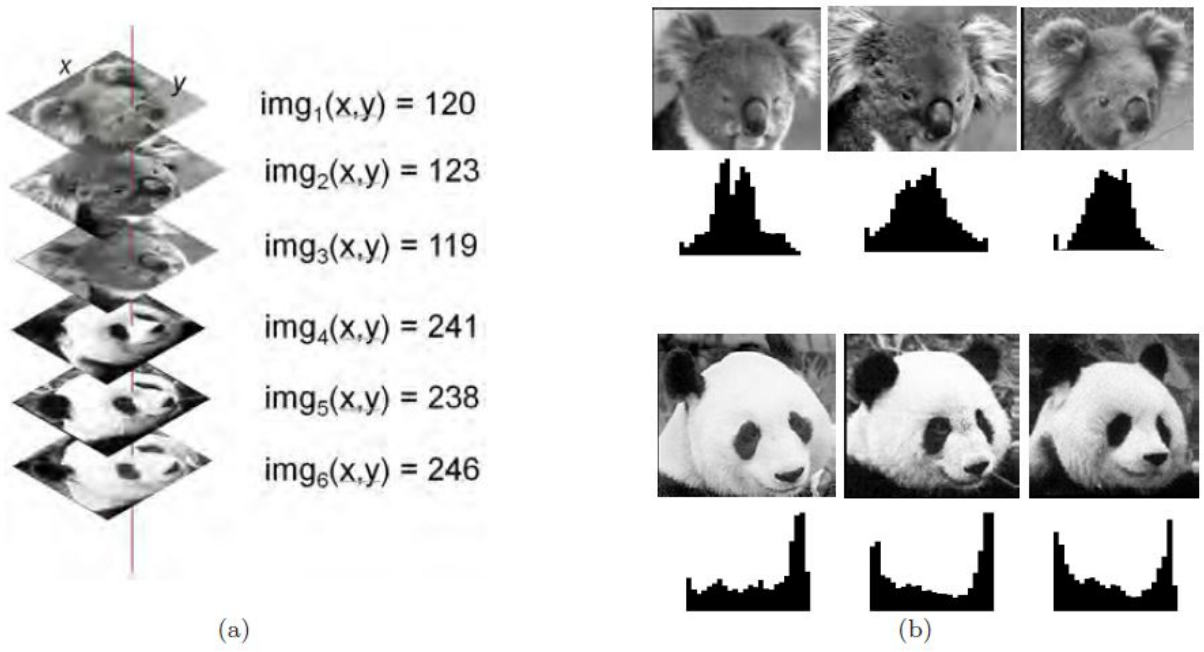


Figure 2.4: Examples of global description of images unordered (a) ordered (b) sets of intensities from [GL11]

Local features are specific, distinctive image structures within an image in the form of interest points, corners, edges or contours, and larger features or regions such as blobs [Kri16, GW18]. In comparison, local features operate better with partial occlusion, deformable objects, and viewpoint changes because smaller, distinctive regions have a relatively lower chance of being

affected [Tre10]. Figure 2.5 shows a local feature-based recognition approach. Feature extraction is performed in two steps. First, the distinctive keypoints of an object are detected, and then regions around these points are described with feature vectors. These feature vectors are matched to determine the correspondence [Low04]. These features are extracted from two images with different geometric configurations. These candidate matches are then used to determine if the features appear in a consistent geometric configuration. The extraction should be precise and repeatable to determine the same features for different images of the same object, particularly concerning rotation, translation, and scaling effects. At the same time, the features should be distinct so that they can be uniquely assigned to corresponding objects [GL11].

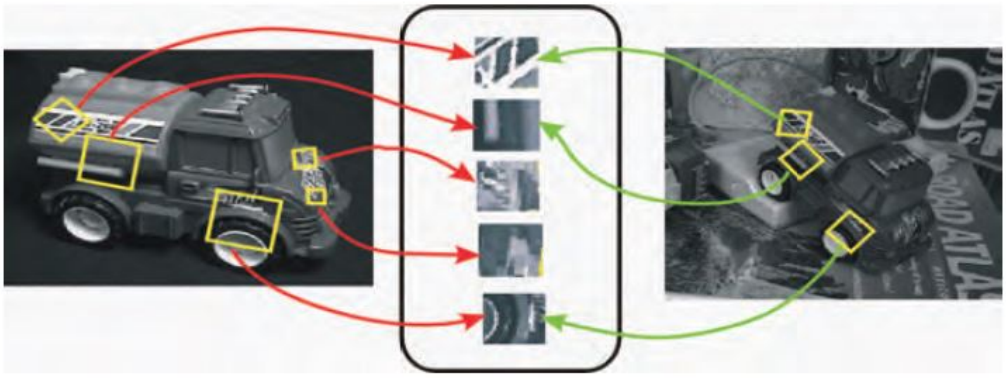


Figure 2.5: Visualization of the local feature-based recognition procedure from [Low04]

2.2 Keypoint Detection

The following section describes the detection and extraction of keypoints in detail. Keypoints should be detected in a scale- and rotation-invariant manner and invariant to illumination and viewpoint changes. Detectors used earlier in the literature focused on detecting signal changes in two directions by considering a subset of points. An example is the Hessian detector based on a matrix of second derivatives called the Hessian. It detects image locations with strong derivatives in two orthogonal directions. Another well-known method is the Harris detector uses a matrix of Gaussian-weighted first derivatives in a window around a point. In general, Harris points are more suitable to precisely localize corners, whereas Hessian provides a denser description with more keypoints of an object. However, both detectors are repeatable only for small-scale changes since they rely on fixed Gaussian derivatives at a fixed scale σ. For a given set of test images, the detected keypoints are susceptible if

the image scales vary significantly [GL11]. It is therefore, necessary to detect keypoints that can be extracted reliably under varying scales. In particular, two such commonly used scale-invariant keypoint detectors, Scale-Invariant Feature Transform (SIFT) and Speeded-Up Robust Features (SURF), will be discussed in this section. This work uses these keypoint detectors to create the data to train different CNN architectures.

2.2.1 Characteristic Scale and Scale-space Representation

The characteristic scale of a local image feature is a measure of its size relative to the image scale. The term characteristic primarily refers to the fact that the chosen scale estimates the characteristic size of the related image structures, similar to the concept of characteristic length used in physics [Lin98].

As described in [Lin98], the objective is to choose a region for which a specific scale-dependent signature function reaches an extremum over scales. The characteristic scale corresponds to the local extremum of the responses. Given a keypoint position, the signature function is evaluated on the keypoint neighbourhood, and the resulting value is plotted as a function of the scale. For two images, if two keypoints correspond to the same structure, their signature functions will have similar shapes. The associated neighbourhood sizes can be calculated by independently looking for the scale-space extrema of the signature function in both images. Since the signature function captures attributes of the immediate image neighbourhood at a given radius, it should have a comparable qualitative form if the two keypoints are centred on corresponding image structures. The only difference is that due to the scale factor between the two images, one function form will be compressed or expanded compared to the other, as shown in Figure 2.6. It shows an example of an image captured at two different focal lengths, and the characteristic scale corresponds to the local extremum of the responses. It can be shown that the scale is related to the structure and not the resolution at which the structure is represented. Thus, matching neighbourhood sizes can be found by looking separately for the extrema of the signature function in both images. The extrema of the signature function, σ, is called the characteristic scale. If the corresponding extrema σ and $\sigma^{'}$ are identified in both cases, the scaling factor between the two images is $\frac{\sigma^{'}}{\sigma}$ [GL11].

A scale space of the responses can be constructed using a local kernel with varying scale parameter σ. It has been shown that the Gaussian kernel $G(x, \sigma)$ and its derivatives are the only image operator that satisfies all the necessary conditions for the scale-space construction [Lin94, Lin98]. A 1D Gaussian filter with a varying scaling parameter σ is given in Equation 2.1:

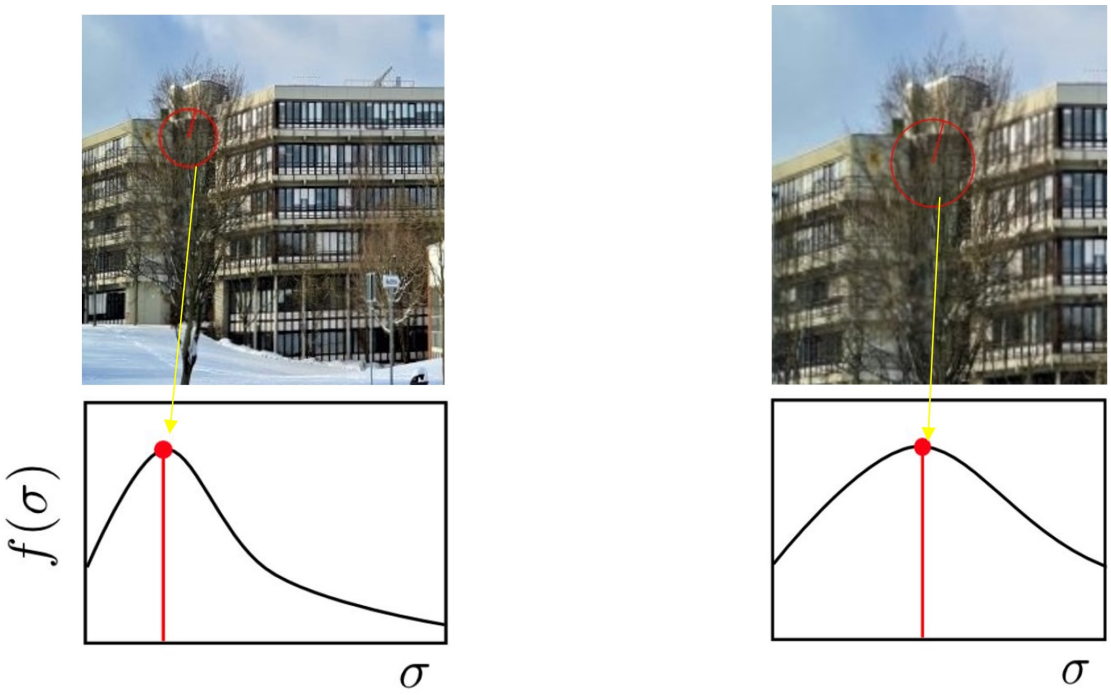


Figure 2.6: Characteristic scale comparison of images captured at different focal lengths. Adapted from [MS04]

$$G(x,\sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}} \tag{2.1}$$

Similarly, a 2D Gaussian filter is shown in Equation 2.2:

$$G(x,y,\sigma) = \frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)/2\sigma^2}. \tag{2.2}$$

Building on that, a detector for blob-like structures has been proposed in [Low99] that searches for scale-space extrema of a scale-normalized Laplacian-of-Gaussian (LoG).

$$L(x,\sigma) = \sigma^2(I_{xx}(x,\sigma) + I_{yy}(x,\sigma)) \tag{2.3}$$

where $L(x,\sigma)$ is the LoG operator and I_{xx}, I_{yy} are the second order derivatives for each image point.

The LoG filter mask corresponds to a circular center-surround structure, as shown in Figure 2.7, with positive weights in the center region and negative weights in the surrounding ring structure. As a result, it will produce the maximum responses when applied to an image neighbourhood with a similar (nearly circular) blob structure at a corresponding

scale. Circular blob structures can be determined by looking for scale-space extrema of the LoG. It should be noted that the blob centre can also be defined as a repeatable keypoint location for such blobs.

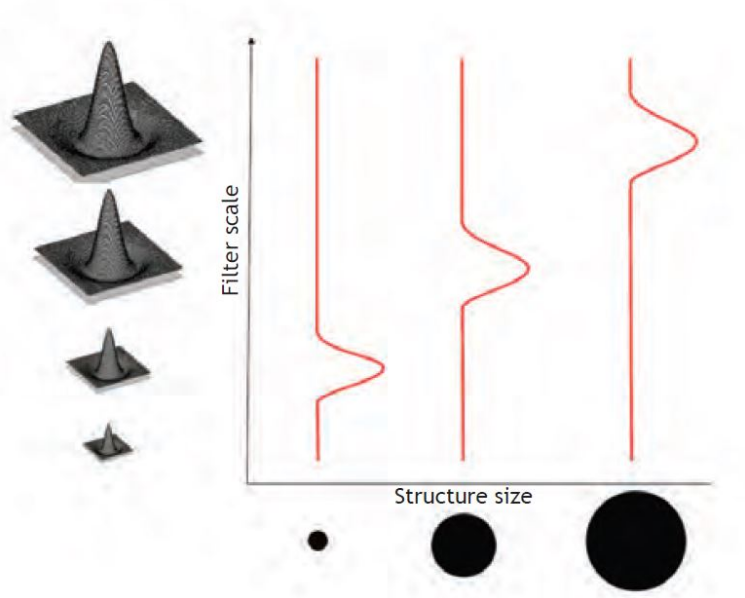


Figure 2.7: The Laplacian-of-Gaussian (LoG) filter has a positive weighted circular centre region surrounded by a negative weighted circular region. The filter response is thus strongest for circular image regions with a radius equal to the filter scale from [GL11]

As shown in Figure 2.8, the LoG can thus be used to determine the characteristic scale for a specific image location and directly detect scale-invariant regions by looking for 3D (location + scale) LoG extrema [MS04].

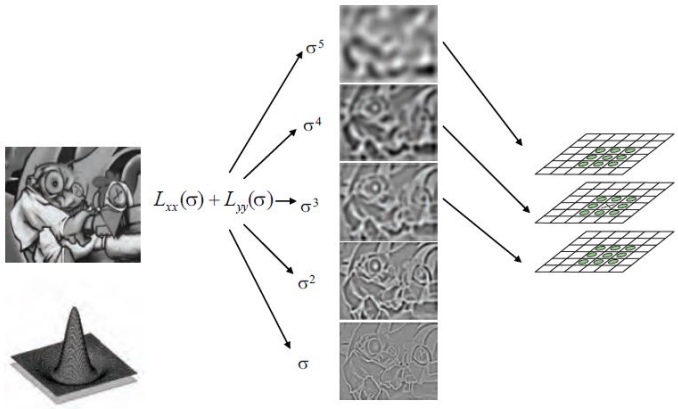


Figure 2.8: Laplacian-of-Gaussian (LoG) filter used to search for 3D scale-space extrema of the LoG function from [MS04]

2.2.2 Scale-Invariant Feature Transform (SIFT)

SIFT is a widely used computer vision algorithm for detecting and describing local features in images. The main idea behind SIFT is to extract and describe local features from an image in a scale, rotation, and illumination invariant manner. View matching for 3D reconstruction, motion tracking and segmentation, robot localization, image panorama assembly, epipolar calibration, and any other applications requiring identifying similar areas between images are possible applications using SIFT descriptors [Low04]. Figure 2.9 shows an example of SIFT keypoints.

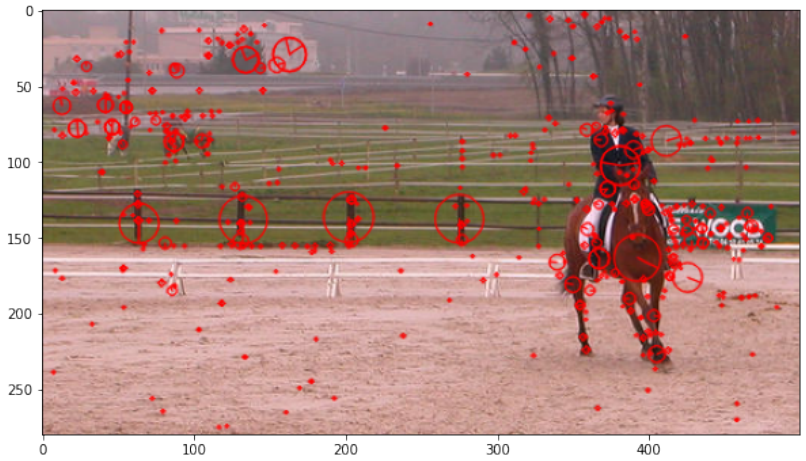


Figure 2.9: SIFT detected keypoints. Adapted from [Low04]

It is necessary to understand the construction of scale-space in SIFT since this work focuses on replacing this scheme. In [Low04], to efficiently detect stable keypoint positions in scale-space, an approximation of LoG detector called Difference-of-Gaussians (DoG) function $D(x, y, \sigma)$ was proposed. This function is computed from obtaining the difference of two consecutive scale levels separated by a constant multiplicative factor k as shown in Equation 2.4:

$$\begin{aligned} D(x, y, \sigma) &= [G(x, y, k\sigma) - G(x, y, \sigma)] * I(x, y) \\ &= L(x, y, k\sigma) - L(x, y, \sigma). \end{aligned} \tag{2.4}$$

The advantage of choosing the DoG function is that the overall computations to construct the scale space can be reduced by image subtraction of the smoothed images from LoG. In addition to that, it has been shown in [Lin94] that the DoG function is a close approximation of the scale-normalized Laplacian-of-Gaussian, which helps to achieve true scale-invariance [Low04].

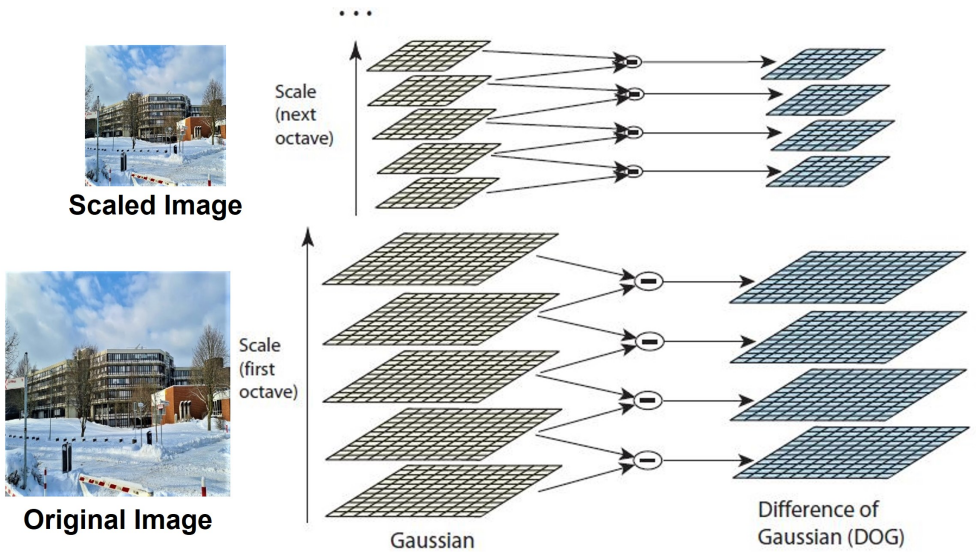


Figure 2.10: Organization of scale-space in octaves. Adapted from [Low04, MS04]

The first stage in SIFT is the scale-space extrema detection. The stage efficiently identifies potential scale- and rotation-invariant keypoints using the DoG function. Figure 2.10 depicts an efficient method for constructing the $D(x, y, \sigma)$ for scale-space construction. The original image is iteratively convolved with a 2D Gaussian function to yield images separated in scale space by a constant factor of k. Each octave of the scale space (i.e., doubling of) is divided into an integer number, s, of intervals. For the final extrema detection to cover an entire octave, $s+3$ images are constructed in the stack of blurred images for each octave. The DoG images on the right are created by subtracting adjacent image scales. After processing an entire octave, the Gaussian image is re-sampled with twice the initial value of σ (two images from the top of the stack) by taking every second pixel in each row and column. While computation is substantially reduced, sampling accuracy is no different than at the start of the previous octave [Low04].

After a scale-space representation is constructed, to detect the local maxima and minima of $D(x, y, \sigma)$, each sample point is compared to its eight neighbours in the present image and nine in the scale above and below as shown in Figure 2.11. It is chosen as an extremum if it is larger or smaller than all its neighbour points. Since most of the sample points will be deleted after the first few comparisons, the cost of this check is appropriate. The initial implementation of this strategy in [Low99] locates keypoints at the position and scale of the centre sample point. However, in [BL02], a method for fitting a 3D quadratic function to the local sample points to estimate the interpolated location of the maximum was proposed. The experiments demonstrated that this significantly improves matching and stability due to the discrete scale levels and the extrema being located somewhere in between. The method

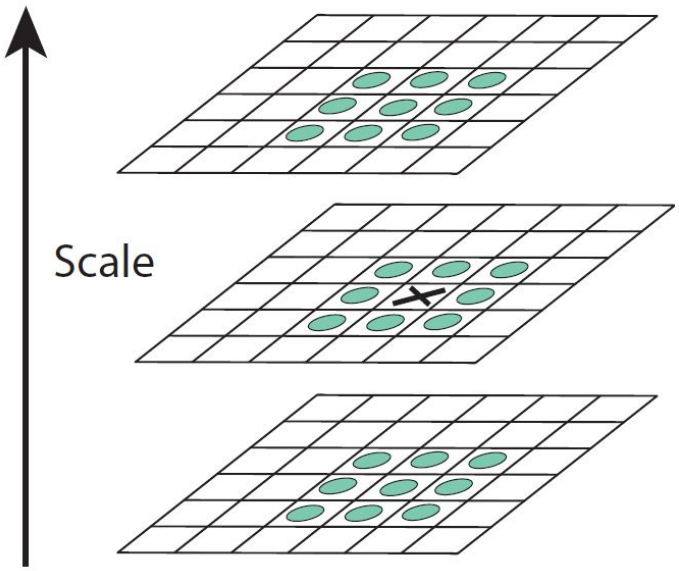


Figure 2.11: Localization of keypoint based on extrema from [Low04]

employs the Taylor expansion (up to quadratic terms) of the scale-space function, $D(x, y, \sigma)$, shifted so that the origin is at the sample point:

$$D(\mathrm{x}) = D + \frac{\partial D^T}{\partial \mathrm{x}}\mathrm{x} + \frac{1}{2}\mathrm{x}^T\frac{\partial^2 D}{\partial \mathrm{x}^2}\mathrm{x} \tag{2.5}$$

where D and its derivatives are computed at the sample point, and $\mathrm{x} = (x, y, \sigma)^T$ is the offset from this point. By taking the derivative of this function with respect to x and setting it to zero, the refined location of the extremum, $\hat{\mathrm{x}}$, is identified as:

$$\hat{\mathrm{x}} = -\frac{\partial^2 D}{\partial \mathrm{x}^2}^{-1}\frac{\partial D}{\partial \mathrm{x}}. \tag{2.6}$$

For rejecting unstable extrema with poor contrast, the function value at the extremum, $D(\mathrm{x})$ is useful, which is given by:

$$D(\hat{\mathrm{x}}) = D + \frac{1}{2}\frac{\partial D^T}{\partial \mathrm{x}}\hat{\mathrm{x}} \tag{2.7}$$

Furthermore, for better stability, keypoints along edges with strong DoG function responses are discarded.

The next step in SIFT is to assign an orientation to each keypoint by choosing a Gaussian-smoothed picture (L) based on the characteristic scale of the keypoint and carrying out all computations in a scale-invariant manner. This step is not relevant to the focus of this work but is essential to understand the overall methodology of SIFT. The gradient magnitude, $m(x, y)$, and orientation, $\theta(x, y)$, is calculated for a given image point, $L(x, y)$. The gradient orientations of the sample points inside the area surrounding the keypoint are used to create an orientation histogram. The 360-degree range of orientations is represented by 36 bins in the orientation histogram. Each sample added to the histogram is assigned a weight based on the gradient's magnitude and a Gaussian-weighted circular window with a σ, 1.5 times the scale of the keypoint.

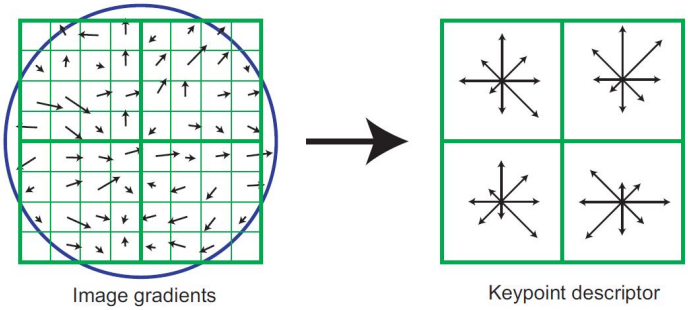


Figure 2.12: Computation of a SIFT keypoint descriptor from [Low04]

The final stage of the algorithm is to compute the SIFT descriptor. With the assigned location, scale, and orientation to keypoints, a 128-dimensional descriptor is computed for the local image region, which is further robust to illumination or 3D viewpoint changes. As shown in Figure 2.12, each image sample point's gradient magnitude and direction in the area surrounding the keypoint are first calculated to build the keypoint description. The magnitude and direction are weighted by a Gaussian window, as indicated by the circle on top. These samples are combined to create orientation histograms that summarize the data over 4×4 sub-regions. The length of each arrow corresponds to the sum of the gradient magnitudes near that direction in the region.

2.2.3 Speeded-Up Robust Features (SURF)

SURF is a feature detection, and the description algorithm introduced as an improvement over SIFT. The primary motivation was to speed up the computation time while maintaining robustness to viewpoint, illumination, and scale changes. SURF uses a similar approach to SIFT by first detecting interest points in the image using the determinant of the Hessian

matrix, a measure of the second-order derivatives of the image intensity. Instead of using a histogram of gradient orientations for computing feature descriptors, it uses a Haar wavelet response in both the x and y directions. This allows for faster computation by using integral images and fewer orientation bins. Additionally, SURF uses a box filter approximation of the Difference of Gaussian (DoG) function for scale space representation, which is faster to compute than the Gaussian kernel used in SIFT [BTVG06].

For a given point $\mathrm{x} = (x, y)$, the Hessian matrix is shown in Equation 2.8:

$$\mathcal{H}(\mathrm{x}, \sigma) = \begin{bmatrix} L_{xx}(\mathrm{x}, \sigma) & L_{xy}(\mathrm{x}, \sigma) \\ L_{yx}(\mathrm{x}, \sigma) & L_{yy}(\mathrm{x}, \sigma) \end{bmatrix} \tag{2.8}$$

where L_{xx}, L_{xy}, L_{yx}, and L_{yy} are derived as in Equation 2.3 and are the resulting terms of a convolution between second-order Gaussian derivatives with an image at point x.

