BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

DETECTION OF A YOGA POSES IN IMAGE

DETEKCE JÓGOVÝCH POZIC V OBRAZE

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **JIŘÍ KUTÁLEK**
AUTOR PRÁCE

SUPERVISOR **prof. Ing. ADAM HEROUT, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačové grafiky a multimédií (UPGM) Akademický rok 2020/2021

Zadání bakalářské práce

Student: **Kutálek Jiří**
Program: Informační technologie
Název: **Detekce jógových pozic v obraze**
Detection of a Yoga Poses in Image
Kategorie: Zpracování obrazu

Zadání:

1. Seznamte se s problematikou počítačového vidění, analýzy obrazu, rozpoznání kostry člověka.
2. Vyhledejte vhodné zdroje o technikách rozpoznání pózy člověka, se zaměřením na jógové pozice.
3. Pořiďte datovou sadu pro učení a vyhodnocování detekce pozic člověka.
4. Experimentujte s algoritmy počítačového vidění a strojového učení, nalezněte vhodné řešení.
5. Vyhodnocujte vytvořené řešení na datové sadě, datovou sadu dále rozšiřujte a řešení iterativně vylepšujte.
6. Demonstrujte schopnosti dosaženého řešení.
7. Zhodnoťte dosažené výsledky a možnosti pokračování projektu; vytvořte plakátek a krátké video pro prezentování projektu.

Literatura:

- Shih-En Wei, Varun Ramakrishna, Takeo Kanade, Yaser Sheikh: Convolutional Pose Machines, CVPR 2016
- Bharath Ramsundar, Reza Bosagh Zadeh: TensorFlow for Deep Learning: From Linear Regression to Reinforcement Learning, O'Reily Media, 2018
- Gary Bradski, Adrian Kaehler: Learning OpenCV; Computer Vision with the OpenCV Library, O'Reilly Media, 2008
- Richard Szeliski: Computer Vision: Algorithms and Applications, Springer, 2011

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3, značné rozpracování bodů 4 a 5.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Herout Adam, prof. Ing., Ph.D.**
Vedoucí ústavu: Černocký Jan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 12. května 2021
Datum schválení: 5. května 2021

Abstract

The motivation for this thesis is the concept of a smartphone app detecting Yoga poses and displaying Yoga frames to a user. The goal of this project is proving that even a simple Convolutional Neural Network (CNN) model can be trained to recognize and classify video frames from a Yoga session. I created an application in which the videos are manually annotated. The data, consisting of frames captured from 162 collected videos based on the annotations, is then passed to train a CNN model. The Dataset consists of 22 000 images of 22 different Yoga poses. The frames are captured using the OpenCV library, the training process is handled by the TensorFlow platform and the Keras API, and the results are visualized in the TensorBoard toolkit. The Model's multi-class classification accuracy reaches 91% when the binary cross-entropy loss function and the sigmoid activation function are used. Despite the promising experimental results, the main contributions are the dataset forming tools and the Dataset itself, which both helped to confirm the proof-of-concept.

Abstrakt

Motivací pro tuto práci je koncept mobilní aplikace detekující jógové pozice a zobrazující výsledky uživateli. Cílem této práce je ověření hypotézy, že i jednoduchý model konvoluční neuronové sítě dokáže rozpoznávat a klasifikovat snímky z nahraných jógových sekvencí. Napsal jsem aplikaci, v níž se pořízená videa ručně oanotují. Výsledná data, sestávající ze snímků extrahovaných ze 162 jógových videí na základě jednotlivých anotací, jsou pak použita k trénování modelu sítě. Vytvořený Dataset obsahuje 22 000 obrázků reprezentující 22 různých jógových pozic. Snímky jsou z videí extrahovány pomocí knihovny OpenCV, trénování modelu je plně v režii platformy TensorFlow a API Keras, a výsledky jsou vizualizovány pomocí nástroje TensorBoard. Přesnost modelu detekovat jógové pozice dosahuje 91% při použití aktivační funkce sigmoid a binary cross-entropy loss function. I přes slibně vypadající dosažené výsledky jsou hlavním přínosem této práce nástroje pro tvorbu datasetu a vytvořený Dataset samotný. Díky nim byla navrhovaná hypotéza úspěšně ověřena.