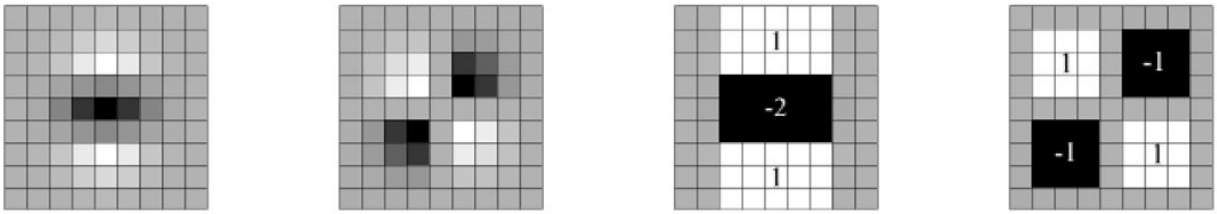


Figure 2.13: Discretized and cropped Gaussian second-order partial derivatives and their approximations with box filters from [BTVG06]

The use of the Gaussian function in scale-space analysis has been widely adopted, as demonstrated by [Low04, Lin98]. However, implementing the Gaussian function in practice requires discretization, cropping of images, and sub-sampling, which can introduce aliasing. As an alternative, SURF uses box filters to approximate second-order Gaussian derivatives, building on the success of Laplacian of Gaussian (LoG) approximations in [Low04]. Integral images can efficiently evaluate these approximations regardless of size, and the performance is comparable to that of discretized, cropped Gaussians [BTVG06]. Figure 2.13 shows the 9×9 box filters used to approximate Gaussian second-order derivatives, with the lowest scale represented by $\sigma = 1.2$. These approximations lead to the determinant of the Hessian matrix, as shown in Equation 2.9:

$$\det(\mathcal{H}_{\text{approx}}) = D_{xx}D_{yy} - (0.9D_{xy})^2 \tag{2.9}$$

where D_{xx}, D_{xy}, and D_{yy} are the box filter approximates of the Gaussian second-order partial derivatives.

Image pyramids are commonly used to implement scale spaces by smoothing an image with a Gaussian filter and sub-sampling it to create a lower-resolution version. However, using box filters and integral images allows filters of any size to be applied directly to the original image and not to the output of a previously filtered layer as in [Low04]. The use of box filters eliminates the need to iteratively apply the same filter to multiple layers of the pyramid. The initial filter size is 9×9, gradually increasing to 15×15, 21×21, and 27×27, with the step size doubling with each additional octave. Similarly, the sampling intervals for extracting interest points are doubled. Interest points are localized in the image over different scales using non-maximum suppression in a $3 \times 3 \times 3$ neighbourhood. Figure 2.14 illustrates the interest points detected by the "Fast-Hessian" detector.

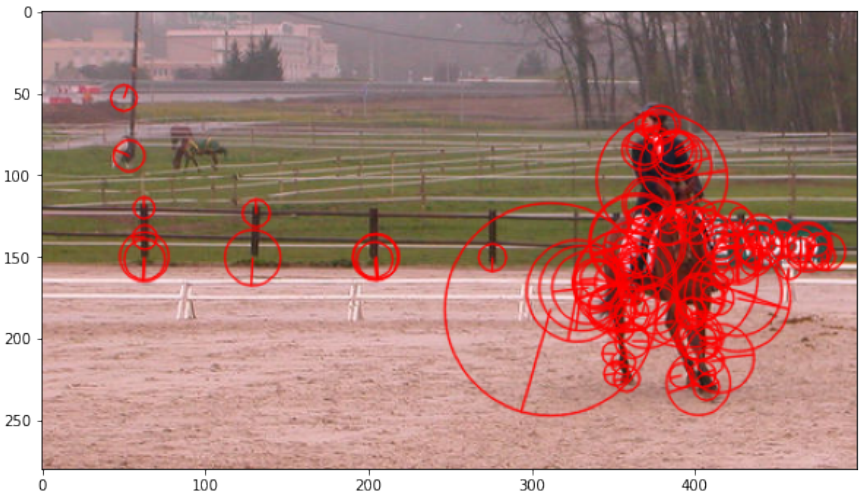


Figure 2.14: SURF keypoints. Adapted from [BTVG06]

The SURF descriptor is generated in two steps. The first step is to assign an orientation to a circular region around the keypoint. For that purpose, Haar-wavelet responses are calculated in x and y direction as d_x and d_y in the selected region and weighted with a Gaussian window according to the keypoint scale. Following that, the SURF descriptor is extracted from a square region around the keypoint and aligned to the dominant orientation.

The region is further split into square sub-regions to retain important spatial information. First, the feature vector is computed by summing Haar wavelet responses d_x and d_y in both directions for each sub-region. Additionally, to include the polarity of intensity changes, the sum of the absolute values of the responses, $|d_x|$ and $|d_y|$, is extracted and appended to the vector. A final 4D descriptor vector $\mathrm{v} = (\sum dx, \sum dy, \sum |d_x|, \sum |d_y|)$ of length 64 is obtained

for each sub-region. Normalizing the descriptor into a unit vector achieves invariance to contrast [BTVG06]. Figure 2.15 shows dividing the region around a keypoint into 4×4 sub-regions to extract the feature vector.

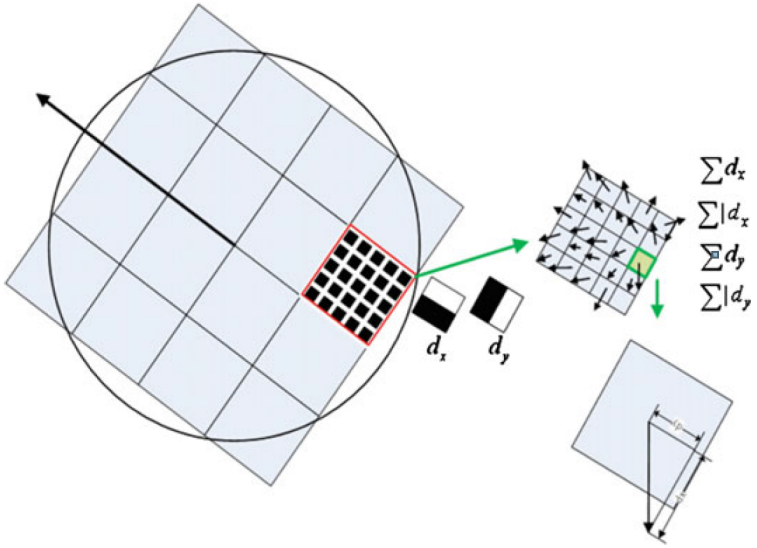


Figure 2.15: Dividing the interest region into 4×4 sub-regions for computing the SURF descriptor from [HAA16]

Both SIFT and SURF detectors are used in this work. As described in Section 1.2, this work aims to develop a deep learning-based framework in combination with these feature detectors. As will be explained later, the characteristic scales from these detectors, along with the keypoint positions in the image, are used to create the training dataset required in this work.

2.3 Deep Learning

Deep learning is a branch of machine learning that focuses on creating neural networks using algorithms inspired by the structure and operation of the human brain. These algorithms using neural networks extract knowledge from massive amounts of data to make a prediction or classification. Unlike conventional machine learning techniques, deep learning algorithms can describe complex relationships and abstraction in the data, improving accuracy in various applications like speech and image recognition, natural language processing, and autonomous systems [Nie15].

Deep neural networks typically consist of many layers, sometimes hundreds or even thousands, with each layer learning a higher-level data representation. These networks are trained using large amounts of data and the optimization algorithms, such as stochastic gradient descent and back-propagation, to update the weights and biases of the network [GBC16].

The following subsections describe the basics of machine learning, artificial neural networks, and convolutional neural networks required to understand the network architectures implemented in this work to estimate characteristic scales.

2.3.1 Artificial Neural Networks

The simplest possible neural network is the perceptron or single-layered neural network. It is inspired by the functioning of the biological neurons in the human brain. Figure 2.16 shows an example schematic of a biological and an artificial neuron. In the human brain, a neuron receives signals from other neurons through its *dendrites*, processes these signals in the *nucleus*, and sends an output signal through its *axon* to other neurons. This process can be modelled as a simple mathematical calculation, where the inputs are received, processed, and transformed into an output. This analogy to the functioning of the human brain has led to the development of artificial neural networks, including the perceptron. However, it is essential to note that the perceptron model is a highly simplified version of the complexity and functioning of the human brain and its biological neurons [Roe17].

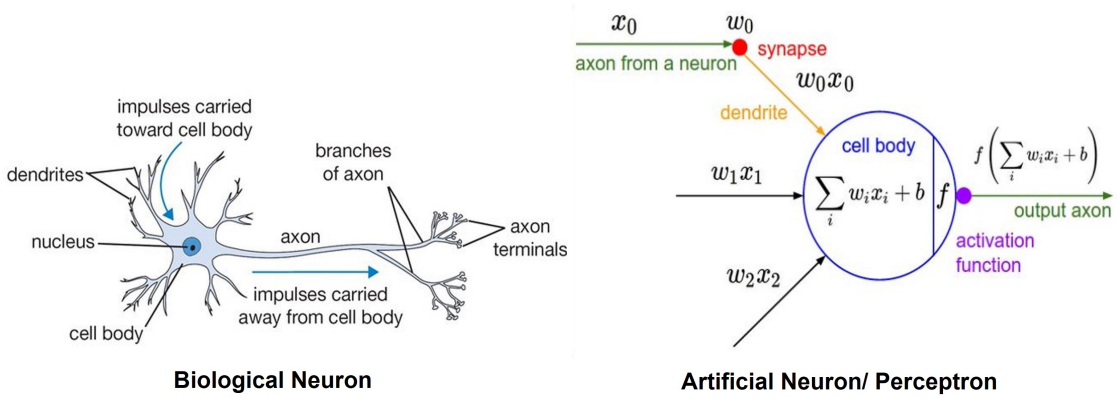


Figure 2.16: Schematic of a biological neuron (left) and an artificial neuron/perceptron (right) from [Roe17]

In contrast, a perceptron takes input from multiple branches, performs a simple calculation (such as a dot product) as seen in Figure 2.16, and produces a single output. A perceptron can only model linear relationships between inputs and outputs and cannot model more complex relationships.

A Multi-layer Perceptron (MLP) is an ANN that consists of multiple layers of perceptrons. It consists of multiple interconnected nodes or neurons organized into layers, with each neuron in a layer connected to every neuron in the previous and subsequent layers. A MLP is a stack of single neurons, where each neuron performs a simple calculation and passes its output to the next layer. The output of the final layer is used as the prediction or classification result for the network [DHS00].

By adding multiple layers of interconnected neurons, an MLP can model more complex relationships between inputs and outputs, as each layer can learn a higher-level representation of the data. The multiple layers of an MLP allow it to model non-linear relationships and make it capable of learning hierarchical representations of the data, thereby making it a powerful tool for various applications such as image and speech recognition, natural language processing, and autonomous systems [Nie15]. Figure 2.17 shows a basic schematic of a multi-layer perceptron with input layers, hidden layers, and an output layer.

However, a primary disadvantage of MLP is that they can become very complex, with many layers and neurons, leading to difficulties in training and understanding the model. A large number of neurons can also lead to overfitting, where the model fits the training data too closely and needs to generalize better to new data. Due to the complexity, the network can take a long time to train, especially when the dataset is large. This complexity is a significant barrier to their practical use.

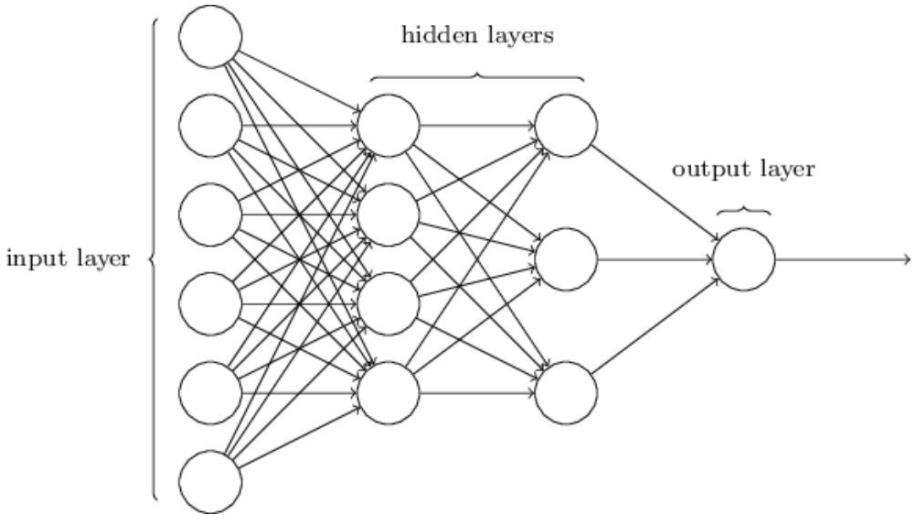


Figure 2.17: Schematic of a Multi-layer Perceptron (MLP) from [Nie15]

As seen in Figure 2.16, an example of a simple NN is shown. It consists of simple computing input units/nodes that exchange information from one unit to another through weight connectors and a bias unit. The net activation is a weighted sum of input units. Then, it

is passed through a non-linear activation function. In a NN, a layer is defined as the transformation of input vectors $\mathrm{x} = (x_1, x_2, ...x_D)^T$ to the output y with the learnable parameter $\mathrm{w} = (w_1, w_2, ...w_k)^T$ as seen in Figure 2.16 [Rei20, Roe17]:

$$\mathrm{y} = f(\mathrm{x}; \mathrm{w}) = f\left(\sum_i w_i x_i + b\right) \tag{2.10}$$

where w are the weight parameters of the network and b is bias.

NNs are usually either used for *classification* or *regression*. In the classification problem, the class labels are assigned to the training data. The binary classification problem deals with two data classes, so the target/labels are either "0" or "1". A regression problem involves predicting continuous target values. Given a set of input features, the model is trained to predict the corresponding target values. The goal is to minimize the difference between the predicted and target values.

Typically, the NN inputs are features extracted from the data, and the weight parameters are initialized with random values. The estimated output of the network is compared with the targets using the loss or error function. The loss function describes the variability of the estimated output to the target values. The objective is to minimize the loss by updating the weight parameters in the network using a back-propagation algorithm based on gradient descent [DHS00]:

$$\nabla \mathrm{w} = -\eta \frac{\partial J}{\partial \mathrm{w}} \tag{2.11}$$

where J is a loss function, and η is the learning rate determining the step size at which the weights are updated.

In this work, two commonly used loss functions for a regression problem *Mean Squared Error (MSE)* and *Mean Absolute Error (MAE)*, estimating continuous values, have been investigated. The MSE loss and MAE loss are shown in Equations 2.12 and 2.13, respectively [AA21].

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \tag{2.12}$$

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|\hat{y}_i - y_i| \tag{2.13}$$

here, n is the number of samples, $\hat{y}_i$ is the predicted value for the i^{th} sample, and y_i is the true value.

For both loss functions, the objective is to minimize the error between predicted and true values. The choice between the two will depend on the specifics of the task and the distribution of the target variable. MSE is more sensitive to large errors, while MAE is more robust to outliers.

Activation functions

As introduced previously, activation functions are mathematical operations applied to the inputs of layers in a neural network to produce the output activation. They play an essential role in the behaviour of the network and are a key component in determining the ability of a network to learn and make predictions. Several commonly used activation functions include $ReLU$, sigmoid, tanh, linear, softmax [DSC22]. The choice of an activation function depends on the task and architecture of the network.

Rectified Linear Unit (ReLU) function

The most used activation function in the hidden layers is the ReLU function. The output is zero when the input vector x is less than zero, and when x is above or equal to zero, the output is equal to x

$$g(x) = \max(0, x) \tag{2.14}$$

where g represents the activation function.

Sigmoid function

The sigmoid function is a mathematical function that takes in any real number and outputs a value between 0 and 1. It is generally used for binary classification problems and computed as:

$$g(x) = \frac{1}{1 + e^{-x}} \tag{2.15}$$

Softmax function

This activation function output is in the range of 0 and 1. The softmax function is used for multi-class classification problems and extends the above sigmoid function for K data classes. For an i_{th} class of data for K number of total classes, the softmax is computed as:

$$L_{S_i} = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}} \tag{2.16}$$

Linear function

The linear activation function, also known as an identity activation function, returns the input as the output, which makes it appropriate for regression problems where the target values are continuous. It is represented as:

$$g(x) = x \tag{2.17}$$

where x is the input to the activation function. This function has been used as an activation function in the output layer for all the networks implemented in this work. Using a linear activation function, the network can directly predict the target values without passing the output through non-linear transformations. This allows for a straightforward interpretation of the network's predictions since the predicted values are directly proportional to the network's output. Additionally, it is computationally efficient, as it does not require any complex mathematical operations to calculate. This can benefit regression problems where computational speed is a concern [LMAPH19].

2.3.2 Convolutional Neural Networks

Owing to the limitations of using NN, researchers developed Convolutional Neural Networks (CNNs), which incorporate convolutional and pooling layers that allow the network to process image data more effectively. The two main advantages of using CNNs over traditional NNs are *Sparse connectivity* and *Parameter sharing*.

In traditional NN, each output is connected to all inputs from the previous layer, leading to a dense connectivity pattern. In CNNs, the units in a layer are only connected to a local region in the previous layer, called a receptive field. This sparse connectivity pattern leads to a

much smaller number of parameters in the network, thereby reducing the computational requirements and allowing for larger and deeper networks. Additionally, each unit has weights and biases, leading to many parameters. In CNNs, the same set of weights and biases are used for each local region in the input, known as weight sharing or parameter sharing. This sharing implies that the same filter is applied to multiple input regions, reducing the number of parameters in the network and allowing for the recognition of translation-invariant features [GBC16]. Figure 2.18 shows sparse connectivity (left) and parameter sharing (right) in CNNs, respectively.

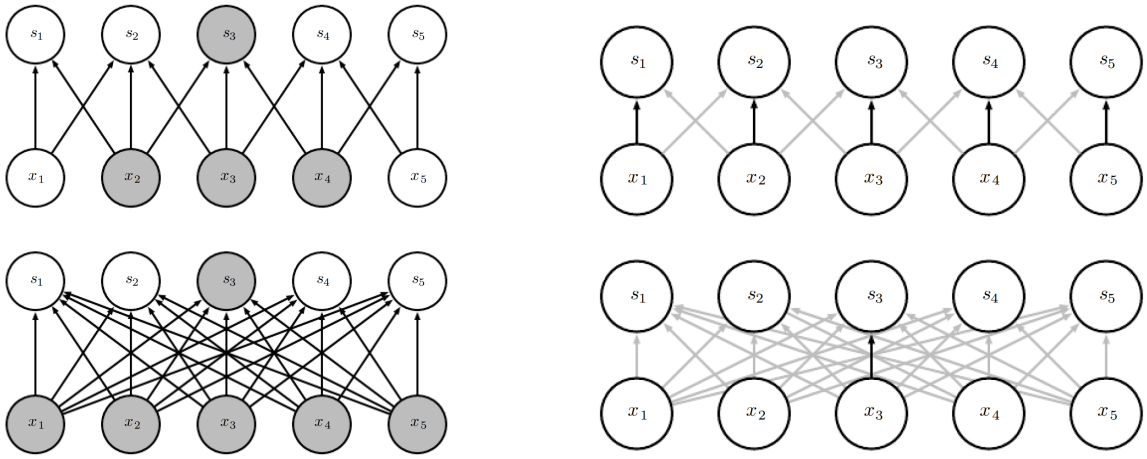


Figure 2.18: Sparse connectivity (left) and Parameter sharing (right) in CNNs from [GBC16]

A CNN comprises different layer types that allow the network to learn to extract patterns in the image data. The pooling layers reduce the spatial resolution of the data, thereby making the network less sensitive to small input translations. These innovations have led to remarkable advances in computer vision, and CNNs have become the dominant method for image classification, object detection, and other related tasks. The success of CNNs in computer vision has also led to the development of new architectures for other domains, such as natural language processing and audio processing [GBC16]. Figure 2.19 shows a typical CNN containing all the basic elements of a LeNet architecture [LBBH98, GW18].

Convolutional layer

A convolutional layer performs convolution operations on the input data or the output from another convolutional layer, where a filter (also called a kernel or weight matrix) is applied to each local region of the input. The filter slides over the input data and performs element-wise multiplications and sums to produce a scalar output, known as a feature map. For a

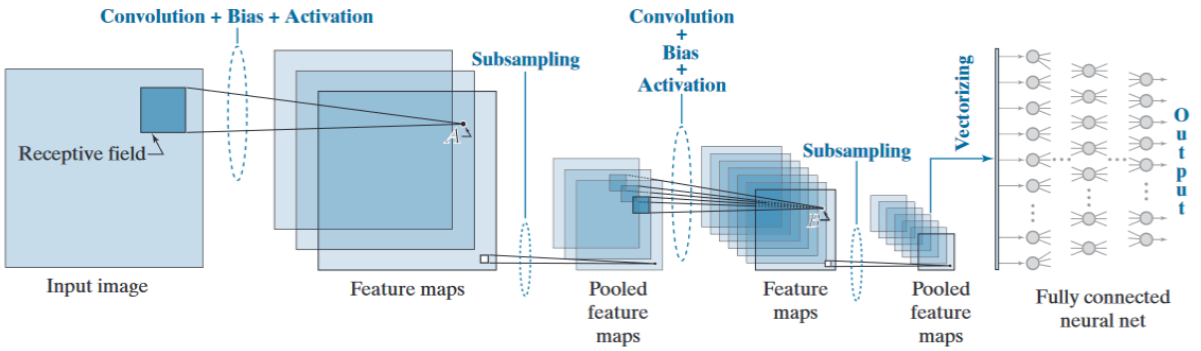


Figure 2.19: A CNN containing all the basic elements of a LeNet architecture from [LBBH98, GW18]

weighting function $w(a)$, where a is a measurement, $x(t)$ is a real-valued continuous function at time t, asterisk $*$ denoting the convolution operation, is given by:

$$s(t) = (x * w)(t) = \sum_{-\infty}^{\infty} x(a)w(t-a) \tag{2.18}$$

For example, a convolution operation over multiple axes at a time, such as for a 2D image I as input with a 2D kernel K is given by:

$$S(i,j) = (K * I)(i,j) = \sum_{m=1}^{M} \sum_{n=1}^{N} I(i+m, j+n)K(m,n) \tag{2.19}$$

Figure 2.20 shows an example of 2D convolution [GBC16].

The weights in a convolutional layer are learned during training to recognize specific features in the images, such as edges and shapes. By using multiple filters in a convolutional layer, the network can learn to recognize multiple types of features in the input. The layer also applies padding to the input data, adding extra zero values around the input border to prevent information loss during convolution. The stride of a filter, or step size, can also be adjusted to control the spatial resolution of the feature maps produced by the convolutional layer.

Pooling layer

The pooling layer performs sub-sampling operations on the feature maps produced by the preceding convolutional layer, reducing the network's spatial resolution and computational

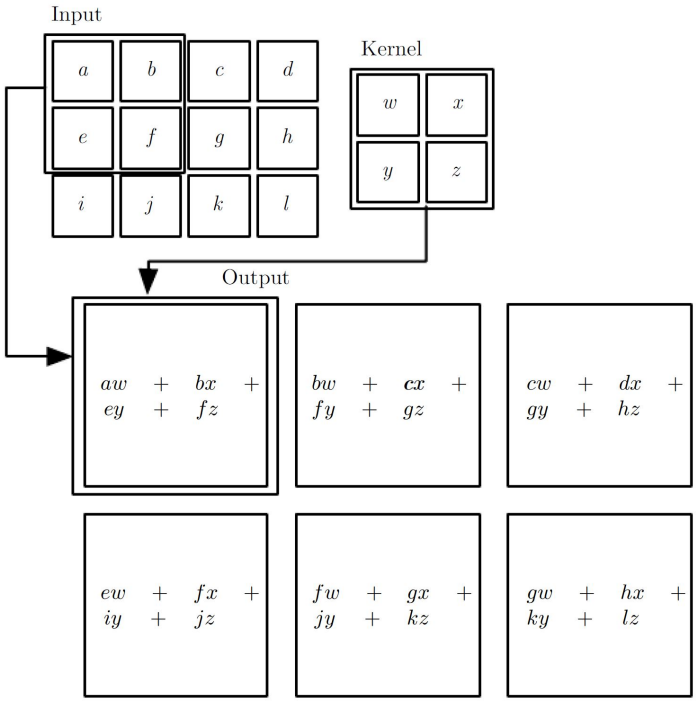


Figure 2.20: An example of 2D convolution by sliding kernel across an input image from [GBC16]

requirements. The pooling layer is usually applied after several convolutional layers and can be considered a form of non-linear down-sampling. The two common types of pooling layers in CNNs are *Max pooling* and *Average pooling*.

Max pooling selects the maximum value from a local region of the feature map. It emphasizes the most prominent features in the feature map while suppressing weaker features. Average pooling computes the average of all values in a local region of the feature map. It can smooth the feature map, thereby reducing the impact of isolated high values and emphasizing the overall pattern in the feature map [GBC16]. Figure 2.21 shows examples of max and average pooling 3×3 kernel down-sampling a 2D representation. These operations drastically reduce the number of parameters, implying a lower chance of overfitting.

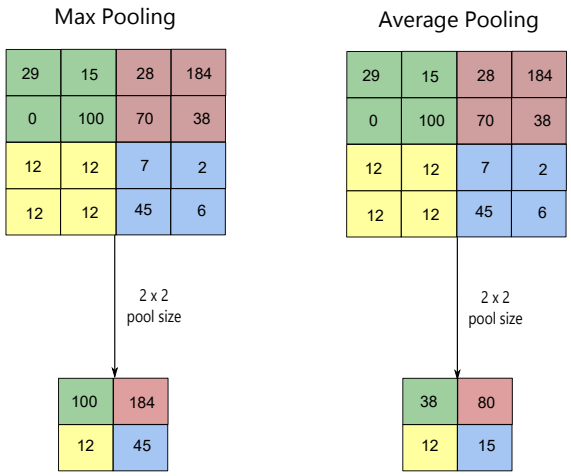


Figure 2.21: An example of pooling operations on an input image from [YSS19]

Fully Connected layer

The fully connected layer flattens the 2D feature maps computed by the preceding convolutional and pooling layers into a single 1D vector to compute outputs, which are the activations of the units in the layer. This layer mimics the functionality of a traditional NN. These activations are then fed into an activation function, such as ReLU or Linear, to compute the output of classification or regression problem. This layer has dense connectivity, meaning that each unit is connected to every unit in the previous layer. This connectivity allows the network to learn a rich set of feature representations, capturing the spatial and semantic relationships between the features. The flattening operation is shown in Figure 2.22.

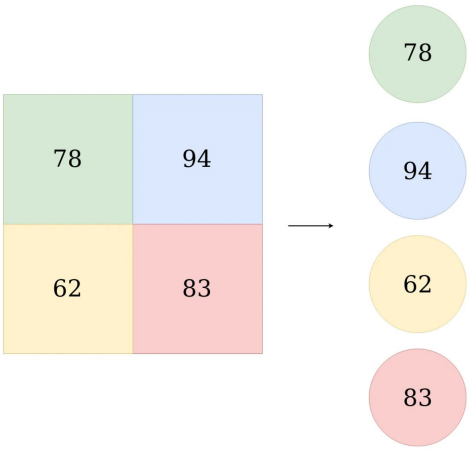


Figure 2.22: Flattening operation to obtain a 1D vector from [Sah18]

2.3.3 Optimization Algorithms

An optimization algorithm is a mathematical procedure to find a function's minimum or maximum value. In deep learning, optimization algorithms are used to find the values of the weights and biases in a neural network that result in the best performance on a given task. The algorithms aim to find the values of the network parameters that minimize a loss function, which measures the difference between the network's predictions and the true target values. Many different optimization algorithms can be used for this purpose, including *Stochastic Gradient Descent (SGD), Root Mean Squared Propagation (RMSprop), Adaptive Moment Estimation (Adam), Adagrad, and Adadelta* [EMA23, GBC16]. Each of these optimization algorithms uses a different approach to find the values of the network parameters that minimize the loss function. Some algorithms, such as SGD, use a simple gradient-based approach to update the parameters, while others, such as Adam, use more complex procedures involving accumulating historical gradients. The choice of an optimization algorithm for a given deep learning problem primarily depends on factors including the size and complexity of the network, the nature of the task at hand, and the computational resources available.

Stochastic Gradient Descent (SGD)

The fundamental principle of SGD is to compute the gradient of the loss function with respect to the network parameters and then update the parameters in the direction of the negative gradient. Given a set of parameters, denoted as θ, and a loss function, $L(\theta)$, the parameters are updated as follows:

$$\theta = \theta - \eta \nabla \theta L(\theta) \tag{2.20}$$

where η is the learning rate, which controls the step size of the parameter updates, and $\nabla \theta L(\theta)$ is the gradient of the loss function with respect to the parameters. This allows the algorithm to scale large datasets and helps to introduce some randomness into the optimization process, which can help to avoid getting stuck in local minima. It is important to note that the learning rate, η, is a hyper-parameter selected in advance and can significantly impact the optimization algorithm's performance. A good value of η can be found through experimentation, which typically decreases over time as the optimization process progresses.

There are several other variants of SGD with hyper-parameters such as *Momentum, Nesterov gradient, regularization* to name a few. More information can be found in the literature [GBC16].

Root Mean Squared Propagation (RMSprop)

RMSprop is similar to SGD with momentum, but instead of using a moving average of the gradients, it uses a moving average of the squared gradients. Given a set of parameters, denoted as θ, and a loss function, $L(\theta)$, the parameters are updated as follows:

$$\begin{aligned} g &= \nabla\theta L(\theta) \\ v &= \beta * v + (1-\beta)g^2 \\ \theta &= \theta - \frac{\eta * g}{\sqrt{v+\epsilon}} \end{aligned} \tag{2.21}$$

where g is the gradient of the loss function with respect to the parameters, v is a moving average of the squared gradients, β is a decay rate that controls the decay of v over time, α is the learning rate, and ϵ is a small value used to avoid division by zero. In practice, the moving average, v, is initialized to 0, and the decay rate, β, is typically set to 0.9.

Adaptive Moment Estimation (Adam)

Adam combines the advantages of SGD and RMSprop and is known for its fast convergence and good performance on various problems due to its ability to handle sparse gradients, a problem arising due to gradients containing zero values during the training process.

Given a set of parameters, denoted as θ, and a loss function, $L(\theta)$, the parameters are updated as follows:

$$\begin{aligned} m &= \beta_1 * m + (1-\beta_1)\nabla\theta L(\theta) \\ v &= \beta_2 * v + (1-\beta_2)(\nabla\theta L(\theta))^2 \\ \theta &= \theta - \frac{\eta * m}{\sqrt{v}+\epsilon} \end{aligned} \tag{2.22}$$

where m and v are moving averages of the gradients and squared gradients, respectively, β_1 and β_2 are decay rates that control the decay of m and v over time, α is the learning rate, and ϵ is a small value used to avoid division by zero. In practice, the moving averages, m

and v are initialized to 0, and the decay rates, β_1 and β_2, are typically set to 0.9 and 0.999, respectively.

SGD, Adam, and RMSprop have been investigated in this work with different network architectures.

2.3.4 Residual Neural Network

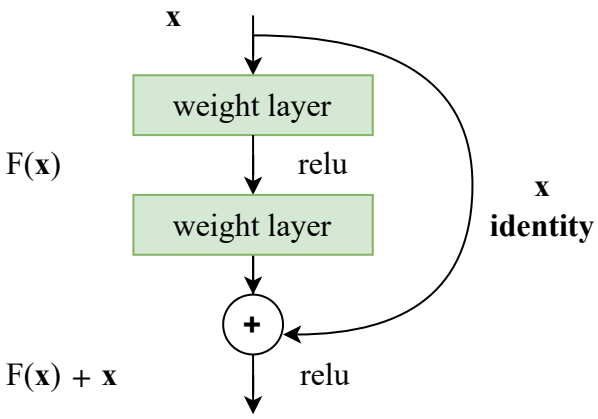


Figure 2.23: The building block for residual learning from [HZRS16]

Some deep CNN architectures have large number of layers that result in the problem of vanishing/exploding gradients. It causes the saturation of accuracy during the training process [SGS15, HS15]. However, a deep residual learning framework called Residual Neural Network (ResNet) is proposed in [HZRS16] to overcome gradient degradation, where residual mapping explicitly connects two layers. The advantage of ResNet is that many layers can be trained easily without the increment of the percentage of error during the training period. Figure 2.23 depicts the basic block of a ResNet. The formulation of F(x) + x is realized by the feed-forward network with a skip or shortcut connections, where the data $\mathbf{x}$ from the previous layer is again added after some operation F(·) (i.e., ReLU) on it.

In a ResNet block, the output of the convolutional layers is added to the input of the block, and the resulting sum is passed through the activation function. The final output of the block is then used as the input to the next block in the network. This process is repeated multiple times, allowing the network to build up a deep architecture while preserving the ability to learn a residual mapping [HZRS16]. A ResNet architecture built using the ResNet block is further explained in Section 4.3.4.

Network performance, hyperparameters, and other training considerations

Neural networks typically have thousands to millions of parameters. With so many learnable parameters, it allows the networks to capture complex data structures. However, two commonly occurring problems affecting a network's performance are "Underfitting" and "Overfitting" [GC19]. To mitigate this, there are several ways to overcome both problems. Hyperparameter tuning is one such used approach. Hyperparameters are parameters set before training and define how the neural network learns. The most important hyperparameters are *Learning rate*, *Batch size*, and *Epochs*, which can be tuned to improve the training process. Two other standard techniques, *Early-Stopping* and *Learning rate decay* that improve the performance and efficiency of the training process, have been employed in this work [GBC16]. The following subsections explain the mentioned problems, hyperparameters, and techniques.

Underfitting and Overfitting

Underfitting occurs when a network is unable to learn the desired patterns or features from the given data. This implies that the network is not complex enough to capture the underlying complexity of the data, which results in high bias and low variance. As a result, the model performs poorly on the training and new data it has not trained on. Whereas *overfitting* occurs when a network is too complex and begins to fit the noise in the training data rather than the underlying pattern. This also leads to poor generalization performance on new, unseen data. Figure 2.24 shows (a) underfitting and (c) overfitting. In both cases, the estimation of a test instance might be very weak, while in a good fit, as in Figure 2.24 (b), a test instance not seen in the training phase is estimated with smaller errors. The ability of the network to classify or estimate the unseen test data is referred to as generalization [GC19]. In this work, underfitting resulted in preliminary experiments with standard VGG16 architecture. Better generalization was achieved with adapted and modified variants of VGG16 architecture as shown in Sections 4.3.2 and 4.3.3. In terms of overfitting, it was seen that there would be overfitting to a certain degree with every network, especially with keypoints having relatively smaller characteristic scales, as discussed in Chapter 5.

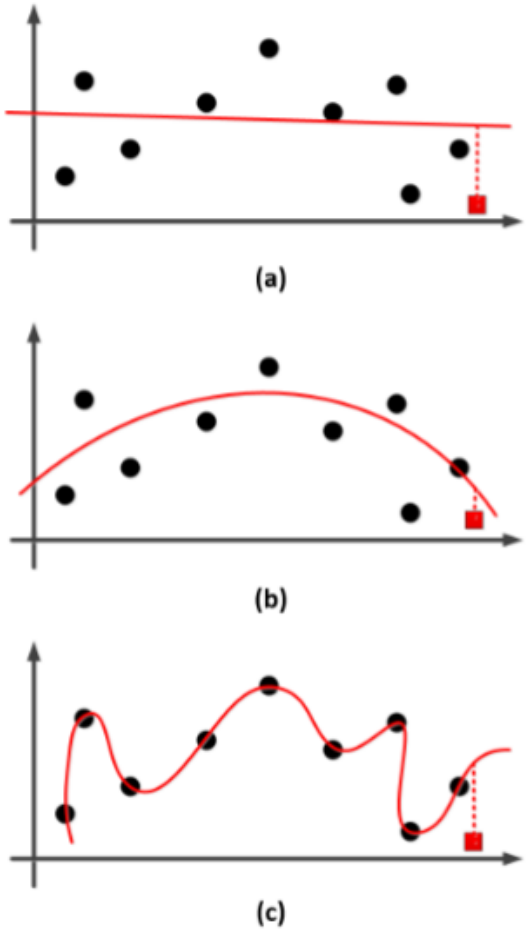


Figure 2.24: Conceptual visualization of (a) underfitting, (b) good fit, and (c) overfitting. The black circles and red squares are training and test instances, respectively. The red curve is the fitted curve from [GC19]

Batch Size and Epoch

Batch size determines the number of samples or *(data, label)* pairs that are used in each iteration of the optimization process. A smaller batch size can lead to more noise in the training process which results in overfitting, but it can also result in faster convergence. On the other hand, a larger batch size can reduce the noise in the training process, but it may also increase the time it takes for each iteration [Bro18]. A custom data generator was implemented in this work as described in Section 4.1.1, which outputs batches of data. Several batch sizes in [32, 64, 128, 256, 512] were tested, and **128** was determined to be the optimal batch size in this work.

An *Epoch* determines the number of times the entire training dataset in batches is used to train the network. A larger number of epochs can lead to overfitting, while a smaller number

can result in underfitting [Bro18]. In this work, the default number of epochs was fixed at **100** with early stopping.

Early-Stopping

Early stopping is a technique commonly used to prevent overfitting by monitoring the performance of the model on the validation dataset during training. The training process is stopped early if the validation performance does not improve or begins to deteriorate for a certain number of epochs. It helps to prevent the model from learning to fit noise in the training data and can improve the generalization of the model on new, unseen data [GBC16]. An early stopping after **10** epochs on nearly no improvement of validation RMSE is employed in this work.

Learning rate

The *Learning rate* controls the step size at which the optimizers *Adam, SGD*, and *RMSprop* update the weights during the optimization process. A smaller learning rate can lead to more accurate results, but it can also increase the time it takes for the network to converge. However, a larger learning rate can speed up the convergence process, but it may result in the network overshooting the optimal solution [GBC16]. Several learning rates were tested in the range of 0.1 to 0.00001, and **0.0001** was determined to be the optimal learning rate in this work. It was combined with a *Learning rate scheduler/decay* to reduce the learning rate further to improve the training process.

Learning rate scheduler/decay

The *Learning rate scheduler/decay* is a technique used to adjust the learning rate during training to improve the convergence of the model. It involves reducing the learning rate over time, either by a fixed amount or based on the performance of the model on the training or validation data. This helps to prevent the optimization process from overshooting the optimal solution and can improve the convergence and generalization of the network [GBC16]. A learning rate decay of $\frac{1}{2}$ for every **5** epochs on no improvement of validation RMSE is employed in this work.

3 Related Work

This chapter provides an overview of the existing literature, previous studies, and research in estimating characteristic scales and deep learning-based feature detection and description. Furthermore, a summary and insights from the overview that is useful for this work are presented.

3.1 Deep Learning-based Feature Detection and Description

Deep learning and the use of CNNs have become increasingly popular for feature detection in computer vision tasks due to several advantages over traditional feature detection methods such as SIFT and SURF. Some of the advantages for using CNNs over traditional feature detection methods are [MJF+21]:

- CNNs are end-to-end trainable, meaning they can learn feature extraction and classification/regression from data in a single network. In contrast, SIFT and SURF require independent processes. For example, in keypoint detection and description, few of the CNN-based methods combine different stages of keypoint detection, such as constructing scale-space and localizing keypoints in one network as opposed to multiple steps of traditional methods.

- In several applications, including object detection, image classification, and segmentation, CNNs outperforms traditional methods due to their ability to extract high-level semantic features from large amounts of training data.

- CNNs are well-suited to large-scale recognition tasks, such as large-scale image classification, where they are trained on large amounts of data to learn robust features.

- CNNs can be optimized for computation and memory, making them efficient for real-time applications. They can also be accelerated using hardware, such as GPUs and TPUs.

In the following subsections, selected methods for CNN-based keypoint detection are discussed. As discussed in Chapter 1, estimating characteristic scales is an important stage in many end-to-end feature detectors. Since this work focuses on methods including the estimation of characteristic scales, only sections of the detectors focusing on estimation are discussed in detail at the end of the method.

3.1.1 LIFT

Learned-Invariant Feature Transform (LIFT) [YTLF16] is a deep learning-based feature detection method designed to learn description vectors robust to viewpoint, illumination, and deformation changes. LIFT provides a comprehensive solution for keypoint detection, orientation assignment, and keypoint description as shown in Figure 3.1. These three key elements, as marked in Figure 3.1 are implemented using individual CNN networks. A unique deep network architecture has been developed using Siamese network architecture that integrates the three components for local feature detection and description into a single differentiable network. A problem-specific learning strategy in LIFT involves learning the Descriptor first, followed by learning the Orientation Estimator using the learnt Descriptor, and finally, the Detector, which is conditioned on the other two. This makes it possible to fine-tune the Orientation Estimator for the Descriptor and the Detector for the other two components.

LIFT method uses layers from the Spatial Transformer Network (STN) in order to achieve invariance to translation and rotation for the output of the Detector and the Orientation Estimator, respectively. A STN is a type of CNN architecture designed to perform spatial transformations on the input image, such as translation, rotation, scaling, and affine transformations. The network is designed to be an end-to-end differentiable module that can be integrated into a larger neural network to improve its ability to achieve invariance to geometric transformations and variations in the input image [JSZ+15].

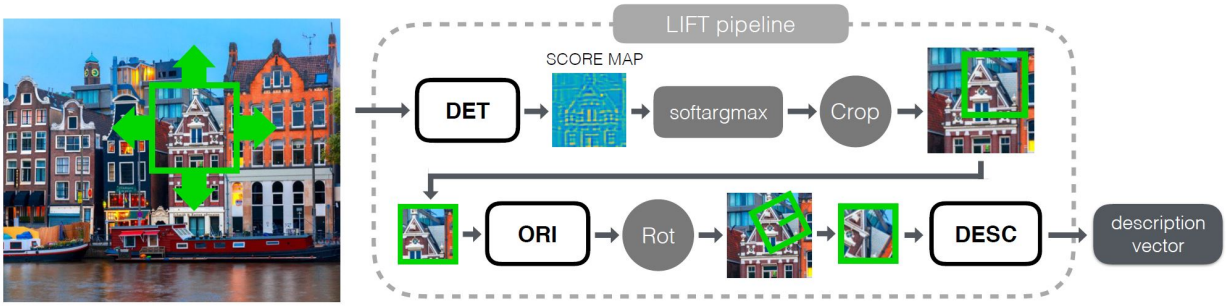


Figure 3.1: LIFT feature extraction pipeline consisting of three major components: the Detector, the Orientation Estimator, and the Descriptor from [YTLF16]

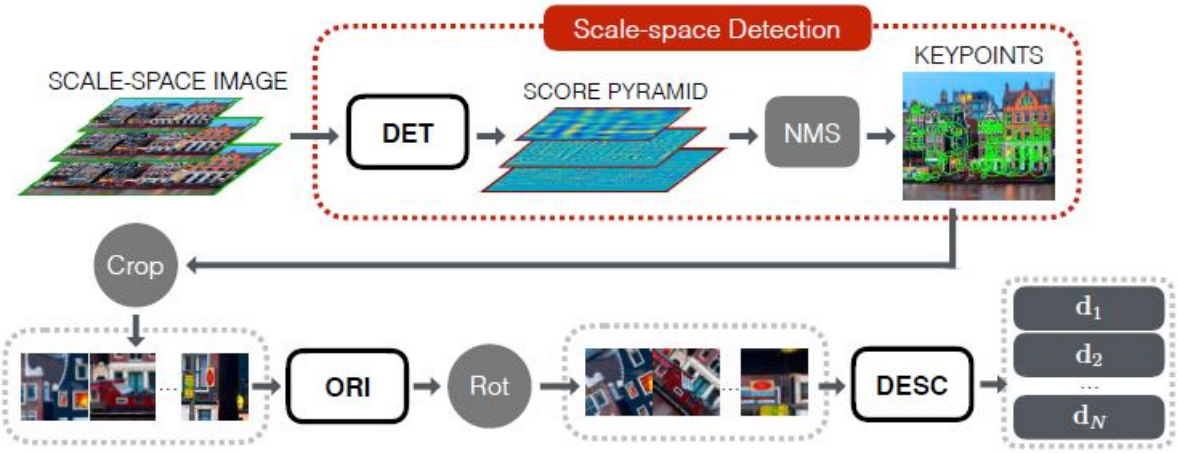


Figure 3.2: An overview of LIFT architecture with a keypoint detector, a spatial transformer, and a descriptor from [YTLF16]

Detector

The detector computes a score map based on the input image patch $\mathbf{P}$. The detector is applied independently on the image patch at multiple resolutions to get score maps in scale space, as shown in Figure 3.2. It uses a convolutional layer with a piecewise linear activation function to compute the score map. Generally, a *Non-Maxima-Suppression (NMS)* identical to that of [Low04] is applied to the score map, which allows the keypoints to be identified. A differentiable substitute for the NMS is the `softargmax` function. Since it must be ensured that the gradient of the cost function can be reliably computed during the learning process, the differentiability characteristic of `softargmax` is particularly valuable. The `softargmax` function is applied to the score map to locate pixels, (x, y) with exceptionally high activation values.

Orientation Estimator

The Orientation Estimator determines the orientation of the image region and rotates it to describe the area around the keypoint in an orientation-invariant manner. From the located pixels (x, y) from the detector, a smaller patch, $\mathbf{p}$, is extracted around the located pixels in input patch $\mathbf{P}$ and used as an input to the Orientation Estimator. A Spatial Transformer Layer `Crop(.)` is used to extract the patch. Together with the patch $\mathbf{p}$ and the location (x, y), the Orientation Estimator predicts an orientation. Another Spatial Transformer Layer `Rot(.)` is used to compute a rotated version of patch $\mathbf{p}$.

Descriptor

For the Descriptor network, three convolutional layers with *tanh* activation function, l_2 pooling, and subtractive normalization are used. It uses the rotated patch from the Orientation Estimator to compute a feature vector.

Characteristic Scale Estimation

The highlighted region in red in Figure 3.2 is responsible for detecting keypoints in scale space. As described in Section 3.1.1, the score maps are obtained in scale space. A conventional NMS technique identical to that of [Low04] is used to find the keypoint location (x, y). The characteristic scale can be estimated by applying a `softargmax` function on the score maps.

3.1.2 LF-Net

Local Feature Network (LF-Net) [OTFY18] is an architecture and a training strategy to learn features that combines the detection and description of keypoints in one architecture. Similar to LIFT, a Siamese network architecture is used for training, which requires only two branches. Figure 3.3 shows the architecture of LF-Net. In LF-Net, a two-branch network is used where two identical copies of the network processing two corresponding images are used. The detector network using CNNs generates a scale space for score maps, which determines the location of keypoints, characteristic scales, and orientations of keypoints. The STN determines image regions for the selected keypoints. The descriptor network then determines feature vectors for the keypoints based on the image regions.

Detector

The detector in LF-Net is a dense, multi-scale, fully convolutional network that outputs keypoint locations, scales, and orientations. The fully convolutional network generates a feature map for an image, which is further used to extract keypoint locations along with their scale and orientation. It has generally been shown that feature maps provide a mid-level representation which estimates multiple attributes that helps improve the predictive capabilities of deep networks. Additionally, such a network allows for larger batch sizes of training images to train a robust detector [Kok17].

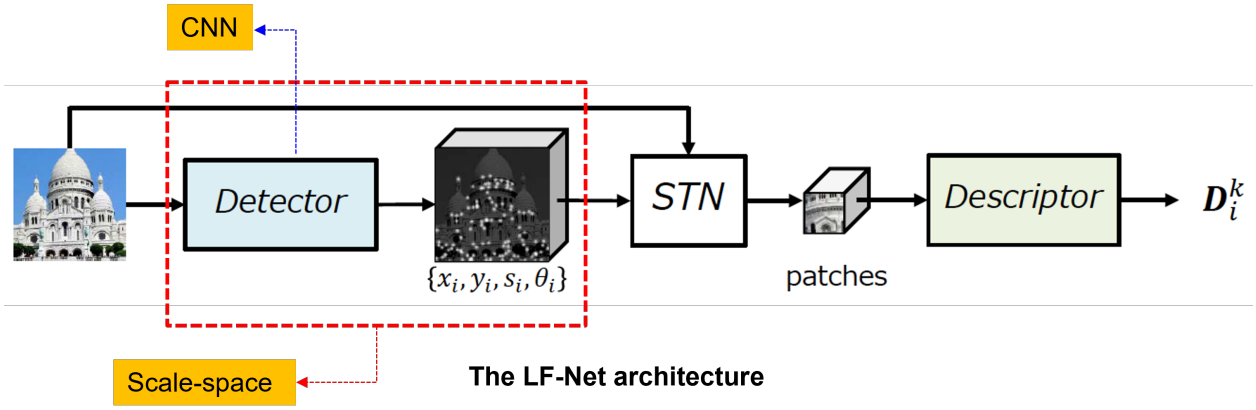


Figure 3.3: The LF-Net architecture from [OTFY18]

A ResNet layout as explained in Section 2.3.4 with three blocks, each with 5×5 convolutional layer with Batch Normalization layer, Leaky-ReLU activation functions, and another set of 5×5 convolutional layer is used as the network. With the generated feature map, a scale-space response is generated by resizing the map N times, at equal intervals between $\frac{1}{R}$ and R, where $N = 5$ and $R = \sqrt{2}$. The resulting maps are further convolved with N independent 5×5 filters, resulting in N score maps $\mathbf{h}^n$ for $1 \leq n < N$, for each scale. Similar to LIFT, A NMS is applied with a softmax operator to produce N sharper, more salient score maps. Finally, all the score maps are merged to form a final scale-space score map, $\mathbf{S}$, which is scale-invariant in nature. It is defined as:

$$\mathbf{S} = \sum_{n=1}^{N} \mathbf{h}^n \odot \text{softmax}_n(\mathbf{h}^n) \tag{3.1}$$

where $\odot$ is the Hadamard product [Hor90].

From the generated scale-invariant map, the top K pixels with the highest score are selected as keypoints, and a local `softargmax` is applied for sub-pixel accuracy.

Orientation estimation

A 5×5 convolution on the feature map, which outputs two values at each keypoint, is used. The two values are the sine and cosine of the orientation to compute a dense orientation map θ using the `arctan` function.

Descriptor

With the scale map $\mathbf{S}$ and the orientation map θ, K quadruplets of the form $\mathbf{p}^k = \{x, y, s, \theta\}^k$, descriptors are to be computed. Patches are cropped around keypoints produced by the detector. The descriptor produces local descriptors for the cropped patches. Input images are converted to grayscale and normalized individually using their mean and standard deviation to achieve robustness with respect to brightness and contrast.

Characteristic Scale Estimation

Similar to LIFT, the highlighted region in red in Figure 3.3 is responsible for detecting keypoints in scale space. From Section 3.1.2, once the scale-invariant score map, $\mathbf{S}$ is generated, by applying a `softargmax` function on N score maps, the characteristic scale of the keypoint can be estimated.

3.1.3 Key.Net

Key.Net [LRPM19] is a keypoint detection method that combines conventional and learned CNN features within a shallow multi-scale architecture. Conventional detectors used in this method for detecting features, for example, are the Harris detector, the Hessian detector, or the DoG as described in Section 2.2.2, using first and second-order derivatives of the images to determine possible keypoints. The conventional features act as soft anchors to reduce the number of parameters for learned CNN features, improving stability and convergence during backpropagation. A scale-space representation is used within the network to detect and extract keypoints at different scales. Figure 3.4 shows the architecture of Key.Net.