Keywords

Yoga Pose Detection, Video Annotation Application, Training CNN for Yoga Poses Recognition

Klíčová slova

Detekce jógových pozic v obraze, Aplikace na anotování videí, Trénování konvoluční neuronové sítě pro rozpoznávání jógových pozic

Reference

KUTÁLEK, Jiří. *Detection of a Yoga Poses in Image.* Brno, 2021. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor prof. Ing. Adam Herout, Ph.D.

Rozšířený abstrakt

Motivace Cvičení jógy je v dnešní době populární. Díky tomu je na trhu dostupná velká spousta aplikací pro mobilní telefony, které učí jejich uživatele cvičit jógu správně. Přestože to některé činí opravdu dobře, neposkytují uživateli vizuální zpětnou vazbu ke cvičení. Tato práce představuje koncept aplikace, která uživateli vybere několik snímků ze zacvičené a natočené jógové sekvence, čímž mu žpětnou vazbu poskytne. Navrhované řešení navíc umožňuje uložení vybraných snímků pro případné pozdější použití (např. konzultace s instruktorem jógy), a je praktičtější než procházení celého nahraného videa. Hlavní překážkou v realizaci takového nápadu je schopnost aplikace rozpoznávát jednotlivé jógové pozice.

Metody pro detekci osob v obraze samozřejmě existují a fungují velmi dobře. Architektura těchto řešení, jako např. OpenPose, detekující kostru celého těla, je však příliš komplexní na to, aby šla implementovat na mobilní telefony. Nabízí se řešení pomocí obyčejné konvoluční neuronové sítě, protože techniky pro její implementaci do mobilního telefonu patrně existují. Cílem této práce je zjistit, zda i jednoduchý model konvoluční neuronové sítě dokáže dobře rozpoznat a klasifikovat jednotlivé jógové pozice. Za tímto účelem byla sesbírána videa různých jógových sekvencí, a z nich byl vytvořen Dataset.

Konvoluční neuronové sítě Většina ze současných řešení pro detekci osob v obraze staví na konvolučních neuronových sítích. Konvoluční neuronová síť je typicky složena z několika konvolučních vrstev extrahujících příznaky z obrázku, následována plně propojenými vrstvami klasifikujícími data. Důležitou roli v extrakci příznaků hrají i aktivační funkce, díky nimž může síť klasifikovat i nelineárně oddělitelná data.

Operace konvoluce je prováděna pomocí konvolučního filtru (matice čísel), "klouzajícího" přes obrázek a násobícího hodnoty čísel ze své matice s hodnotami jednotlivých pixelů. Výsledky jsou sečteny a na vzniklé matice je pak aplikována daná aktivační funkce. Při extrakci příznaků se využívá i tzv. pooling vrstev, a to k redukci rozměrů těchto výsledných matic a ke snížení parametrů modelu sítě. Po několika kombinacích konvolučních a pooling vrstev jsou výstupní matice transformovány do jednoho dlouhého vektoru, který je předán plně propojeným vrstvám sítě starajícím se o samotnou klasifikaci.

Existuje mnoho významných architektur modelů konvolučních neuronových sítí. Jeden z vůbec prvních modelů, LeNet-5, byl navržen pro rozpoznávání ručně psaných znaků. Revoluční síť AlexNet, představená v roce 2012, pak odstartovala vlnu zájmu v oblasti konvolučních neuronových sítí, jejichž architektury záhy začaly nabírat na složitosti. Důkazem toho jsou například modely VGG-16 nebo ResNet.