Conventional features are composed of first-order gradients I_x and I_y, second-order moments $I_x * I_y$, I_x^2 and I_y^2 as in Harris detector [HS+88]. Second-order derivatives I_{xx}, I_{yy} and I_{xy} as in the Hessian matrix, $\mathcal{H}$ in the Hessian detector and DoG detectors. To compute the

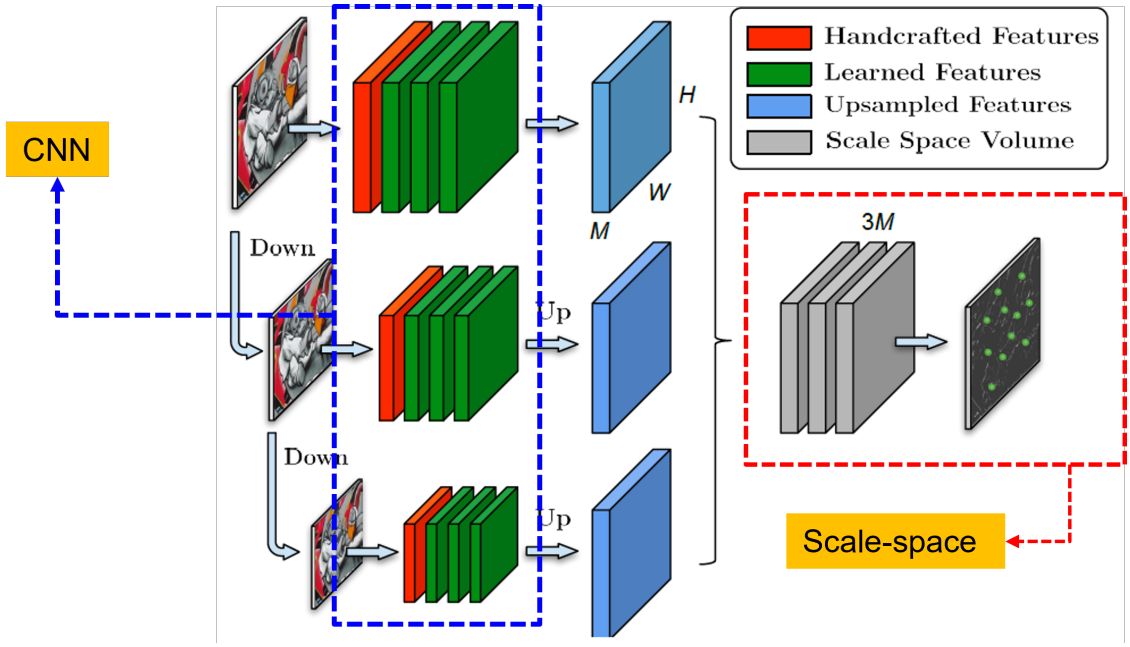


Figure 3.4: An overview of Key.Net architecture from [LRPM19]

learned CNN features, a convolutional layer with M filters, a batch normalization layer and a ReLU activation function is used.

The input to the network are three scale levels of the input image, where each image is blurred and downsampled by a factor of 1.2. The feature maps from the conventional features are concatenated and provided to the stack of learned filters at each scale level. Feature maps are then upsampled, concatenated and given as input to the last convolutional filter, which computes the final response map $\mathcal{R}$.

Key.Net consists of an Index Proposal (IP) layer where the local maxima corresponding to the keypoint locations are extracted from the response map $\mathcal{R}$ output. A spatial softmax function is used over a window w_i of size $N \times N$ in $\mathcal{R}$, where the score value at each coordinate (u, v) for the window w_i, is exponentially scaled and normalized:

$$m_i(u, b) = \frac{e^{w_i(u,v)}}{\sum_{j,k}^{N} e^{w_i(j,k)}} \tag{3.2}$$

The exponential scaling ensures the maximum for the score value is significant and the coordinates calculated as the weighted averages $[x_i, y_i]$ leads to an approximation of the maximum coordinates [LRPM19]:

$$(x_i, y_i)^T = (\bar{u}_i, \bar{v}_i)^T = \sum_{u,v}^{N} [W \odot m_i, W^T \odot m_i]^T + c_w \tag{3.3}$$

where W is a kernel of dimension $N \times N$, $\odot$ the Hadamard product, and c_w is the top-left coordinates of w_i. The operation in Equation 3.3 is similar to NMS but, unlike NMS, it is differentiable, which allows the function to be optimized through gradient-based methods such as backpropagation. Furthermore, a covariant constraint loss is minimized to detect robust features in a transformation-invariant manner. Figure 3.5 shows the Siamese training process. The covariant loss $\mathcal{L}_{\mathcal{IP}}$ for two images I_a and I_b with a ground truth homography $H_{b,a}$ is given as:

$$\begin{aligned} \mathcal{L}_{IP}(I_a, I_b, H_{a,b}, N) = \sum_i \alpha_i ||[x_i, y_i]_a^T - H_{b,a}[\hat{x}_i, \hat{y}_i]_b^T||^2, \\ \text{and} \quad \alpha_i = \mathcal{R}_a(x_i, y_i)_a + \mathcal{R}_b(\hat{x}_i, \hat{y}_i)_b \end{aligned} \tag{3.4}$$

where $\mathcal{R}_a$ and $\mathcal{R}_b$ are the response map of I_a and I_b respectively. $[x_i, y_i]_a$ and $[\hat{x}_i, \hat{y}_i]_b$ are the points extracted by the IP layer and actual coordinates (NMS), respectively. α_i controls the contributions of individual points to the loss function based on their score. Accordingly, keypoints with high scores contribute more strongly to the loss function, meaning the system must pay more attention to the relevant keypoints during the training process.

The IP layer computes only one location per window w_i, impacting the overall number of keypoints detected for an image. It will depend on the window size N. To mitigate this challenge, an extension to the IP layer called the Multi-Scale Index Proposal (M-SIP) layer is proposed where image features are accumulated for a spatial window w_i and within the surrounding scales as indicated in [DS15]. Multiple window sizes allow the network to estimate keypoints across various scales. Additionally, other keypoints within a larger window act as anchors for the estimated dominant keypoint location.

In M-SIP, the response map $\mathcal{R}$ is split into grids, each of a window size $N_s \times N_s$. Each window computes candidate keypoint locations, and the network is allowed to learn robust features over different scales through an intrinsic process of scoring and ranking of keypoints within the network. A new loss function is proposed, which is the average of covariant constraint losses from multiple scale levels:

$$\mathcal{L}_{MSIP}(I_a, I_b, H_{a,b}) = \sum_{s=1}^{M} \lambda_s \mathcal{L}_{\mathcal{IP}}(I_a, I_b, H_{a,b}, N_s) \tag{3.5}$$

where s is the index of the scale level, M is the total number of scale levels, N_s is the window size, $\mathcal{L}_{\mathcal{IP}}$ is the covariant constraint loss, λ_s at scale level s is a parameter that decreases proportionally to the increasing window area. Figure 3.5 shows the Siamese training process and response maps for different window sizes. The covariant constraint loss $\mathcal{L}_{MSIP}$ is computed for five scale levels, with $N_s \in [8, 16, 24, 32, 40]$ and $\lambda_s \in [256, 64, 16, 4, 1]$. Larger window sizes usually have higher mean errors, and thus λ_s is the largest for the smallest window [LRPM19].

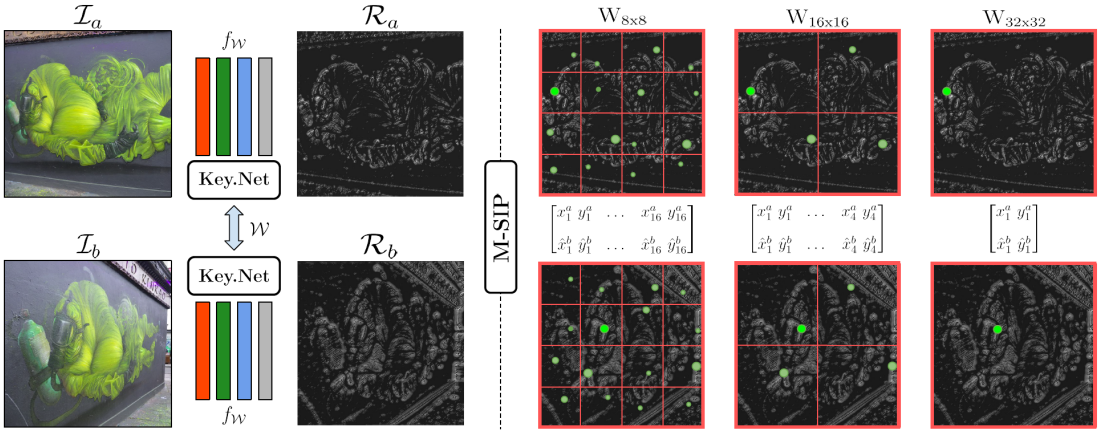


Figure 3.5: Siamese training process from [LRPM19]

Characteristic Scale Estimation

In order to have a scale space representation of the input image, three blurred and downsampled versions of the input image are utilized to extract conventional and learned features. Further, a multi-scale covariant loss function is implemented, which is shown in Equation 3.5. By minimizing this loss function, it is also possible to detect keypoints in a scale-invariant manner. It is possible to determine the characteristic scale by obtaining the scale level index s of the detected keypoint.

3.2 Summary

Various CNN-based keypoint detectors that employ an end-to-end approach for feature detection and description have been discussed. Assigning characteristic scales is one of the components of these detectors. For instance, in [YTLF16] and [OTFY18], score maps generated by CNNs are used to detect keypoints in a scale-space manner. These detectors follow similar scale-space construction to accurately localize keypoints as the conventional algorithms like SIFT and SURF using NMS and `softargmax` function. Directly using these detectors can make isolating the scale-space construction and estimating characteristic scales from the other steps of keypoint detection challenging. In [LRPM19], keypoints are detected in multiple scale levels, and the characteristic scale is estimated by obtaining the scale level index of the detected keypoint. This method could be used as a starting point to develop a loss function that incorporates multi-scale information to estimate characteristic scales. Moreover, the network architectures used in these detectors, such as ResNet blocks in LF-Net, can be leveraged to design a framework for characteristic scale estimation.

Popular datasets like PASCAL Visual Object Classes [EVGW$^+$] and ImageNet are used as training data. These datasets are particularly suitable because they contain several thousand to millions of real images with various viewing conditions, such as pose and illumination [MJF$^+$21].

4 Development and Implementation

In this chapter, the development and implementation of the framework for characteristic scale estimation are discussed in detail. The first step is to create a dataset of input images with keypoints and their characteristic scales. The images for the dataset are obtained from the PASCAL Visual Object Classes [EVGW+]. For the images, feature detectors, SIFT and SURF are used to obtain the keypoints and their characteristic scales. Section 4.1 describes the dataset creation and pre-processing. In Section 4.2, the overall framework and the implementation details in this work are described. Finally, Section 4.3 describes the implementation of the selected network architectures.

4.1 Dataset Creation and Pre-processing

4.1.1 Dataset Creation

The goal is to create a dataset containing input images and corresponding keypoint information so that a CNN can learn to estimate characteristic scales with a supervised training procedure. The images for the dataset are obtained from the PASCAL Visual Object Classes (VOC) [EVGW+]. It is a publicly available dataset of real images and annotations mainly used by the vision and machine learning communities. It was created by the Visual Geometry Group at the University of Oxford in 2005 as its first edition and comprised approximately 5000 images collected from the photo-sharing website *flickr*. The latest edition in 2012, used in this work, comprises approximately 17,000 images. The main objective of making this dataset publicly available was as a standard for the VOC challenge for *object classification* and *object detection*. This dataset is now the standard for the two mentioned tasks. However, this dataset is chosen for this work since it comprises real images with various viewing conditions, such as pose and illumination. Figure 4.1 shows some example images from the PASCAL dataset.

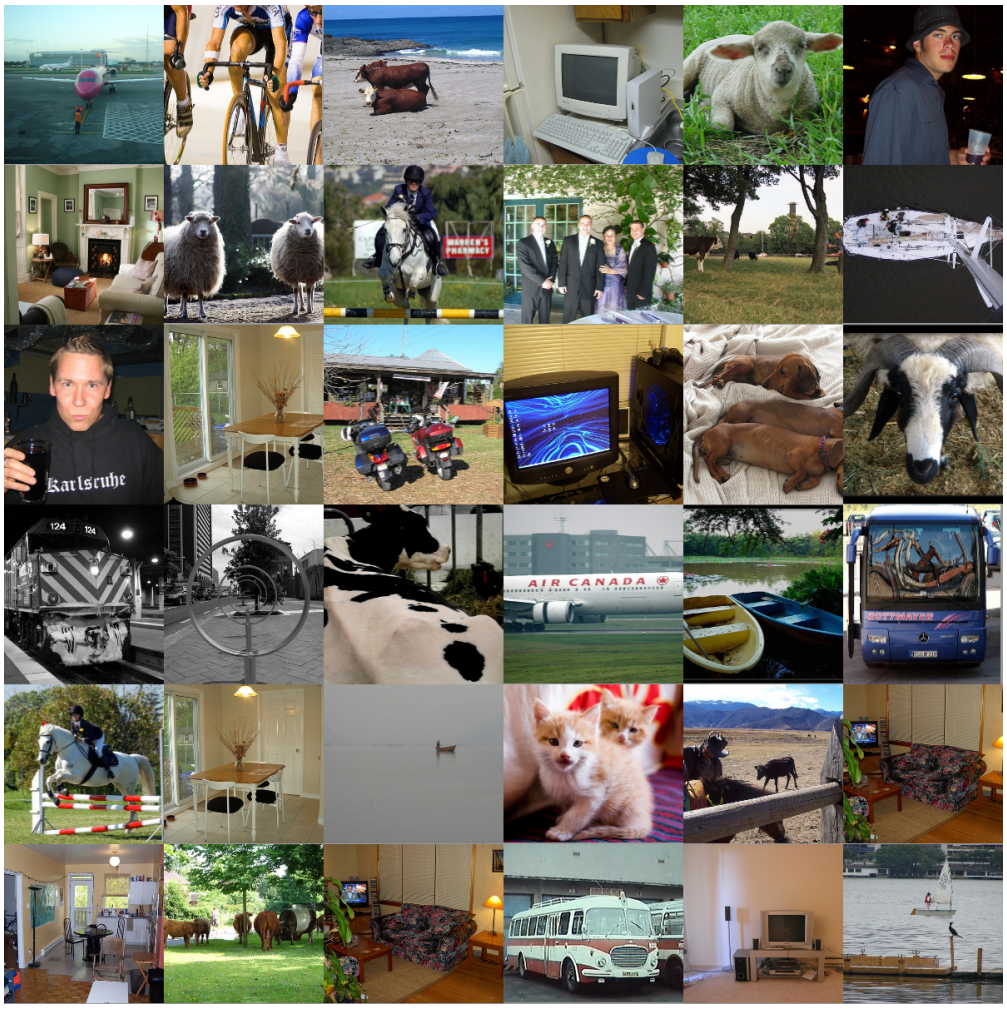


Figure 4.1: Examples of images from the PASCAL VOC dataset. Adapted from [EVGW+]

A subset of images was selected randomly to separate into training and test datasets. For the training dataset, initially, a small subset with 10 images and later with 100, 1000, 2000, and 5000 images were selected and investigated in experiments. As mentioned earlier, SIFT and SURF are used to obtain keypoints and their information for a given set of images. The OpenCV implementations are used for both SIFT and SURF. Figure 2.9 shows an example of applying the SIFT algorithm with default parameters to an image.

It can be seen in Figure 4.2 that there is a massive imbalance in the scales of detected keypoints. The number of smaller keypoints exceeds scales that convey meaningful information. Figure 4.3 shows the histogram for 100 images, and it can be seen that the imbalance increases exponentially when more images are considered. In order to have a balanced distribution of different scaled keypoints, a filtering approach is employed to create the training dataset.

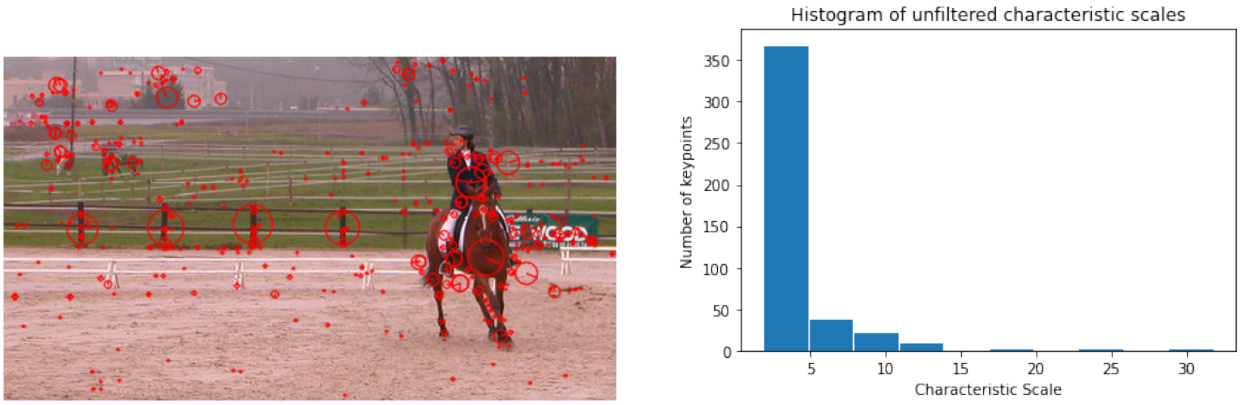


Figure 4.2: Unfiltered SIFT keypoints (left) and histogram of unfiltered characteristic scales (right)

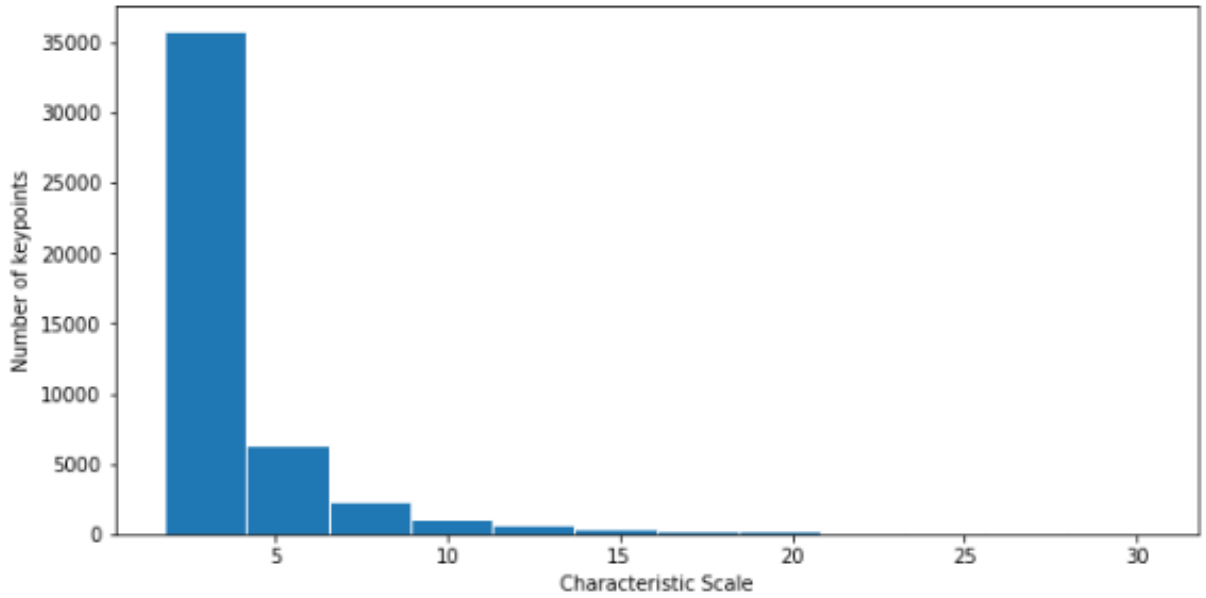


Figure 4.3: Histogram of unfiltered characteristic scales for 100 images

Histogram-based filtering is applied to the obtained keypoints for a given image. First, a histogram of all the keypoints is obtained according to their characteristic scales. The histogram divides the keypoints into different bins depending on the scale values, which depicts the imbalance shown on the right of Figure 4.2. In order to have a balanced set of keypoints, 20 keypoints from every bin are selected randomly. If there are bins with less than 20 keypoints, some keypoints are repeated to have an evenly spaced distribution of scales among all keypoints. It is important to note that not all scales are available for an image, and some bins can remain unfilled. Figure 4.4 depicts the filtered keypoints on the left and a histogram of keypoints after applying the filtering operation on the right. This filtering

operation is applied to every image, and the filtered keypoints are accumulated to create the training dataset.

Figure 4.4: Filtered SIFT keypoints (left) and histogram of filtered characteristic scales (right)

Along with the characteristic scales, the following information are stored for each keypoint: the position in the image plane (coordinates), characteristic orientation, and the response specifying the keypoint strength. Information other than the position and the corresponding characteristic scale is only stored for the sake of completeness. A JSON (JavaScript Object Notation) file [PRS+16] is created that contains all the above information for the filtered set of keypoints for the entire subset of training images. The advantage of using JSON compared to CSV to store information is that it allows for more accurate and precise data handling, especially when handling and storing data with different data types [GSGK22]. In the context of the dataset creation in this work, since the coordinates are a `tuple` and the characteristic orientation, the characteristic scales, and the response specifying the keypoint strength are `float` data type, it is convenient to store the dataset in JSON.

In order to have a systematic analysis of the performance of trained networks, a test dataset is created. For a subset of 100 images different from those selected for training, 1000 keypoints are extracted and selected randomly. It is important to note that no filtering is applied to create the test dataset. The keypoints are sorted and further divided according to their scales into 3 subsets: *test_small*, *test_medium* and *test_large*. Dividing into subsets allows for systematic analysis of the performance of a trained network to different-sized scales.

4.1.2 Data Pre-processing

In order to prepare the patches and the corresponding characteristic scales for training, a custom data generator was implemented from the JSON dataset of training images. An

advantage of using data generators is that they allow the processing of large datasets without loading them entirely into memory. Due to the manner of the created JSON dataset, commonly used data generators from Tensorflow and Sci-kit cannot be used. Custom data generators enable customization of the data loading process, such as resizing images, changing the batch size, and shuffling the data, allowing data optimization for the specific model architecture and training objectives [Mur21]. The created JSON dataset is loaded using the function `pandas.read_json` from Pandas library [McK10], and the training dataset is further split into *training*, and *validation* sets in the ratio 75% : 25%. The functions of the custom data generator for a specified batch size are as follows:

1. Inputs the dataframe of training/validation sets.
2. Searches for the path of the image, which has the detected keypoint.
3. Reads the image, collects the coordinates of the keypoint, and extracts a patch centred around the keypoint for the provided input size for the patch.
4. The extracted patch is normalized between 0 and 1 by dividing all the pixels by 255.
5. Collects the characteristic scale from the dataframe corresponding to the keypoint.
6. Groups the extracted patch and the scale and prepares batches of (data, label) pairs.

Depending on the provided batch size, the output of the data generator is the pre-processed batches of data used to train the network. The created dataset is named "SIFT-KP5000", which indicates that the keypoints are extracted for 5000 images using SIFT on PASCAL dataset images. Similarly, the "SURF-KP5000" dataset indicates the keypoints extracted using SURF.

4.2 System Overview

As described earlier in Chapter 1.2, the developed framework estimates the characteristic scales of local image features. The inputs to the framework are real images with keypoints, and the outputs are the characteristic scales of the keypoints. Figure 4.5 shows an overview of the developed framework, which offers flexibility in interchangeably using different feature detectors and network architectures.

As described in Section 4.1.2, once the dataset is created, the data has to be prepared such that it can be utilized to train neural network architectures. In order to do so, a

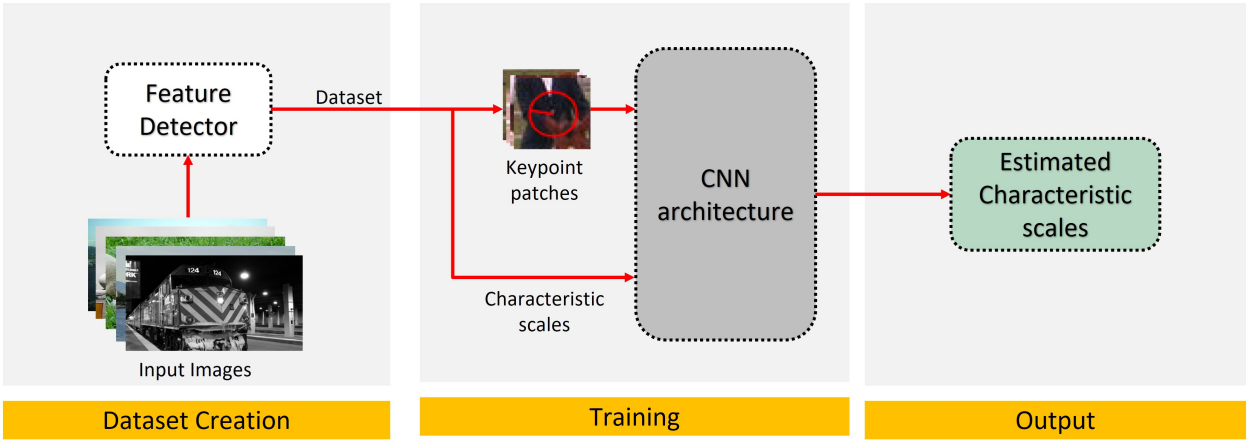


Figure 4.5: Overview of the framework to estimate characteristic scales

square region (patch) centred on a keypoint is extracted by the custom data generator. The patch is extracted for every keypoint in the dataset, and such a collection of patches form the input to the CNN. From the dataset, each keypoint is associated with a corresponding characteristic scale. This scale is a continuous value assigned as a label for the keypoint patch. The extracted patch and the corresponding scale form a *(data, label)* pair to train a CNN architecture. Figure 4.6 shows the extraction of patches around keypoints to form a collection.

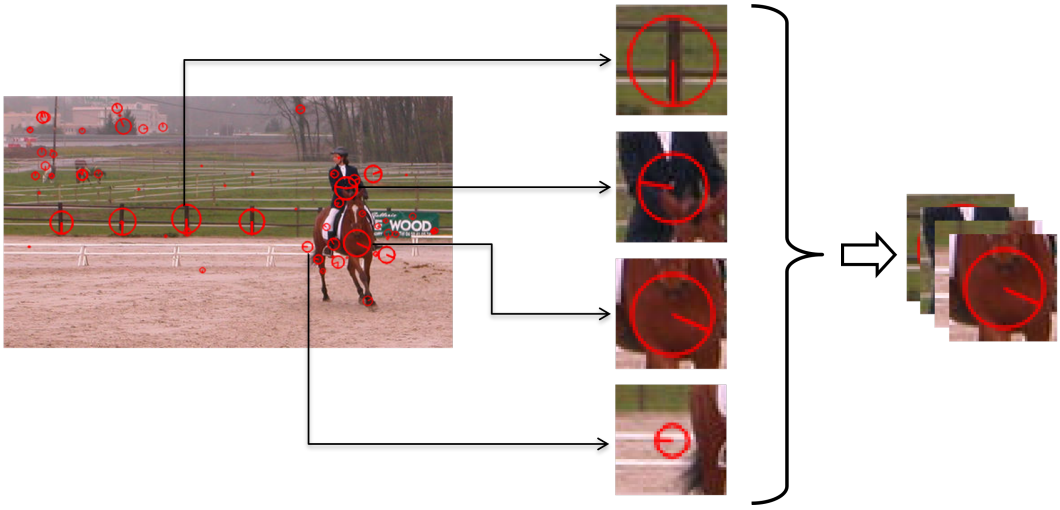


Figure 4.6: Examples for keypoint patches

Extraction of Keypoint Patches

Several factors were taken into consideration regarding the size of the patch being extracted around a keypoint. These factors are briefly discussed in the following subsections.

Fixed vs Variable Patch Size

A fixed-sized patch or a patch proportional to the scale of the keypoint can be extracted. However, it is beneficial to train a CNN with fixed-sized inputs. Fixed-sized patches are computationally efficient to extract and process since the patch size is predetermined and does not depend on the scale or location of the keypoints. This also helps in increasing the robustness of the network estimating scales. The most significant disadvantage of using a variable patch size is that during the estimation of scale for a keypoint, the region's size as input to the trained network is unknown, which is more likely to result in a wrong estimation. Another disadvantage concerns handling patch sizes proportional to keypoints with small scales. Most of the architectures investigated in this work required input images of a minimum size to learn, and hence small patches could not be used. In order to mitigate these disadvantages, fixed-size patches were used in this work.

Boundary Keypoints

A special case is keypoints near the image boundary. The extraction of patches around these keypoints would result in a patch of varying size and a scenario of uneven patches in the data. To mitigate this problem, every input image is extended around the border using `cv2.copyMakeBorder()` from OpenCV with a border type of `cv2.BORDER_REPLICATE`, which replicates pixels around the image border. By carrying out a border extension, fixed-sized patches can be extracted around all the keypoints on and around the border of an input image.

Table 4.1 provides an overview of the different input dimensions used in the CNN-based keypoint detection methods from the literature to determine an appropriate patch size for extracting the region around a keypoint. For instance, LIFT from [YTLF16] selected a 64×64 patch for the Orientation Estimator (O.E) and Detector stages in the pipeline, whereas a 128×128 patch was chosen for computing the descriptor patch. On the other hand, LF-Net from [OTFY18] chose a 32×32 patch to compute the descriptor. Key.Net, from [LRPM19], utilized 192×192 images derived from the ImageNet ILSVRC 2012 dataset. However, no specific explanation is provided for choosing a particular size. Some architectures, such as Key.Net, utilized entire images with an unmodified size from the selected dataset. In contrast, specific standard pre-trained CNN architectures, such as VGG and ResNet, require the input size to be $(height \times width \times channels) = (224 \times 224 \times 3)$.

Architecture	Input Size
LIFT [YTLF16]	128×128 for Descriptor (Patch) 64×64 for O.E and Detector (Patch)
LF-Net [OTFY18]	32×32 for Descriptor (Patch)
Key.Net [LRPM19]	192×192 for Siamese pipeline (Image)

Table 4.1: An overview of the input dimensions used in the architectures from literature

As there is no conclusive reasoning in the literature for selecting a particular patch size, this work utilizes a statistical approach for the created dataset to determine a suitable patch size. The starting point for selecting a size was to compute the average of all the characteristic scales in the dataset. This approach ensures that the selected size includes keypoints with more meaningful information. The subsequent experiments and results obtained by varying the patch size around the initially selected size are discussed later in Chapter 5.

4.3 Network Architectures

This section discusses the different network architectures adapted and implemented in this work. To begin, an overview of various network architectures used in the literature, as discussed earlier in Chapter 3, is presented in Table 4.2. LIFT uses different networks for its three stages. For the Descriptor, it employs three CNN layers, while spatial transformers are used to learn the orientations, and a CNN layer combined with a piecewise linear activation function is used for the Detector. In LF-Net, a structure with a ResNet layout, including three ResNet blocks, is used. In Key.Net, training is carried out in a Siamese pipeline. It is important to note that these architectures are trained for either obtaining a feature map with localized keypoints or feature maps used to construct a scale-space analysis to localize keypoints. However, with modifications to fit the data in this work, these network architectures can be a good starting point for estimating characteristic scales.

Architecture	Network
LIFT [YTLF16]	3 CNN Layers (Descriptor) Spatial transformers (Orientation estimator) 1 CNN Layer + Piecewise linear activation (Detector)
LF-Net [OTFY18]	ResNet layout with three blocks
Key.Net [LRPM19]	Siamese architecture (A convolutional layer with 8 filters)

Table 4.2: An overview of network architectures from literature

In this work, several network architectures have been adapted and modified from the literature and standard architectures, VGG and ResNet, mainly by changing the activation function used in the output layer. Instead of the typical `softmax` for classification tasks, this work uses a `linear` activation function that outputs continuous values. This modification was necessary because the scales to be estimated are not predefined classes as in classification tasks but linear, continuous values. In the following subsections, different architectures adapted and implemented in this work have been discussed, and a summary of the chosen architectures is provided.

4.3.1 VGG-16 Architecture

The VGG architecture is a deep convolutional neural network developed by the Visual Geometry Group (VGG) at the University of Oxford in the 2014 ImageNet Large Scale Visual Recognition Challenge (ILSVRC). It is commonly used in computer vision tasks such as object recognition, image classification, and feature extraction. Its simplicity and effectiveness have made it a popular benchmark model for evaluating the performance of new deep-learning architectures [SZ15].

There are several variants of the VGG architecture, such as VGG11, VGG13, VGG16, and VGG19. Among these variants, the most commonly used is the original VGG16 architecture. Figure 4.7 shows the VGG16 architecture. The number “16” here indicates 16 main layers, consisting of 13 convolutional layers and 3 fully connected layers. It is important to note that a standard VGG16 architecture requires a fixed input image dimension of $224 \times 224 \times 3$. In common cases, a pre-trained VGG16 architecture is used, but in this work, only the network

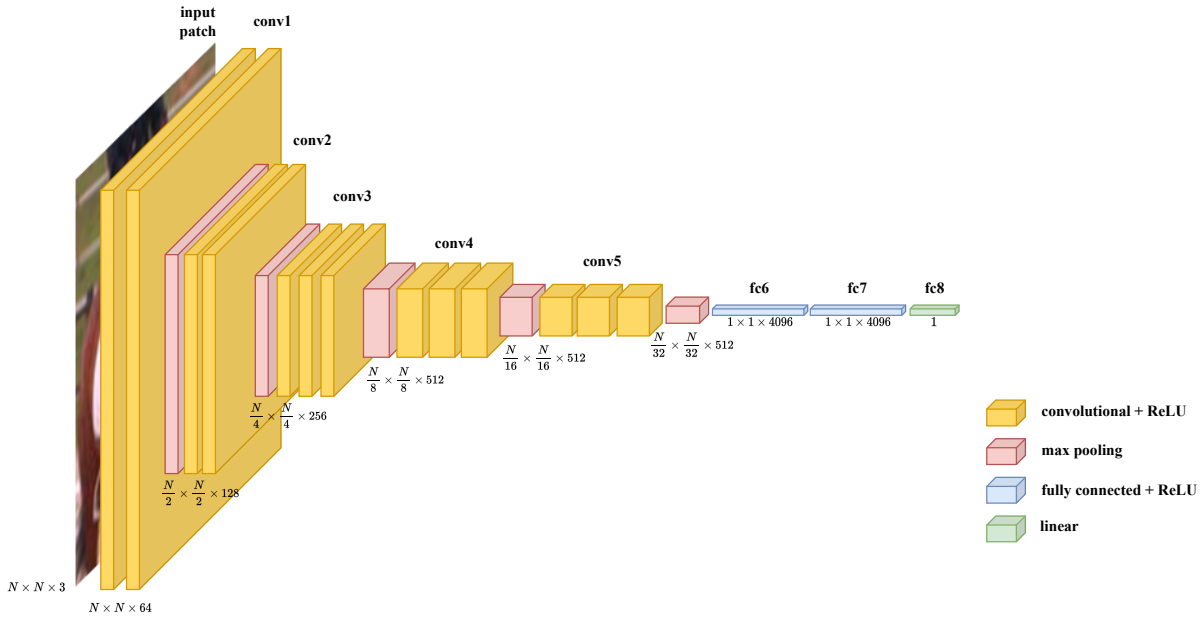


Figure 4.7: VGG16 Architecture, adapted from [SZ15]

structure is being adapted due to the large fixed size. However, extracting such large patches around keypoints in an image is not feasible, especially for keypoints with relatively small characteristic scales. Hence, $N \times N$ indicates the patch size, and several patch sizes have been investigated in this work.

4.3.2 Mod-VGG16 Architecture

An architecture adapted from the standard VGG16 architecture [SZ15] is implemented in this work as "Modified VGG16" (Mod-VGG16) architecture. It has the same number of layers but reduced channel widths for the convolutional layers. Figure 4.8 shows the Mod-VGG16 architecture. The reasoning for the choices is summarized in Section 4.3.4.

4.3.3 ModRed-VGG Architecture

Another architecture adapted and implemented in this work is the "Modified-and-Reduced VGG" (ModRed-VGG) architecture, also adapted from the standard VGG16 architecture [SZ15]. It is a smaller network with reduced channel widths for the convolutional layers and a single FC layer. Figure 4.9 shows the ModRed-VGG architecture.

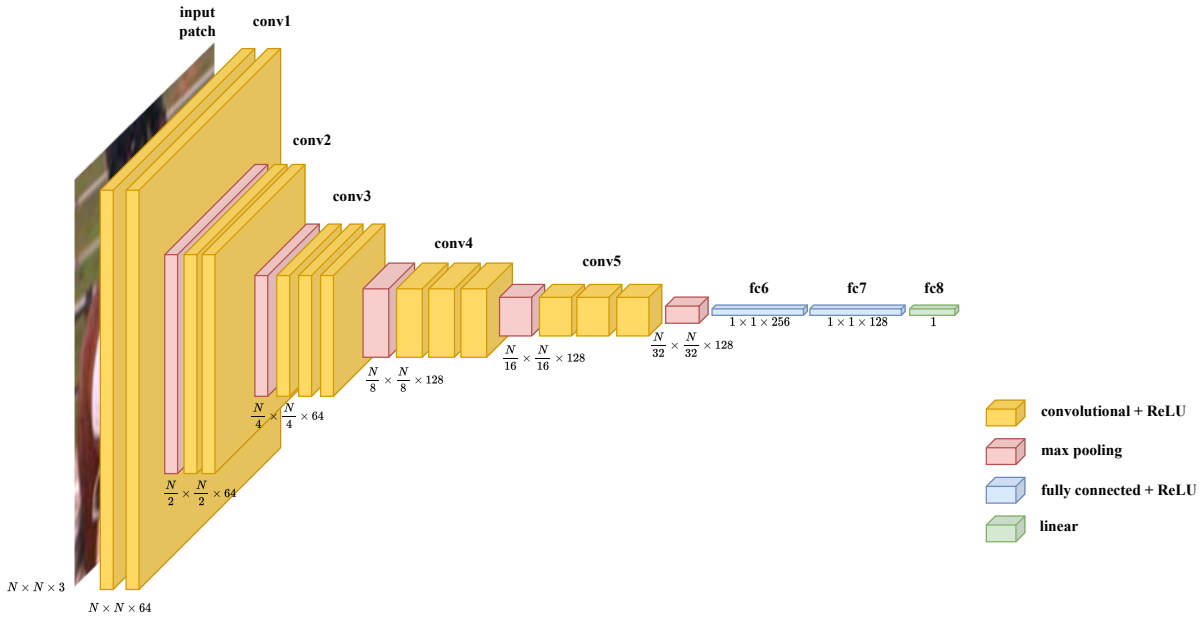


Figure 4.8: Mod-VGG16 architecture, adapted from [SZ15]

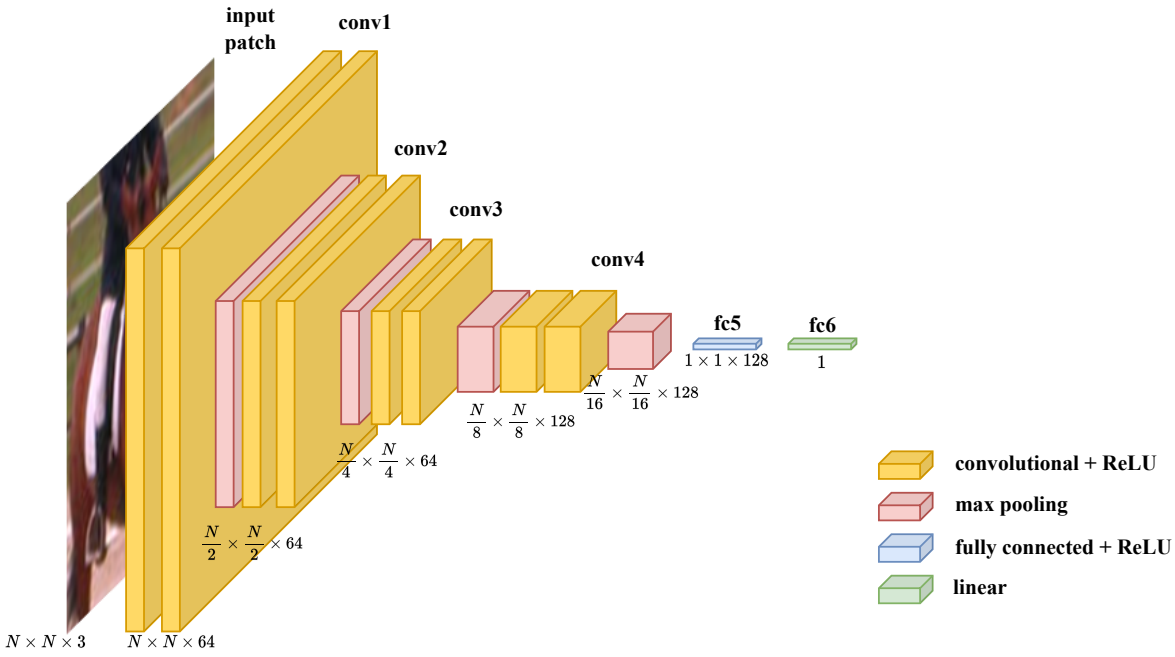


Figure 4.9: ModRed-VGG architecture, adapted from [SZ15]

4.3.4 Mod-ResNet34 Architecture

In 2015, Microsoft researchers developed an architecture for deep convolutional neural networks called *ResNet*. It uses residual connections to enhance deep neural network training and performance. As explained in Chapter 2.3.4, residual connections enable the network

to learn residual functions, which are the variations between a block's input and output. Similar to VGG architectures, ResNet has several variations, where the main difference is the total number of convolutional layers. ResNet18, ResNet34, ResNet50, and ResNet101 are a few to name. State-of-the-art performance on various computer vision tasks, including image classification, object detection, and semantic segmentation, has been attained with the ResNet architectures [WYW+19].

Figure 4.10 shows the Mod-ResNet34 architecture, which has been adapted and modified from the standard ResNet34 architecture. From Figure 4.10, the dotted and non-dotted lines indicate different connections between layers. Specifically, the non-dotted lines represent the *identity* or *shortcut* connections that skip over one or more layers. These connections are added to improve the flow of information through the network and help prevent vanishing gradients during training. On the other hand, the dotted lines represent the convolutional layers that are applied to the shortcut connections. These layers adjust the dimensions of the shortcut connections such that they can be added to the output of the residual block. The use of convolutional layers in this way is known as *projection shortcut* that ensures that the shortcut connections have the same dimensions as the output of the residual block, which makes it easier to add the two together [HZRS16].

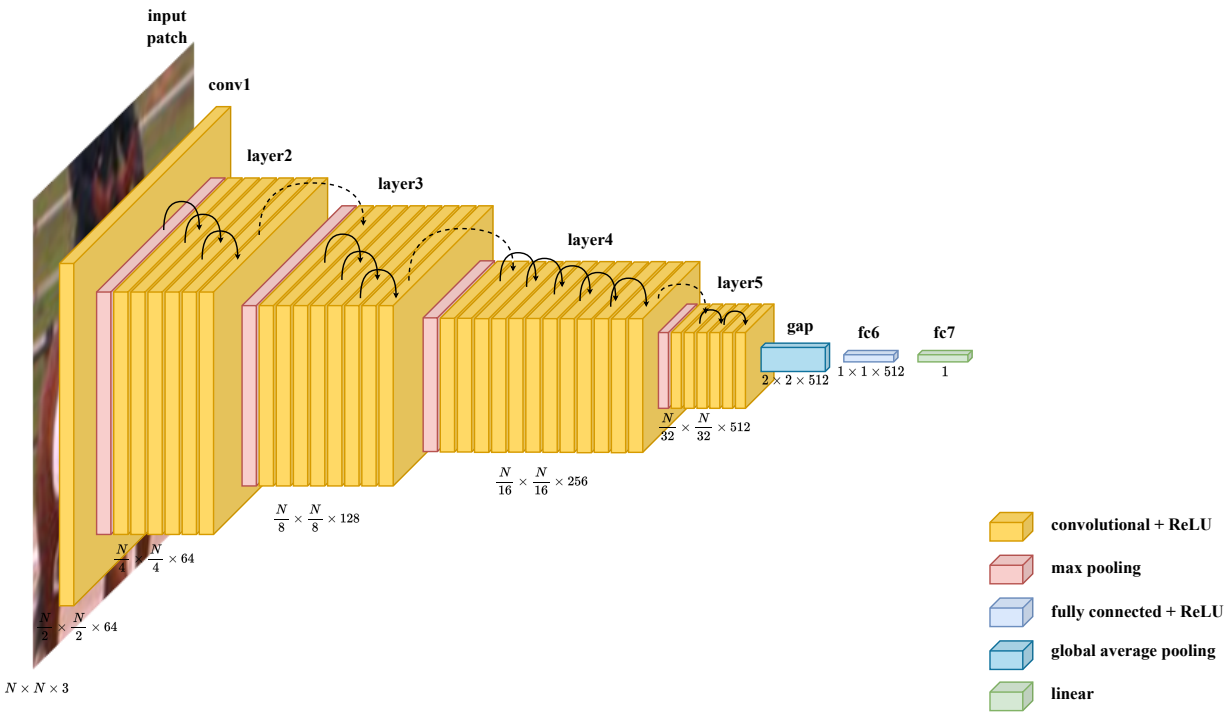


Figure 4.10: Mod-ResNet34 architecture adapted from [HZRS16]

Summary on Architecture Selection

Preliminary experiments with the standard VGG16 architecture resulted in *underfitting* as explained in Section 4.3.4. From Table 4.3, which shows the number of parameters in different network architectures, it can be seen that the standard VGG16 architecture has 28.88M parameters. One reason for underfitting is due to a large number of parameters and deeper channel widths of the CNN layers present in the architecture. Another reason is that the standard pre-trained VGG16 is trained on fixed input images of $224 \times 224 \times 3$ compared to the relatively smaller patch widths used in this work. In comparison, a Modified VGG16 (Mod-VGG16), as described in the Subsection 4.3.2, has the same number of layers as the standard VGG16 but with narrower channel widths for the convolutional layers. Another architecture was adapted from the standard VGG16 with reduced layers and narrower channel widths, Modified-and-Reduced VGG (ModRed-VGG), as described in Subsection 4.3.3. It can be seen that the adapted architectures have significantly reduced number of parameters, which helps to improve training process. Another commonly used standard architecture, ResNet34, was adapted and implemented in this work as Mod-ResNet34. The results of training these architectures are documented in Chapter 5.

Network	Total parameters	Trainable parameters
VGG16	28,882,765	28,882,759
Mod-VGG16	1,007,169	1,007,169
ModRed-VGG	777,933	777,415
Mod-ResNet34	21,569,793	21,554,561

Table 4.3: Number of parameters of the network architectures

5 Experiments and Results

This chapter describes the experiments conducted in this work and presents the results and corresponding analyses. Section 5.1 explains the experimental design and outlines the network architectures investigated. Section 5.2 discusses the accuracy metric used in this work and presents the results in both quantitative and qualitative terms. Additionally, Section 5.3 briefly introduces another approach using the inherent multi-scale nature of CNNs. It is important to note that the characteristic scale is defined in pixels, and the same unit applies to the loss function and metrics used.

5.1 Experiments

This section documents the various experiments using the network architectures "Mod-VGG16", "ModRed-VGG", and "Mod-ResNet34" described in Chapter 4.3. Sections 5.1.1 and 5.1.2 discuss the experiments using SIFT-KP5000 and SURF-KP5000, respectively.

5.1.1 Experiments with SIFT keypoints

The dataset SIFT-KP5000 created in this work is used to train all the network architectures. From preliminary experiments, the standard VGG16 architecture fails to perform well, as explained earlier in Section 4.3.4. Hence, only the variants of VGG16 adapted and modified are implemented in this work. The characteristic scale is the radius of the blob-like keypoint from SIFT. The initial patch size is selected by determining the average of all the characteristic scales in SIFT-KP5000, which is determined to be 29 pixels and the default patch size, which contains the entire keypoint for the input images to the network, is 58×58.

Experiment A

Table 5.1 summarizes the network architectures implemented for a fixed set of parameters in Experiment A. It can be seen from the table that all network architectures are trained with the same configuration of the loss function, optimizer, and patch size. The performance of a specific network was evaluated by observing the RMSE on the validation and the test dataset. Furthermore, the best-performing network from Experiment A is chosen as the baseline for the following experiments, where different parameters are varied, and the network's overall performance is observed. Additionally, the number of training epochs is listed for each network in Tables 5.1 and 5.2. For the epochs, the already explained early-stopping procedure is used.

Network	Loss Function	Optimizer	Input Patch Size	Epochs	RMSE	
					Validation	Test
Mod-VGG16	MAE	RMSprop	58×58	100	3.36	8.13
ModRed-VGG	MAE	RMSprop	58×58	100	3.35	8.30
Mod-ResNet34	MAE	RMSprop	58×58	100	**2.20**	**7.35**

Table 5.1: Summary of different network architectures trained in Experiment A

Experiment B

To analyze the impact of specific parameters on the overall performance of the network, the effects of varying the loss function, optimizer, and input patch size are investigated. In Experiment B, two loss functions, *MAE* and *MSE* are evaluated and determined that **MAE** marginally outperforms MSE in terms of RMSE on both the validation and test datasets. Therefore, the loss function is fixed to MAE for subsequent experiments. Next, three optimizers (*Adam*, *SGD*, and *RMSprop*) are tested and found that **RMSprop** yields the best results, and thus is selected for the next set of experiments. Finally, the input patch sizes are varied around an initial size of 58×58, and it turns out that larger sizes generally improve performance. Specifically, a patch size of $\mathbf{88 \times 88}$ achieves the best RMSE on both the validation and test datasets.

Table 5.2 presents the results of Experiment B, where the baseline network from Experiment A is trained with different parameter combinations. The best-performing network from Experiment B, shown in Table 5.3, is selected for further quantitative and qualitative analyses. The learning curves for the loss and RMSE are displayed in Figure 5.1. The plots show

that after approximately 40 epochs, there is only marginal improvement in both the loss and RMSE. The early-stopping procedure is employed, and the network is saved at the best checkpoint after 107 epochs.

Network	Loss Function	Optimizer	Input Patch Size	Epochs	RMSE	
					Validation	Test
Mod-ResNet34	**MAE**	RMSprop	58 × 58	100	**2.20**	**7.35**
	MSE			100	2.25	7.55
	MAE	Adam	58 × 58	100	2.25	7.46
		SGD		100	2.24	7.43
		RMSprop		100	**2.20**	**7.35**
	MAE	RMSprop	48 × 48	100	2.93	9.73
			58 × 58	100	2.20	7.35
			78 × 78	100	2.12	6.91
			88 × 88	107	**1.99**	**6.32**

Table 5.2: Summary of Mod-ResNet34 trained for different parameter configurations in Experiment B. The highlighted cells imply the best-performing parameter and corresponding RMSE values.

Network	Loss Function	Optimizer	Input Patch Size	Epochs	RMSE	
					Validation	Test
Mod-ResNet34	MAE	RMSprop	88 × 88	107	1.99	6.32

Table 5.3: Best-performing Mod-ResNet34 configuration

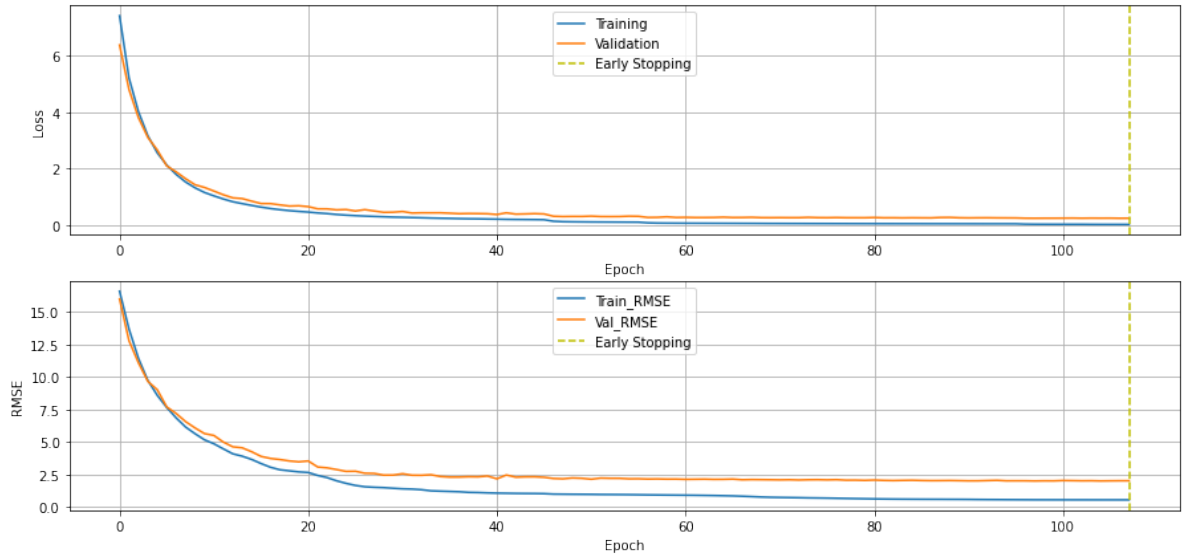


Figure 5.1: Learning curves for Mod-ResNet34

5.1.2 Experiments with SURF keypoints

Similar to SIFT-KP5000, for the same 5000 images, SURF is applied, and a new training dataset is created as "SURF-KP5000". The network architecture with the best-performing parameter configuration from the previous Experiment B was selected for this experiment. Table 5.4 shows the network architecture with the obtained RMSE on the validation and training datasets.

Network	Loss Function	Optimizer	Input Patch Size	Epochs	RMSE	
					Validation	Test
Mod-ResNet34	MAE	RMSprop	168×168	100	6.65	15.78

Table 5.4: Mod-ResNet34 configuration on SURF-KP5000 dataset

5.2 Evaluation

This section describes the accuracy metric implemented in this work and discusses the performance of the Mod-ResNet34 network in quantitative and qualitative terms.

5.2.1 Evaluation Metric

In order to evaluate the performance of a trained network quantitatively, an accuracy metric to assess the networks in this work is implemented. Since the estimated characteristic scales are continuous values, typically used metrics to evaluate networks for the classification problem, such as Precision, Recall, F1-score, and ROC, cannot be used in this context.

Accuracy

A continuous value can be classified as correct using a threshold value, τ. It indicates the allowable error between the ground-truth, s_i and estimated scales, $\hat{s_i}$, for every keypoint i. A threshold_factor is computed by multiplying the threshold value, τ, with the ground-truth scale, s_i. The absolute difference between the two scales must be below the threshold_factor to be considered a correct instance. By varying the threshold value, the performance of a network can be evaluated over a range of threshold values. Equation 5.1 shows the computation of accuracy:

For a given threshold τ,

$$\text{Accuracy}_{overall} = \frac{1}{N}\sum_{i=1}^{N}[|s_i - \hat{s_i}| < (\tau \cdot s_i)] \tag{5.1}$$

where N is the total number of keypoints, s_i and $\hat{s_i}$ are the ground-truth and estimated scales, for a keypoint i, respectively, and $\tau \in [10\ \%, 15\ \%, 20\ \%]$.

Several metrics commonly used for evaluating regression networks include Mean Absolute Error (MAE), Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and R-squared [LMAPH19]. In this work, MAE and MSE are used as loss functions to minimize predictive error, while RMSE is used during training to monitor network performance. The epoch at which RMSE reaches its minimum is observed, and the network with its weights and parameters at that epoch is saved for further analysis.

5.2.2 Quantitative Analysis

Table 5.5 presents the results for the Mod-ResNet34 network trained on SIFT-KP5000 and evaluated on the test dataset. The table shows the accuracy for three error thresholds, namely 10 %, 15 %, and 20 %. As expected, the accuracy tends to improve as the allowable error threshold increases. It is interesting to note that the extent of improvement is not uniform across all keypoints. In particular, small-scaled keypoints are penalized more than others due to their size. This penalty is evident in the low accuracy for test_small at an error threshold of 10 %, which is considerably lower than those for test_medium and test_large. However, as the error threshold increases to 15 % and 20 %, the accuracy for test_small shows a significant increase of 18.32 % and 30.03 %, respectively. In contrast, the improvement for test_medium and test_large is marginal.

Error Threshold (%)	Accuracy test_small (%)	Accuracy test_medium (%)	Accuracy test_large (%)	Overall Accuracy (%)
10	39.34	81.98	79.04	66.78
15	57.66	93.69	89.22	80.19
20	**69.37**	**95.80**	**92.51**	**85.89**

Table 5.5: Accuracy at different thresholds for Mod-ResNet34 trained on SIFT-KP5000

Table 5.6 presents the results for the Mod-ResNet34 network trained on SURF-KP5000 dataset and tested on a separate test dataset generated using SURF keypoints. Notably, the overall accuracy is higher than the results obtained for the SIFT-KP5000 dataset. Moreover, there is a significant improvement in the accuracy for small-scaled keypoints at an error threshold of 10% in the case of SURF keypoints as opposed to SIFT keypoints. This improvement can be attributed to the larger average scale of SURF keypoints, **15.69**, compared to 3.02 for SIFT keypoints in test_small. This suggests that a larger average scale of keypoints may enable the network to capture more relevant information from the region surrounding the keypoint, leading to better performance.

Error Threshold (%)	Accuracy test_small (%)	Accuracy test_medium (%)	Accuracy test_large (%)	Overall Accuracy (%)
10	64.26	64.86	85.33	71.48
15	73.27	79.28	94.01	82.19
20	**80.18**	**87.69**	**96.71**	**88.19**

Table 5.6: Accuracy at different thresholds for Mod-ResNet34 trained on SURF-KP5000

Overall, the Mod-ResNet34 network trained with SURF-KP5000 marginally performs better than the same architecture trained with SIFT-KP5000 by a 2.3% improvement in accuracy.

5.2.3 Qualitative Analysis

This section discusses results by selecting and analyzing specific keypoints from each subset of the test dataset. The idea is to visualize and interpret the outputs of intermediate activation layers as heatmaps of the trained Mod-ResNet34 network when a keypoint patch is provided as an input to estimate the characteristic scale. The Mod-ResNet34 has 33 activation layers, and a few of the most interesting layer outputs are selected to identify any discernible patterns and gain insights. Since the dimensions of these intermediate layers decrease as the network gets deeper, all the heatmaps are upsampled to the input patch size 88×88 and normalized between 0 and 1. This section discusses only the results from the Mod-ResNet34 architecture trained with SIFT-KP5000. The following figures in this section show the ground-truth scale represented by a red circle, while the predicted scale is represented by a blue circle. It is important to note that the terms “predict” and “estimate” are used interchangeably in this work.

Case Study 1: test_small

Figures 5.2 and 5.3 show examples from the test_small with incorrectly estimated characteristic scales. The keypoint in Figure 5.2 is located near the edge of the image, making it a special case. By examining the heatmaps in 5.2, it is difficult to identify any clear pattern that would allow for a definitive conclusion about scale estimation. One possible explanation is that the keypoint region is too small to be of significant importance to the network, instead focusing on the overall patch and the stronger responses at the edges. Similarly, in Figure 5.3, the keypoint region lacks distinctive structures, which may be another plausible explanation for the incorrect estimation. It is also possible that the quality of the keypoint region itself is a factor in the network's performance, not just the network architecture.

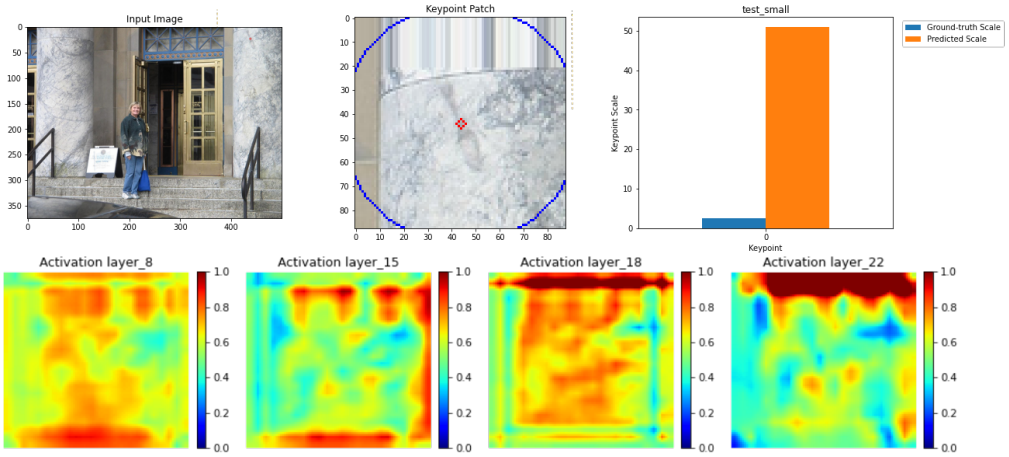


Figure 5.2: Incorrect scale-estimation from test_small (1)

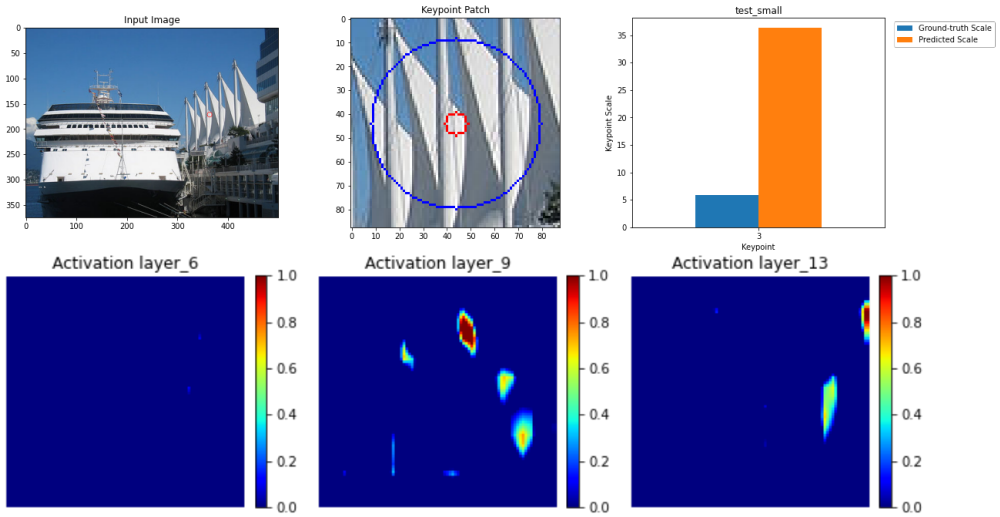


Figure 5.3: Incorrect scale-estimation from test_small (2)

Figures 5.4 and 5.5 show correctly estimated characteristic scales from the test_small. The keypoint regions in these examples are small and appear inadequate for the network to make accurate estimations, and the heatmaps reveal that the network also identifies patterns outside the keypoint region in the patch. This suggests that the network may consider distinguishable structures in addition to the keypoint region when estimating the scales. In Figure 5.5, the patch appears noisy, and the keypoint region does not provide useful information for accurate estimation. Examining the heatmaps, it is noticeable that the network responds more strongly to regions with relatively less noise, which may account for the correct estimation.

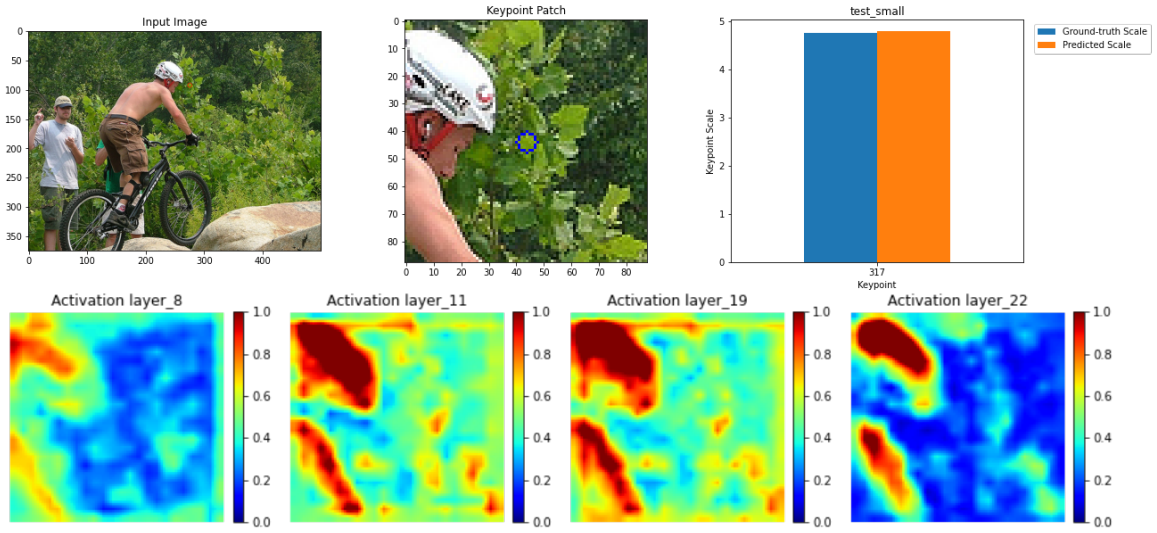


Figure 5.4: Correct scale-estimation from test_small (1)

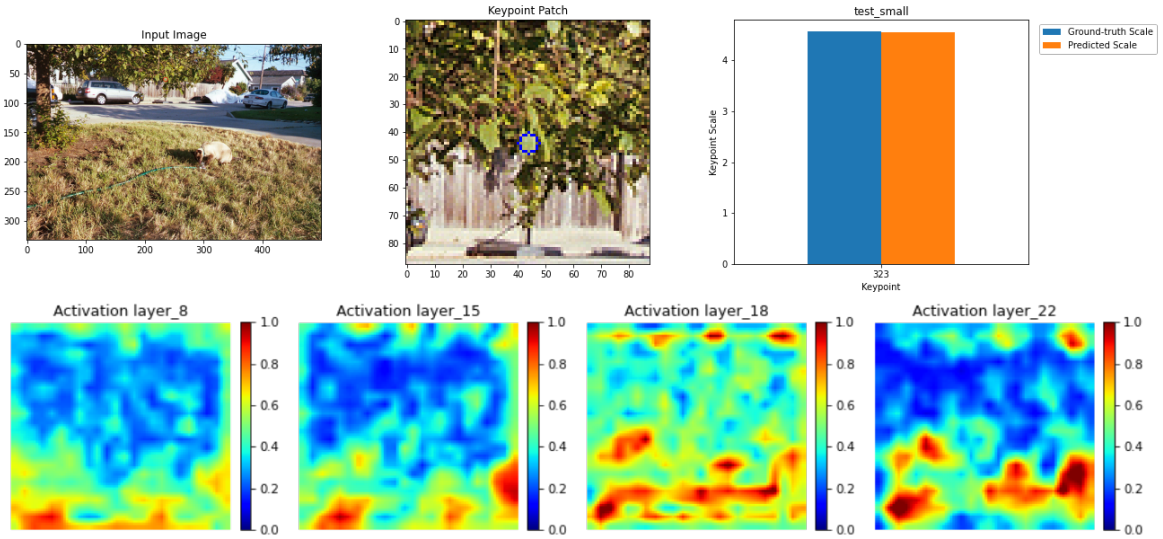


Figure 5.5: Correct scale-estimation from test_small (2)

Case Study 2: test_medium

Figures 5.6 and 5.7 show examples from the test_medium with incorrectly estimated characteristic scales. The heatmaps in both examples exhibit a strong response towards the top portion of the patch, which could be due to the distinct texture difference in that region. In Figure 5.7, there is some response around the keypoint, but it is insignificant compared to the strong response near the edges. In some cases, it can be challenging to analyze the network's behaviour, such that no conclusive explanation can be provided for the incorrect estimation.

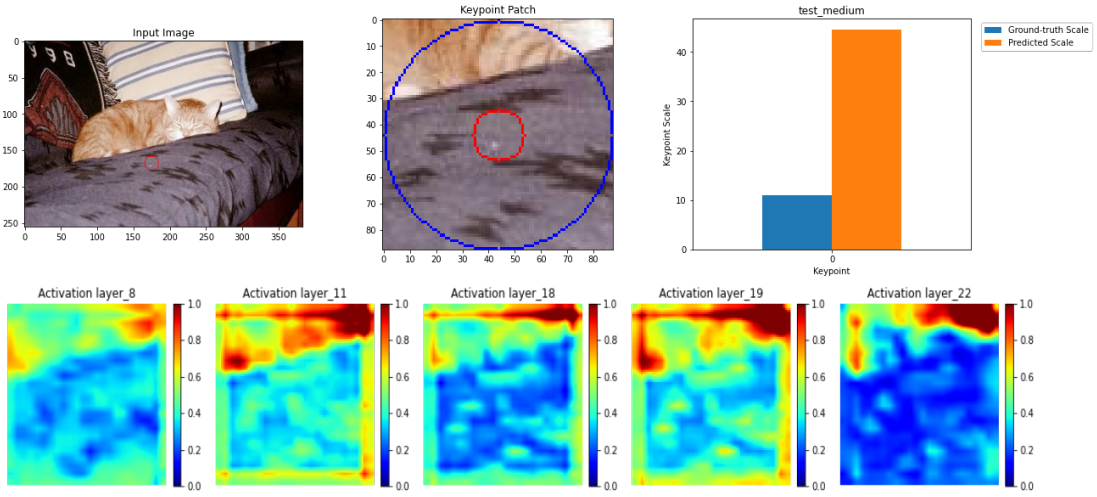


Figure 5.6: Incorrect scale-estimation from test_medium (1)

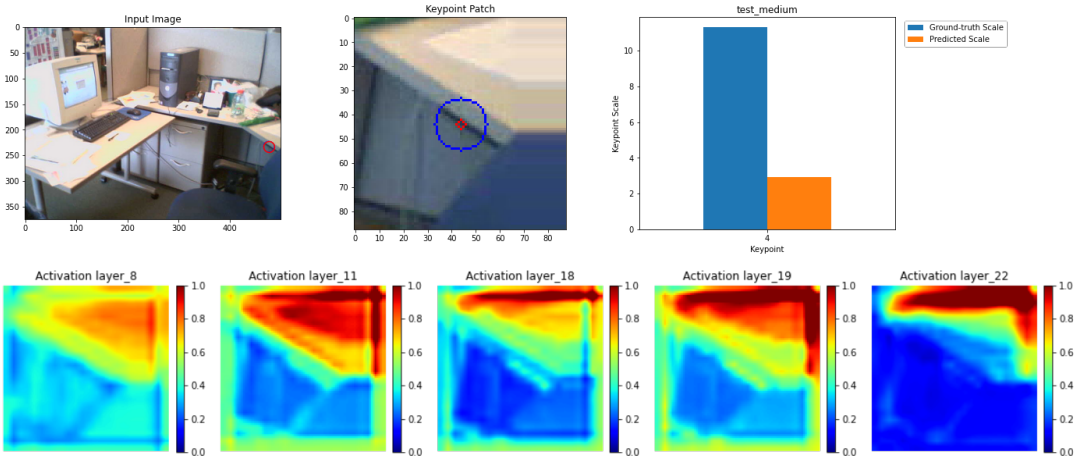


Figure 5.7: Incorrect scale-estimation from test_medium (2)

Figures 5.8 and 5.9 show examples of correctly estimated characteristic scales from the test_medium. The heatmaps in Figure 5.8 indicate that the network identifies patterns within and outside the keypoint region. This behaviour suggests that there are discernible regions in the patch that the network utilizes to make correct predictions. On the other hand, in Figure 5.9, the network mainly relies on the region surrounding the keypoint to make the correct prediction. This suggests that in some cases, the keypoint region contains the most discernible information from the entire patch, and the network may only need to focus on that region. Furthermore, the keypoint in Figure 5.9 is more exposed than the remaining patch region, which could have contributed to the correct prediction.

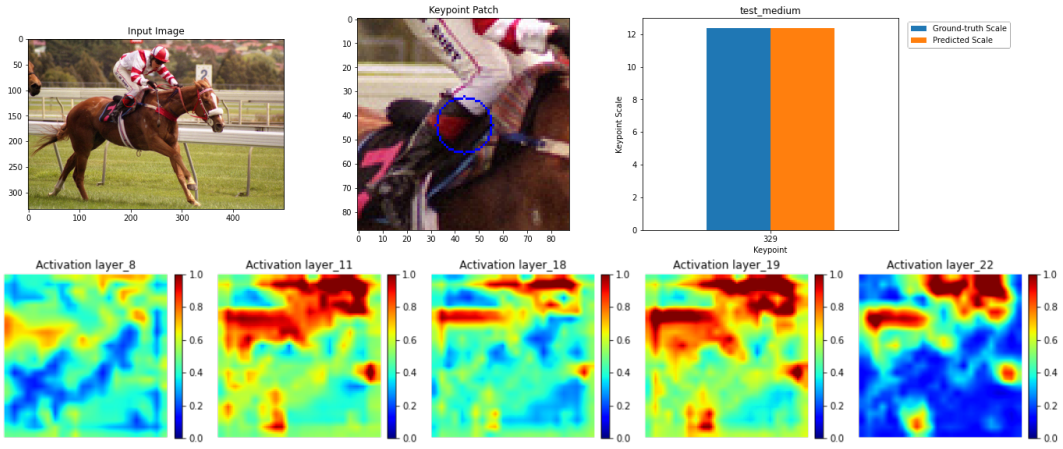


Figure 5.8: Correct scale-estimation from test_medium (1)

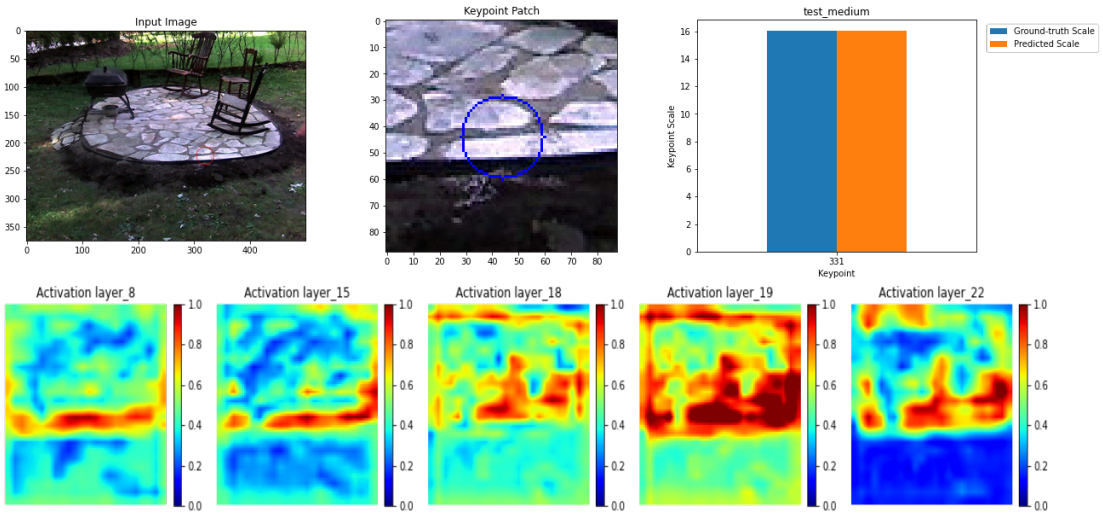


Figure 5.9: Correct scale-estimation from test_medium (2)

Case Study 3: test_large

Figures 5.10 and 5.11 show examples from the test_large with incorrectly estimated characteristic scales. Both keypoint patches are larger than the entire patch size, which could have affected the network's estimation. The heatmaps for Figure 5.10 show a strong response at the bottom right of the patch, indicating a contrast and texture difference from the overall patch. However, it might not be conclusive enough to explain the incorrect estimation. On the other hand, in Figure 5.11, although the heatmaps show some response in the keypoint region, it still resulted in an erroneous prediction. This suggests that the large keypoint scales or outliers could have impacted the network's estimation in these cases.

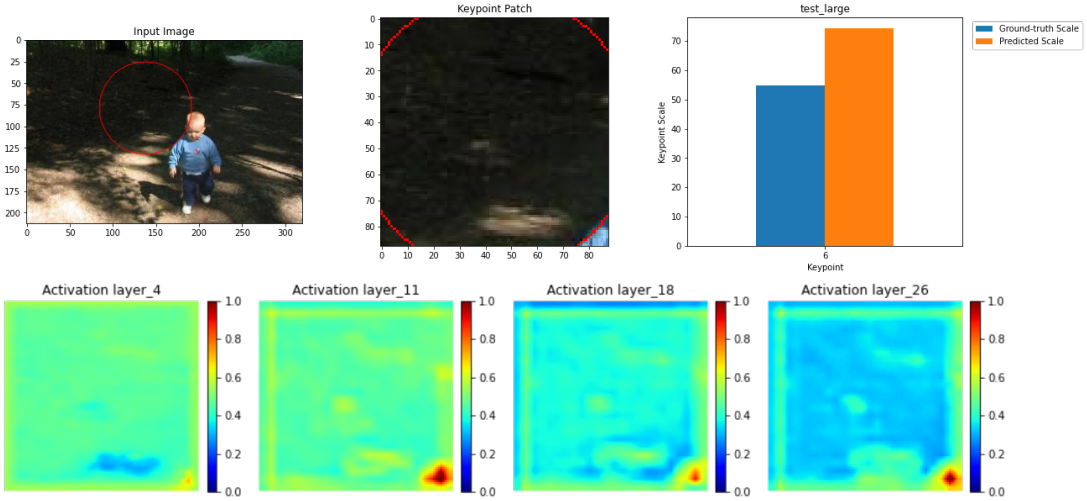


Figure 5.10: Incorrect scale-estimation from test_large (1)

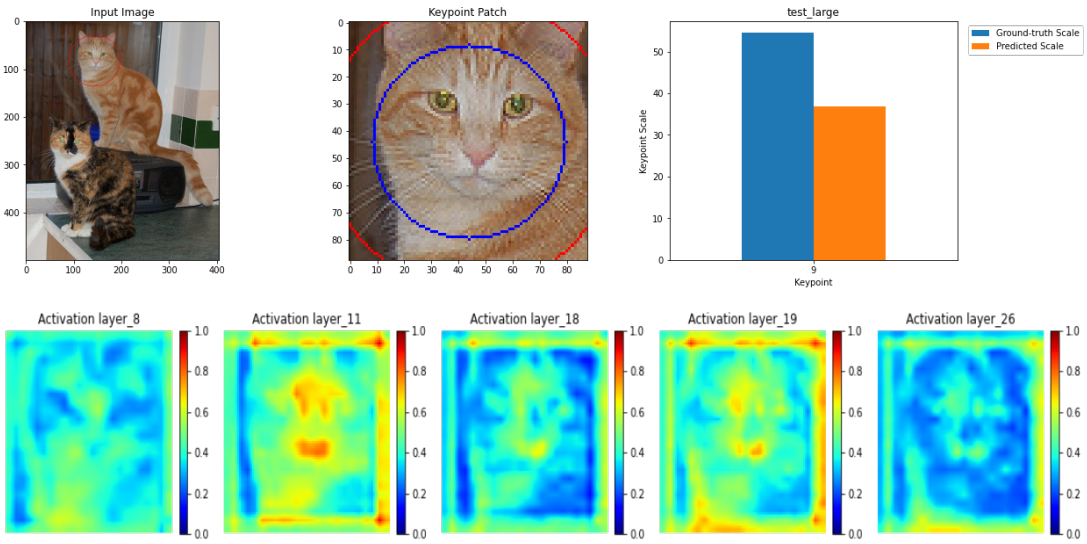


Figure 5.11: Incorrect scale-estimation from test_large (2)

Examples of keypoints from the test_large dataset with correctly estimated characteristic scales are shown in Figures 5.12 and 5.13. The heatmaps in Figure 5.12 indicate that the region of the keypoint has played a significant role in the correct estimation. In contrast, in Figure 5.13, the region of the keypoint does not appear to have impacted the correct prediction as much as the highly distinguishable structural differences in the patch.

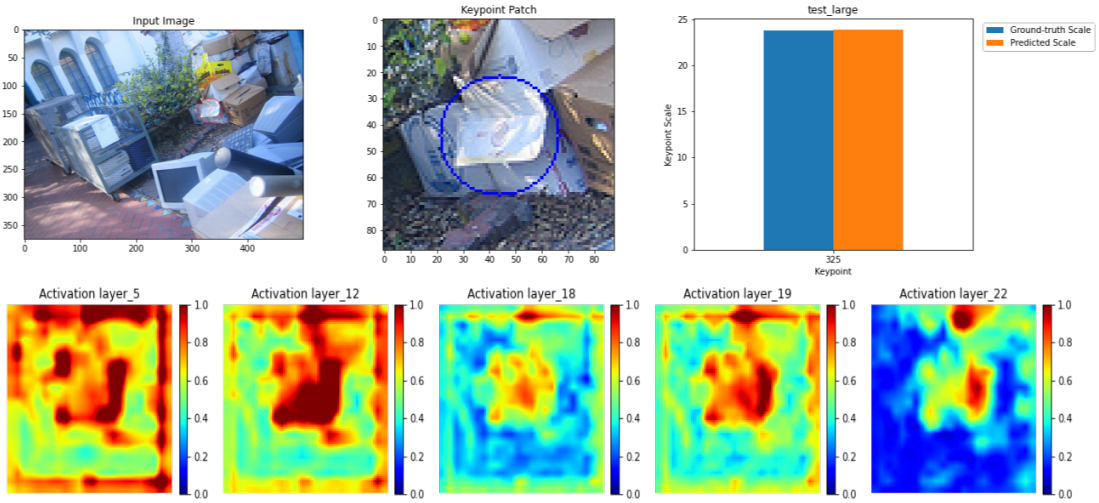


Figure 5.12: Correct scale-estimation from test_large (1)

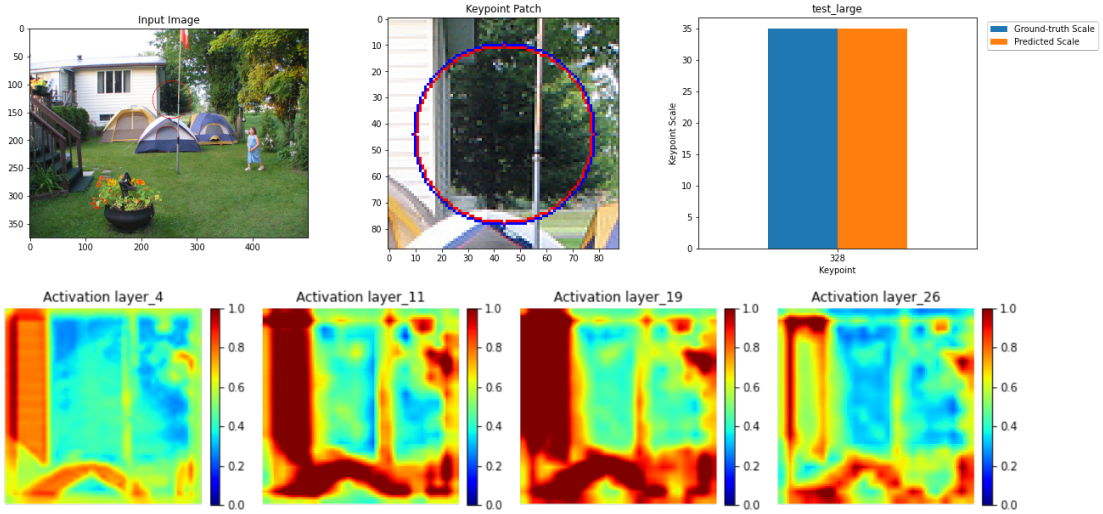


Figure 5.13: Correct scale-estimation from test_large (2)

Case Study 4: Scaled test_small

The test_small dataset contains small-sized keypoints with an average characteristic scale of 3.02, while the input patch size at which the keypoint is centered is 88×88, which is approximately 14.5 times larger than the average. To address this discrepancy, scaled versions of the original input patch are investigated by zooming in by a factor of 2 and 4. By considering the scaled versions, a higher level of abstraction is being provided to the network. By including these scaled versions, the accuracies improved from 39.3 % to 66.07 % and 81.08 %, respectively. These improved accuracies suggest that the chosen fixed input size is insufficient for predicting small-scaled keypoints accurately and performs significantly better when the scaled versions are included. Moreover, including the scaled inputs marginally increases the accuracies for the test_medium and test_large datasets as well. The results are summarized in Table 5.7 and Figure 5.14.

Input Keypoint (%)	Accuracy test_small (%)	Accuracy test_medium (%)	Accuracy test_large (%)
Original	39.34	81.98	79.04
+ Scaled by 2	66.07	93.39	82.63
+ Scaled by 4	81.08	94.89	83.23

Table 5.7: Overall Accuracy variation for original and scaled patches at an error threshold of 10%

The performance of the network for 25 keypoints from the test_small dataset, which were initially estimated incorrectly using the original input, was improved by considering the scaled input instead. Figure 5.15 shows the improvement in network performance. It shows that the scaled input often leads to a more accurate estimation of the keypoints initially estimated incorrectly using the original input.

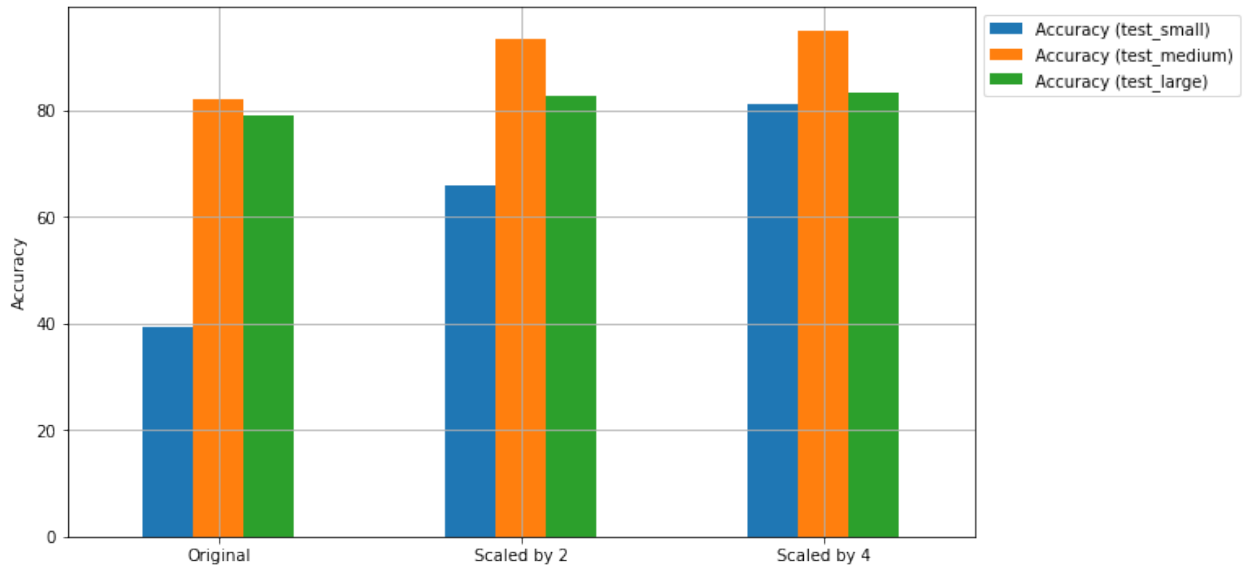


Figure 5.14: Overall Accuracy variation for original and scaled patches at an error threshold of 10%

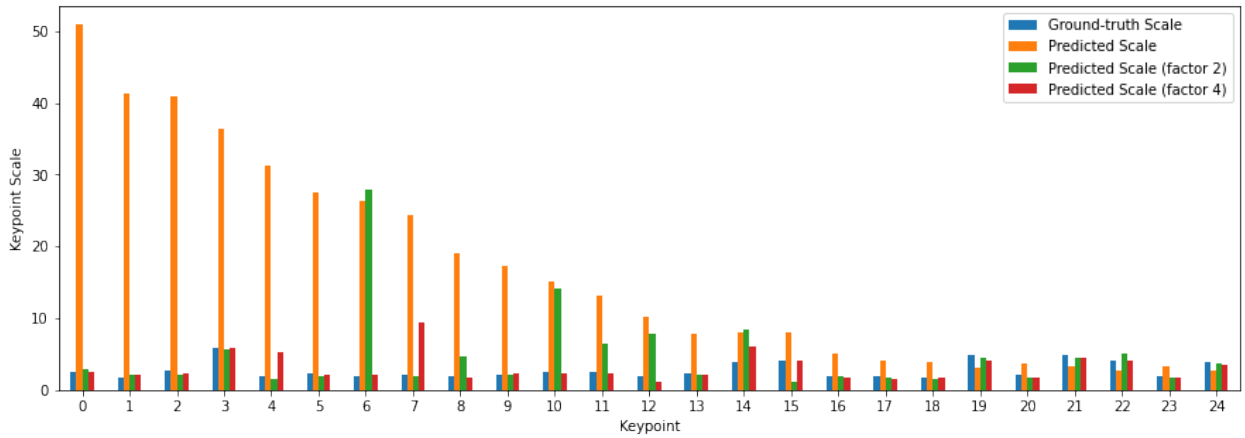


Figure 5.15: Comparison of estimated scales for original, scaled by 2, and scaled by 4 input patch in test_small

Figures 5.16, 5.17, and 5.18 show examples of the estimated scales with original and their scaled input patches. These examples highlight the incorrect estimation of characteristic scale when the original patch is provided as input, compared to the improved performance of scaled versions.

The first example discussed in Section 5.2.3 is revisited in Figure 5.16 to elaborate on the observed improvement. The same heatmaps are examined for all three cases, and although a high response towards the corners of the patch is seen in all cases, the response becomes stronger with the scaled inputs. The second example from Section 5.2.3 is considered in Figure 5.17, where there is a clear improvement in network performance, as seen by analyzing the heatmaps. In addition to the dependency on the region surrounding the keypoint, nearby regions also have strong responses, which supports earlier observations regarding the network performance considering other regions of the patch beside the keypoint. Another example in Figure 5.18 shows that the heatmaps become stronger with the scaled input patches and more concentrated around the keypoint.

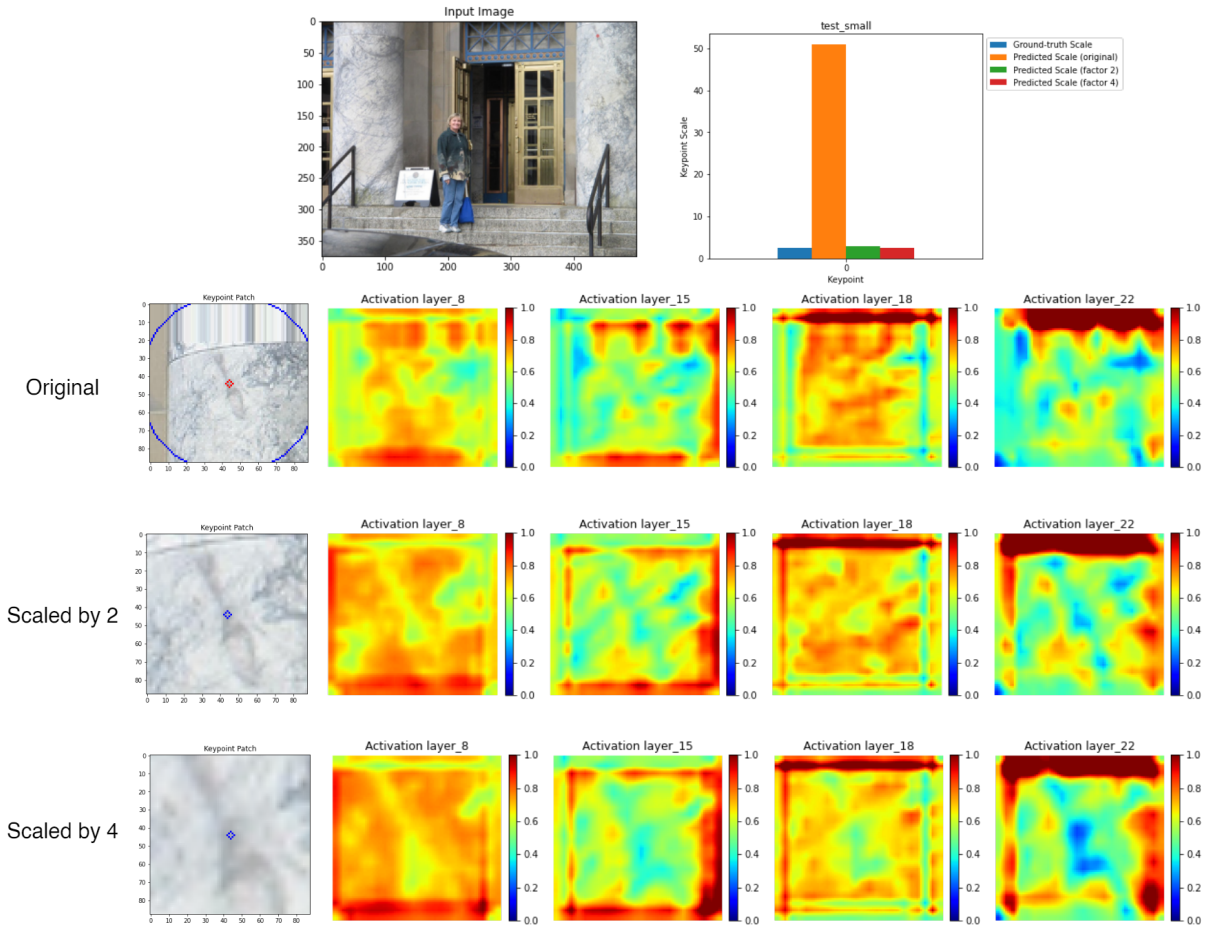


Figure 5.16: Original, scaled by 2, and scaled by 4 with their corresponding heatmaps (1)

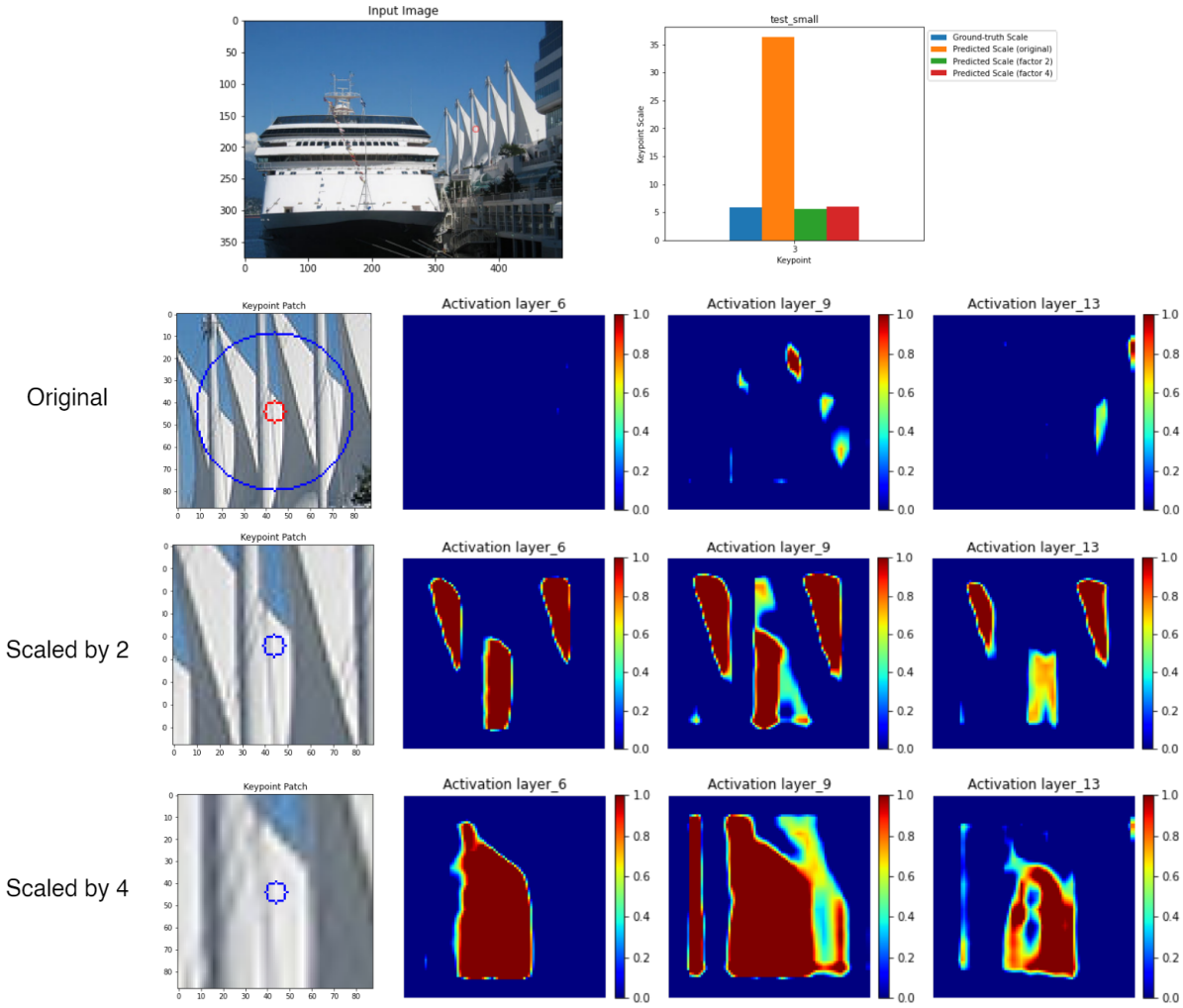


Figure 5.17: Original, scaled by 2, and scaled by 4 with their corresponding heatmaps (2)

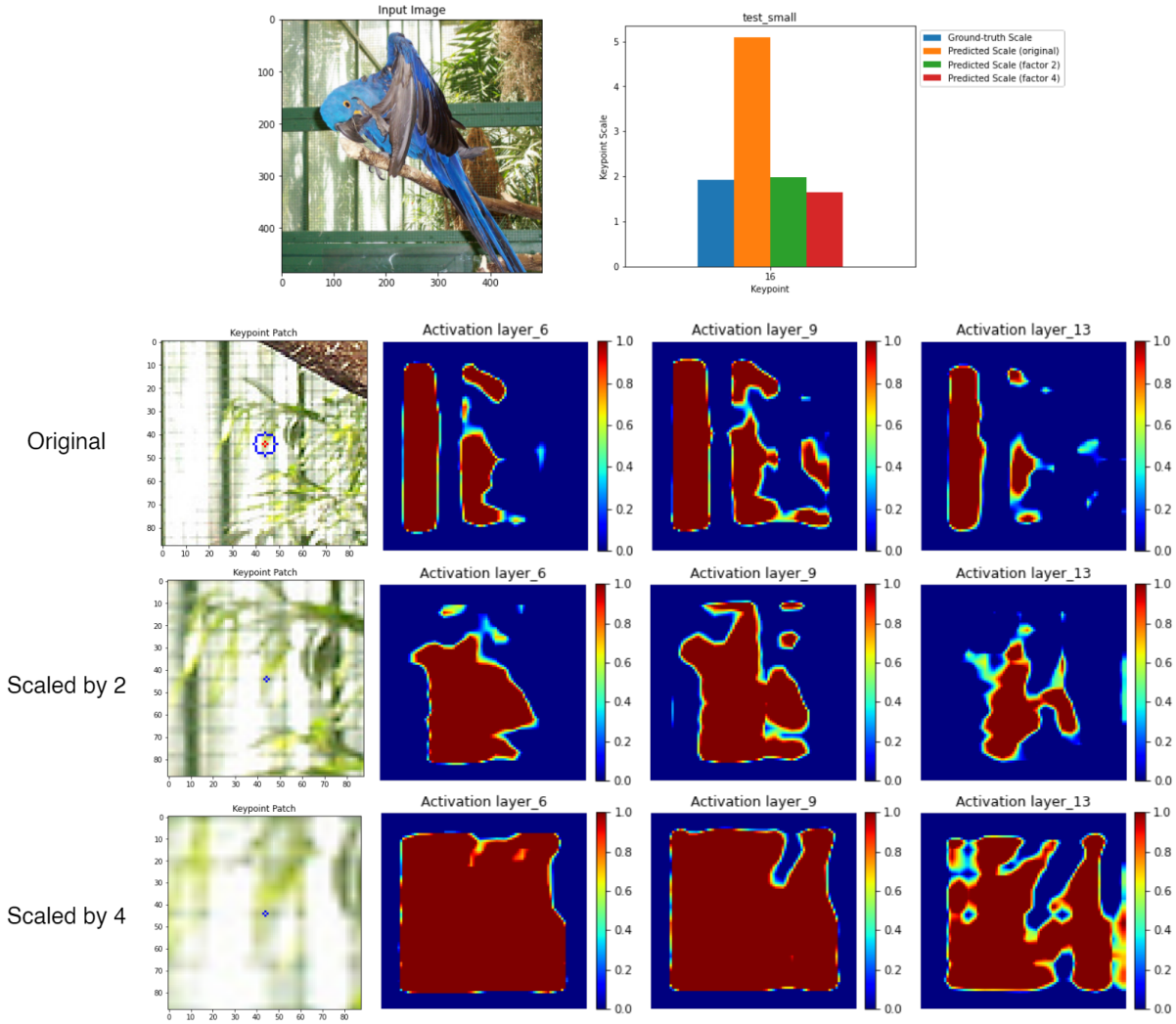


Figure 5.18: Original, scaled by 2, and scaled by 4 with their corresponding heatmaps (3)

Summary of the Qualitative Analyses

The outputs from specific activation layers of the Mod-ResNet34 network have been considered in this analysis. Although some layers may not provide helpful information, analyzing them still provides insight into the regions of the patch that the network uses to estimate the characteristic scales.

- In most cases, the distinctiveness of the keypoints affects the network's performance. Some keypoints in test_small have little distinguishing information, while outliers larger than the patch size appear in test_large.
- When the keypoint region does not provide much information, as seen from small-scaled keypoints in test_small, it is unclear what the network identifies. The network may focus on distinguishable patterns elsewhere in the patch, which could lead to either a correct or an incorrect estimation.
- After analyzing specific examples in test_medium, it can be concluded that the higher accuracy for this particular dataset is due to the regions being sufficiently large for the network to focus predominantly on to make an estimation. In addition to the region of the keypoint, the network also utilizes other characteristics that can be found elsewhere in the patch.
- In some cases, the network relies entirely on the keypoint region to make correct predictions, while in others, it uses other characteristics found elsewhere in the patch. This indicates that the network utilizes the overall characteristics of the patch, making its estimations robust to variations in input scale.
- Increasing the level of abstraction by zooming into the input patch by a factor of 2 and 4 improved the estimation of small-scaled keypoints significantly, indicating that a higher level of abstraction leads to better estimation of such keypoints.

5.3 Other approaches

The implemented framework directly obtains keypoint information from SIFT and SURF and uses CNN to regress the characteristic scales but does not fully utilize the multi-scale nature of CNNs. However, the network can become more robust by considering keypoint information from different levels of abstraction (different layer outputs). Since increasing the level of abstraction improved estimation during testing, it is important to explore such

approaches. This section briefly introduces an approach that utilizes the multi-scale nature of CNNs to estimate the characteristic scale of local image features.

Feature Pyramid Networks

The method in [LDG+17] aims to utilize the inherent multi-scale, pyramid-like structure of CNNs for detecting objects at different scales. The goal is to construct a top-down network architecture for obtaining high-level semantic feature maps over various scales. When provided with a single-scale input image of an arbitrary size, the network produces proportionally sized feature maps at multiple scales like a typical CNN would. This system provides a solution for multi-scale object detection [LDG+17].

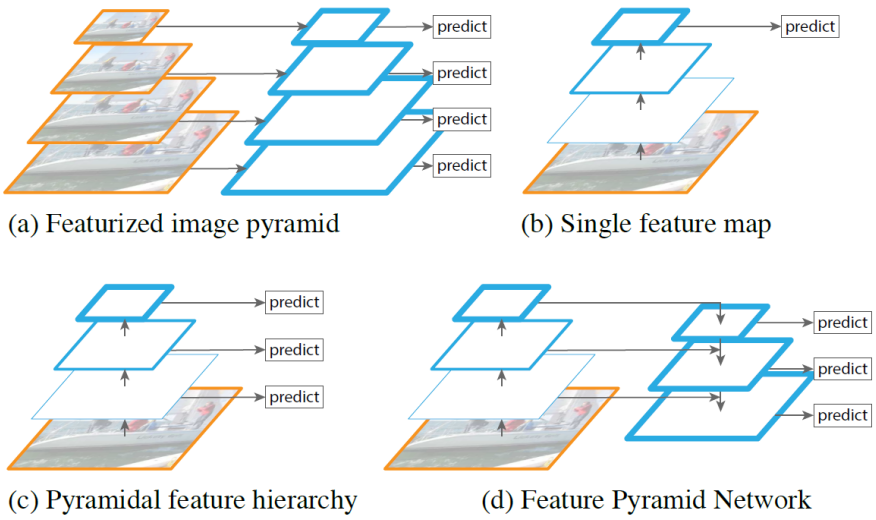
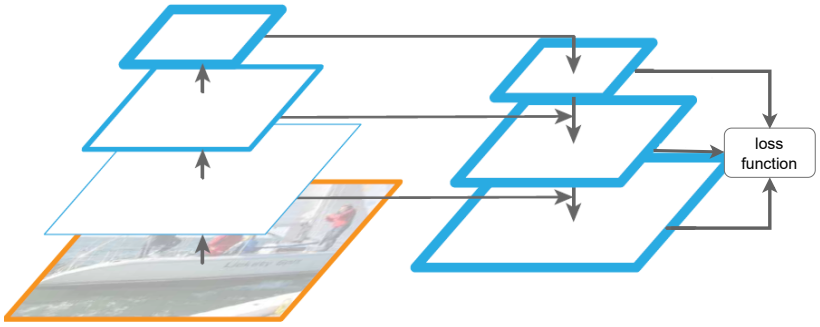


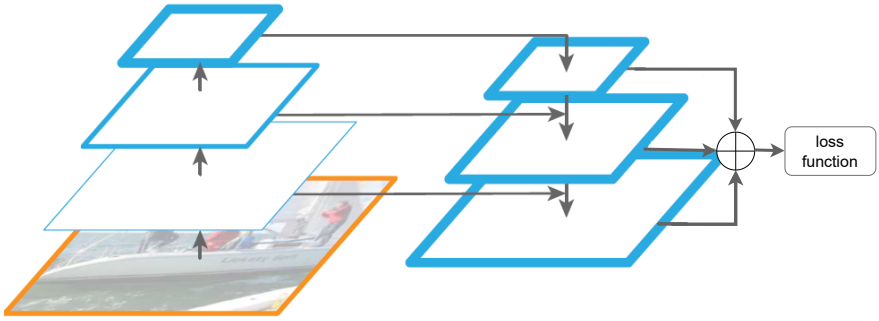
Figure 5.19: Construction of a Feature Pyramid Network (blue outlines indicate feature maps with the thicker outlines denoting semantically stronger features and orange outline indicates the input image) from [LDG+17]

Figure 5.19(a) shows a feature pyramid constructed from an image pyramid. The disadvantage of such an approach is that the features are computed independently at each image scale, slowing the overall process. The idea is to combine the pyramidal feature hierarchy found commonly in methods like the Single Shot Detector (SSD) [LAE+16] shown in Figure 5.19(c) and the robustness to scale variance that is achieved from features computed at a single scale shown in Figure 5.19(b). The Single Shot Detector (SSD) uses multi-scale feature maps from different layers to predict. However, it fails to reuse higher-resolution feature maps from the pyramidal hierarchy. Figure 5.19(d) shows the developed Feature Pyramid Network (FPN), where an architecture which combines both low-level, semantically strong,

and high-level, semantically weak features via a top-down approach with lateral connections. Hence, provided with an input image at a single scale, the developed network can build a feature pyramid with semantics at all levels [LDG+17].



(a) Feature Pyramid Network intermediate representations used independently



(b) Feature Pyramid Network intermediate representations combined

Figure 5.20: Feature Pyramid Networks to optimize a network to learn/predict characteristic scales. Adapted from [LDG+17]

In the context of this work, such an architecture designed for multi-scale representation of an object could be adapted for estimating characteristic scales of local features. Such a multi-scale representation could be achieved by applying a pooling/subsampling after every stage in a CNN architecture, for example, the outputs at each stage's last residual block of a ResNet architecture. There are several ways these representations can be used to estimate characteristic scales. Representations at multiple scales can be treated as independent predictions, and either (1) can be used independently, as shown in Figure 5.20(a), or (2) combined, as shown in Figure 5.20(b) to design a new loss function which considers multi-scale information and optimize the network to learn/predict characteristic scales more robust to scale variance.

6 Conclusion and Outlook

Conclusion

In this work, the objective of developing a framework to estimate characteristic scales using deep learning has been accomplished. For this purpose, relevant basics of object recognition, traditional methodologies for keypoint detection, and deep learning are elaborated. A literature review was conducted to identify suitable approaches for CNN-based scale estimation of local image features. The vast majority of the identified papers deal exclusively with end-to-end feature detectors and descriptors. There is no direct focus on the estimation of characteristic scales but on localizing keypoints accurately in scale space, and therefore no existing results to serve as a benchmark could be determined. However, the datasets and the CNN architectures used in the approaches served as a good starting point for experiments in this work. Specifically, the ResNet blocks used in LF-Net helped to adapt and modify the standard ResNet34 architecture as Mod-ResNet34 for this work. Another commonly used standard architecture, VGG16, was adapted and modified as Mod-VGG16 and ModRed-VGG, where the main modifications have been with respect to reducing the channel widths and, in turn, reducing the total number of learnable parameters.

One significant contribution of this work is creating the training data using the PASCAL VOC dataset. SIFT was applied on a subset of 5000 images from the PASCAL dataset to create the training dataset, SIFT-KP5000. Similarly, another dataset was created by applying SURF on the same subset of images, SURF-KP5000. A histogram-based filtering approach was followed on different characteristic scales for keypoints to create a balanced training dataset with keypoint information.

Another subset of 100 images was randomly chosen to create a test dataset with 1000 unfiltered keypoints. The test dataset was further divided into three subgroups of keypoints, test_small, test_medium, and test_large, which allowed for a systematic analysis. An accuracy metric was implemented based on the thresholding of the absolute difference between

the ground-truth and predicted scales. This metric provided a quantifiable measure of accuracy, similar to accuracy for classification problems, since characteristic scales are continuous values.

This work investigated various network architectures, and the Mod-ResNet34 architecture performed best. An overall accuracy of 85.89 % and 88.19 % on test datasets created using SIFT and SURF, respectively, was achieved. A detailed analysis regarding the network performance on different scaled keypoints was carried out.

In conclusion, this work presents a solution using traditional methodologies, SIFT and SURF, with commonly used architecture, ResNet34, to estimate characteristic scales. There are several shortcomings concerning the chosen patch size and the small-scaled keypoints, and several such cases were discussed.

Outlook

- Since networks struggle to estimate characteristic scales of relatively small keypoints accurately, further research can be carried out to filter smaller local image features using a filtering approach based on, for example, the strength of the keypoint or contrast of the region around the keypoint.
- This work mainly worked with SIFT and SURF detectors. Other keypoint detectors such as ORB, KAZE, and AKAZE can be investigated.
- Further optimizations in terms of hyperparameter tuning can be carried out with a grid-search approach to obtain better network performance.
- This work mainly adapted and modified standard architectures such as VGG16 and ResNet34, and other commonly used architectures such as U-Nets, and Encoder-Decoder-based networks can be investigated.
- As introduced in Section 5.3, the multi-scale, pyramidal nature of CNNs can be utilized to develop a more robust framework to estimate characteristic scales.
- The visualization of the heatmaps in Section 5.2.3 can be improved by employing a weighting scheme, such as Gaussian.

Bibliography

[AA21] Atmaja, Bagus T.; Akagi, Masato: Evaluation of Error-and Correlation-based Loss Functions for Multitask Learning Dimensional Speech Emotion Recognition. In: *Journal of Physics: Conference Series* Bd. 1896 IOP Publishing, 2021, pp. 012004

[BL02] Brown, Matthew; Lowe, David G.: Invariant Features from Interest Point Groups. In: *The British Machine Vision Conference* Bd. 4, 2002, pp. 398–410

[Bro18] Brownlee, Jason: *What is the Difference Between a Batch and an Epoch in a Neural Network*. `https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/`. 2018

[BTVG06] Bay, Herbert; Tuytelaars, Tinne; Van Gool, Luc: SURF: Speeded Up Robust Features. In: *Computer Vision - European Conference on Computer Vision (ECCV)* Springer, 2006, pp. 404–417

[CW19] Carvalho, LE; Wangenheim, Aldo von: 3D Object Recognition and Classification: A Systematic Literature Review. In: *Pattern Analysis and Applications* 22 (2019), pp. 1243–1292

[DHS00] Duda, Richard O.; Hart, Peter E.; Stork, David G.: *Pattern Classification and Scene Analysis*. 2. Wiley New York, 2000. – ISBN 978–0–471–05669–0

[DS15] Dong, Jingming; Soatto, Stefano: Domain-size Pooling in Local Descriptors: DSP-SIFT. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 5097–5106

[DSC22] Dubey, Shiv R.; Singh, Satish K.; Chaudhuri, Bidyut B.: Activation Functions in Deep Learning: A Comprehensive Survey and Benchmark. In: *Neurocomputing* (2022)

[EMA23] El Moudden, Tarik; Amnai, Mohamed: Building an Efficient Convolution Neural Network from Scratch: A Case Study on Detecting and Localizing Slums. In: *Scientific African* (2023), pp. e01612

[EVGW+] Everingham, Mark.; Van Gool, Lue.; Williams, Christopher. K. I.; Winn, John.; Zisserman, Andrew.: *The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results*. http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html,

[GBC16] Goodfellow, Ian; Bengio, Yoshua; Courville, Aaron: *Deep Learning*. MIT Press, 2016. – ISBN 9780262035613

[GC19] Ghojogh, Benyamin; Crowley, Mark: The Theory Behind Overfitting, Cross-validation, Regularization, Bagging, and Boosting: Tutorial. (2019)

[GL11] Grauman, Kristen; Leibe, Bastian: Visual Object Recognition. In: *Synthesis Lectures on Artificial Intelligence and Machine Learning* 5 (2011), No. 2, pp. 1–181

[GSGK22] Gohil, Arnav; Shroff, Anshul; Garg, Arnav; Kumar, Shailender: A Compendious Research on Big Data File Formats. In: *6th International Conference on Intelligent Computing and Control Systems (ICICCS)*, 2022, pp. 905–913

[GW18] Gonzalez, Rafael C.; Woods, Richard E.: Digital Image Processing, Global Edition. 19 (2018). ISBN 9780133356724

[HAA16] Hassaballah, Mahmoud; Abdelmgeid, Aly A.; Alshazly, Hammam A.: Image Features Detection, Description and Matching. In: *Image Feature Detectors and Descriptors: Foundations and Applications* (2016), pp. 11–45

[HM20] Hennig, Markus; Mertsching, Bärbel: Fast SIFT-like Keypoint Detection for Contours [Unpublished Manuscript]. (2020)

[Hor90] Horn, Roger A.: The Hadamard Product. In: *Proc. Symp. Appl. Math* Bd. 40, 1990, pp. 87–169

[HS+88] Harris, Chris; Stephens, Mike et al.: A Combined Corner and Edge Detector. In: *Alvey Vision Conference* Bd. 15 Citeseer, 1988, pp. 10–5244

[HS15] He, Kaiming; Sun, Jian: Convolutional Neural Networks at Constrained Time Cost. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 5353–5360

[HZRS16] He, Kaiming; Zhang, Xiangyu; Ren, Shaoqing; Sun, Jian: Deep Residual Learning for Image Recognition. In: *Proceedings of the IEEE Conference on computer Vision and Pattern Recognition*, 2016, pp. 770–778

[JP20] Joshi, Khushbu; Patel, Manish I.: Recent Advances in Local Feature Detector and Descriptor: A Literature Survey. In: *International Journal of Multimedia Information Retrieval* 9 (2020), No. 4, pp. 231–247

[JSZ+15] Jaderberg, Max; Simonyan, Karen; Zisserman, Andrew et al.: Spatial Transformer Networks. In: *Advances in Neural Information Processing Systems* 28 (2015)

[Kok17] Kokkinos, Iasonas: Ubernet: Training a Universal Convolutional Neural Network for Low-, Mid-, and High-Level Vision Using Diverse Datasets and Limited Memory. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 6129–6138

[Kri16] Krig, Scott: Interest Point Detector and Feature Descriptor Survey. In: *Computer vision metrics*. Springer, 2016, pp. 187–246

[LAE+16] Liu, Wei; Anguelov, Dragomir; Erhan, Dumitru; Szegedy, Christian; Reed, Scott; Fu, Cheng-Yang; Berg, Alexander C.: SSD: Single Shot Multibox Detector. In: *Computer Vision - European Conference on Computer Vision (ECCV)* Springer, 2016, pp. 21–37

[LBBH98] LeCun, Yann; Bottou, Léon; Bengio, Yoshua; Haffner, Patrick: Gradient-based Learning Applied to Document Recognition. In: *Proceedings of the IEEE* 86 (1998), No. 11, pp. 2278–2324

[LDG+17] Lin, Tsung-Yi; Dollár, Piotr; Girshick, Ross; He, Kaiming; Hariharan, Bharath; Belongie, Serge: Feature Pyramid Networks for Object Detection. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 2117–2125

[Lin94] Lindeberg, Tony: Scale-space Theory: A Basic Tool for Analyzing Structures at Different scales. In: *Journal of Applied Statistics* 21 (1994), No. 1-2, pp. 225–270

[Lin98] Lindeberg, Tony: Feature Detection with Automatic Scale Selection. In: *International Journal of Computer Vision* 30 (1998), No. 2, pp. 79–116

[LMAPH19] Lathuilière, Stéphane; Mesejo, Pablo; Alameda-Pineda, Xavier; Horaud, Radu: A Comprehensive Analysis of Deep Regression. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42 (2019), No. 9, pp. 2065–2081

[Low99] Lowe, David G.: Object Recognition from Local Scale-invariant Features. In: *Proceedings of the IEEE International Conference on Computer Vision* Bd. 2 IEEE, 1999, pp. 1150–1157

[Low04] Lowe, David G.: Distinctive Image Features from Scale-Invariant Keypoints. In: *International Journal of Computer Vision* 60 (2004), No. 2, pp. 91–110

[LRPM19] Laguna, Axel B.; Riba, Edgar; Ponsa, Daniel; Mikolajczyk, Krystian: Key.Net: Keypoint Detection by Handcrafted and Learned CNN Filters. In: *IEEE/CVF International Conference on Computer Vision (ICCV)* (2019), pp. 5835–5843

[McK10] McKinney Wes: Data Structures for Statistical Computing in Python. In: *Proceedings of the 9th Python in Science Conference*, 2010, pp. 56 – 61

[MJF+21] Ma, Jiayi; Jiang, Xingyu; Fan, Aoxiang; Jiang, Junjun; Yan, Junchi: Image Matching from Handcrafted to Deep features: A Survey. In: *International Journal of Computer Vision* 129 (2021), pp. 23–79

[MS04] Mikolajczyk, Krystian; Schmid, Cordelia: Scale & Affine Invariant Interest Point Detectors. In: *International Journal of Computer Vision* 60 (2004), No. 1, pp. 63–86

[Mur21] Muraleedharan, Arjun: *Write Your Own Custom Data Generator for TensorFlow Keras*. https://medium.com/analytics-vidhya/write-your-own-custom-data-generator-for-tensorflow-keras-1252b64e41c3. 2021

[Nie15] Nielsen, Michael A.: *Neural Networks and Deep Learning*. vol. 25. Determination Press San Francisco, CA, USA, 2015. – 15–24 S.

[OTFY18] Ono, Yuki; Trulls, Eduard; Fua, Pascal; Yi, Kwang M.: LF-Net: Learning Local Features from Images. In: *Advances in Neural Information Processing Systems* 31 (2018)

[PRS+16] Pezoa, Felipe; Reutter, Juan L.; Suarez, Fernando; Ugarte, Martín; Vrgoč, Domagoj: Foundations of JSON Schema. In: *Proceedings of the 25th International Conference on World Wide Web* International World Wide Web Conferences Steering Committee, 2016, pp. 263–273

[Rei20] Reinhold Haeb Umbach: Statistical and Machine Learning. (2020)

[Roe17] Roell, Jason: *From Fiction to Reality: A Beginner's Guide to Artificial Neural Networks*. https://towardsdatascience.com/from-fiction-to-reality-a-beginners-guide-to-artificial-neural-networks-d0411777571b. 2017

[Sah18] Saha, Sumit: *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way*. https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53. 2018

[SGS15] Srivastava, Rupesh K.; Greff, Klaus; Schmidhuber, Jürgen: Highway Networks. (2015)

[SZ15] Simonyan, Karen; Zisserman, Andrew: Very Deep Convolutional Networks for Large-scale Image Recognition. In: *3rd International Conference on Learning Representations* (2015), pp. 1–14

[Tre10] Treiber, Marco A.: *An Introduction to Object Recognition: Selected Algorithms for a Wide Variety of Applications*. Springer Science & Business Media, 2010. – ISBN 978–1–84996–234–6

[TS18] Tareen, Shaharyar Ahmed K.; Saleem, Zahra: A Comparative Analysis of SIFT, SURF, KAZE, AKAZE, ORB, and BRISK. In: *International Conference on Computing, Mathematics and Engineering Technologies (iCoMET)* (2018), pp. 1–10

[WYW+19] Wang, Wei; Yang, Yujing; Wang, Xin; Wang, Weizheng; Li, Ji: Development of Convolutional Neural Network and its Application in Image Classification: A Survey. In: *Optical Engineering* 58 (2019), No. 4, pp. 040901–040901

[YSS19] Yani, Muhamad; S, M.T. Budhi I. Si.; S.T., M.T. Casi S.: Application of Transfer Learning using Convolutional Neural Network Method for Early Detection of Terry's Nail. In: *Journal of Physics: Conference Series* Bd. 1201 IOP Publishing, 2019, pp. 012052

[YTLF16] Yi, Kwang M.; Trulls, Eduard; Lepetit, Vincent; Fua, Pascal: LIFT: Learned Invariant Feature Transform. In: *Computer Vision - European Conference on Computer Vision (ECCV)* (2016), pp. 467–483

Appendix

Code Snippets

Custom Data Generator. Adapted from [Mur21]

```
class CustomDataGen(tf.keras.utils.Sequence):

    def __init__(self, df, X_col, y_col, batch_size=32,
    input_size=(58, 58, 3), patch_width, color=True):
        self.df = df.copy() # A copy of the input dataframe
        self.X_col = X_col # Inputs path of the image and co-ordinates of
            the keypoint
        self.y_col = y_col # Inputs the characteristic scale of the keypoint
        self.batch_size = batch_size
        self.input_size = input_size
        self.patch_width = patch_width
        self.color = color # BGR or Grayscale
        self.n = len(self.df) # Length of the dataframe

    def __get_input(self, path, cord):
        if self.color==True:
            image = cv2.imread(path) # Reads the image in BGR
        else:
            image = cv2.imread(path, 0) # Reads the image in Grayscale

        # Adds a border to the image to account for keypoints on the border
        image = cv2.copyMakeBorder(image, self.patch_width*2,
            self.patch_width*2,
```

```
22            self.patch_width*2, self.patch_width*2,
                  borderType=cv2.BORDER_REPLICATE)
23
24        x, y = cord[1]+self.patch_width*2, cord[0]+self.patch_width*2
25
26        # Extracts the patch around the keypoint
27        patch = np.array(image[x-self.patch_width:x+self.patch_width,
28             y-self.patch_width:y+self.patch_width])
29        patch = patch.reshape(self.patch_width*2, self.patch_width*2,
              self.input_size[-1])
30        return patch/255 # Return the normalized keypoint patch
31
32    def __get_output(self, label):
33        return np.array(label)
34
35    def __get_data(self, batches):
36        # Generates data containing batch_size samples
37        path_batch = batches[self.X_col['path']]
38        cord_batch = batches[self.X_col['cord']]
39        size_batch = batches[self.y_col['size']]
40
41        X_batch = np.asarray([self.__get_input(x, y) for x, y in
              zip(path_batch, cord_batch)])
42        y_batch = np.asarray([self.__get_output(y) for y in size_batch])
43        return X_batch, y_batch
44
45    def __getitem__(self, index):
46        batches = self.df[index * self.batch_size:(index + 1) *
              self.batch_size]
47        X, y = self.__get_data(batches)
48        return X, y
49
50    def __len__(self):
51        return self.n // self.batch_size
```

Mod-ResNet34 Architecture, Adapted from [HZRS16]

```
1
2  # Mod-ResNet34 Architecture
3
4  def identity_block(x, filter):
5      # Copy input tensor to a variable called x_skip
6      x_skip = x
7      # Layer 1
8      x = tf.keras.layers.Conv2D(filter, (3,3), padding = 'same')(x)
9      x = tf.keras.layers.BatchNormalization(axis=3)(x)
10     x = tf.keras.layers.Activation('relu')(x)
11     # Layer 2
12     x = tf.keras.layers.Conv2D(filter, (3,3), padding = 'same')(x)
13     x = tf.keras.layers.BatchNormalization(axis=3)(x)
14     # Add Residue
15     x = tf.keras.layers.Add()([x, x_skip])
16     x = tf.keras.layers.Activation('relu')(x)
17     return x
18
19 def convolutional_block(x, filter):
20     # Copy input tensor to a variable called x_skip
21     x_skip = x
22     # Layer 1
23     x = tf.keras.layers.Conv2D(filter, (3,3), padding = 'same', strides =
           (2,2))(x)
24     x = tf.keras.layers.BatchNormalization(axis=3)(x)
25     x = tf.keras.layers.Activation('relu')(x)
26     # Layer 2
27     x = tf.keras.layers.Conv2D(filter, (3,3), padding = 'same')(x)
28     x = tf.keras.layers.BatchNormalization(axis=3)(x)
29     # Processing Residue with conv(1,1)
30     x_skip = tf.keras.layers.Conv2D(filter, (1,1), strides = (2,2))(x_skip)
31     # Add Residue
32     x = tf.keras.layers.Add()([x, x_skip])
33     x = tf.keras.layers.Activation('relu')(x)
34     return x
```

```
35
36 def Mod-ResNet34(shape = (88, 88, 3)):
37     # Step 1 (Setup Input Layer)
38     x_input = tf.keras.layers.Input(shape)
39     x = tf.keras.layers.ZeroPadding2D((3, 3))(x_input)
40
41     # Step 2 (Initial Conv layer along with MaxPool)
42     x = tf.keras.layers.Conv2D(64, kernel_size=7, strides=2,
           padding='same')(x)
43     x = tf.keras.layers.BatchNormalization()(x)
44     x = tf.keras.layers.Activation('relu')(x)
45     x = tf.keras.layers.MaxPool2D(pool_size=3, strides=2, padding='same')(x)
46
47     # Define size of sub-blocks and initial filter size
48     block_layers = [3, 4, 6, 3]
49     filter_size = 64
50
51     # Step 3 Add the Resnet Blocks
52     for i in range(4):
53         if i == 0:
54             # For sub-block 1 Residual/Convolutional block not needed
55             for j in range(block_layers[i]):
56                 x = identity_block(x, filter_size)
57         else:
58             # One Residual/Convolutional Block followed by Identity blocks
59             # The filter size will go on increasing by a factor of 2
60             filter_size = filter_size*2
61             x = convolutional_block(x, filter_size)
62             for j in range(block_layers[i] - 1):
63                 x = identity_block(x, filter_size)
64
65     # Step 4 End Dense Network
66     x = tf.keras.layers.AveragePooling2D((2,2), padding = 'same')(x)
67     x = tf.keras.layers.Flatten()(x)
68     x = tf.keras.layers.Dense(512, activation = 'relu')(x)
69     x = tf.keras.layers.Dense(1, activation='linear')(x)
```

```
70      model = tf.keras.models.Model(inputs = x_input, outputs = x, name =
            "Mod-ResNet34")
71      return model
```

Glossary

Adam Adaptive Moment Estimation. 29–31, 34, 60

ANN Artificial Neural Network. 21

ANNs Artificial Neural Networks. 5

CNN Convolutional Neural Network. 5, 6, 10, 25, 26, 31, 35, 36, 40, 45, 50–52, 77–79, 96

CNNs Convolutional Neural Networks. 2, 5, 24, 25, 35, 36, 38, 59, 76, 77, 80, 95

DoG Difference-of-Gaussians. 1, 13–15, 40

LF-Net Local Feature Network. 2, 3, 38, 39, 51, 52, 96

LIFT Learned-Invariant Feature Transform. 2, 3, 36–40, 51, 52, 96

LoG Laplacian-of-Gaussian. 11–13, 95

MAE Mean Absolute Error. 22, 23, 60–63

MLP Multi-layer Perceptron. 21, 95

MSE Mean Squared Error. 22, 23, 60, 61, 63

NMS Non-Maxima-Suppression. 37–39, 42, 44

NN Neural Network. 21, 22, 24, 28

ReLU Rectified Linear Unit. 23, 28, 31, 39, 41

ResNet Residual Neural Network. 31, 39, 44

RMSE Root Mean Squared Error. 34, 60–63, 94

RMSprop Root Mean Squared Propagation. 29–31, 34, 60–62

SGD Stochastic Gradient Descent. 29–31, 34, 60, 61

SIFT Scale-Invariant Feature Transform. 1, 2, 5, 10, 13, 14, 16, 17, 19, 35, 44–49, 59, 79, 80, 95, 96

STN Spatial Transformer Network. 36, 38

SURF Speeded-Up Robust Features. 2, 5, 10, 16–19, 35, 44–46, 49, 62, 79, 80, 95

List of Tables

List of Figures

Declaration

I hereby certify that I have prepared the submitted Master thesis with the title

Deep Learning-based Scale Estimation of Local Image Features

independently and have not used any sources or aids other than those specified. Quotations have been marked as such.

Paderborn, March 30, 2023

Dheeraj Rajashekar Poolavaram