Tvorba Datasetu Za pomoci vedoucího mé práce jsem sesbíral 162 videí tří různých jógových sekvencí a sestavil z nich Dataset. Přestože videa pocházejí pouze od dvou různých osob, liší se pozicemi kamery, pozadím, světelnými podmínkami při natáčení, i barvami úboru cvičících osob.

Za účelem anotace videí byla vytvořena anotační aplikace. Anotace definuje pro každé video přesná čísla snímků, v jejichž rozmezí jsou prováděny jednotlivé jógové pózy. Tento interval udává pro každou zacvičenou pózu množinu snímků, použitelných jako výsledná data. Skript pak prochází soubory s anotacemi a pomocí knihovny OpenCV extrahuje z videí snímky ve specifikovaných intervalech, čímž tvoří Dataset.

Přestože jsem díky anotacím schopen vytvořit dataset čítající stovky tisíc obrázků, ve skutečnosti obsahuje Dataset 22 000 obrázků reprezentující 22 různých jógových pozic. Jednotlivé pozice jsou v Datasetu zastoupeny rovnoměrně. Čtvrtina obrázků, 5 500, tvoří validační sadu, zbytek, 16 500, pak tu trénovací. Rozlišení všech obrázků v Datasetu je 256×256 px.

Trénování modelu sítě Celý proces trénování modelu konvoluční neuronové sítě, včetně načítání a konfigurace dat, je postaven na platformě TensorFlow a přidruženém API Keras. Trénovací i validační data jsou po načtení normalizována a jejich rozlišení je sníženo dle potřeby (96×96 px – 224×224 px). Ještě předtím, než jsou předány modelu sítě, jsou obrázky z trénovací sady augmentovány kvůli minimalizaci rizika přetrénování modelu.

Samotný model pak sestává z šesti až deseti konvolučních, čtyř až pěti max pooling a nejvýše tří plně propojených vrstev. Aktivační funkcí konvolučních vrstev je typicky ReLU a v případě poslední vrstvy Modelu klasifikující data je to buď sigmoid, nebo softmax. Podle toho je pak vybrána příslušná cross-entropy loss function, penalizující Model za chybné predikce.

Trénovací proces probíhá typicky ve 100 – 150 iteracích, během nichž je Model natrénován na trénovacích datech a validační data pak prověřují jeho přesnost predikce. Na konci každé iterace je spočítána nová hodnota learning rate, udávající míru změny hodnot vah. Data i přesnost Modelu jsou vizualizovány pomocí nástroje TensorBoard.

Experimenty a dosažené výsledky K vyladění přesnosti Modelu bylo provedeno tisíce hodin experimentů. Vyšlo najevo, že kombinace aktivační funkce sigmoid a binary cross-entropy loss function přináší stabilně lepší výsledky než kombinace funkcí softmax a categorical cross-entropy. Mimo to jsem našel zdánlivě optimální množství jednotlivých vrstev modelu sítě a jejich parametrů, a zjistil jsem, že významné modely sítí AlexNet a VGG-16 nedokáží klasifikovat data z Datasetu tak přesně, jako některé jiné mnou navržené modely. Hledal jsem také optimální hodnoty algoritmu aktualizujícího hodnoty learning rate. Experimentoval jsem s jednotlivými augmentačními technikami, abych zjistil, jak jejich parametry nastavit, aby přinášely co nejvíce užitku. Otestoval jsem, které jógové pozice síť často zaměňuje za jiné, a zjistil jsem, že schopnost Modelu detekovat jednotlivé pózy je závislá i na výběru videí pro trénovací a validační sadu.

Model sestávající z osmi konvolučních, pěti max pooling a tří plně propojených vrstev dosáhl úspěšnosti 91% v detekci jógových pozic i na záměrně těžké validační sadě. Jde o průměrnou hodnotu v posledních 40 iteracích procesu trénování Modelu. Pro sadu, do níž byla videa vybrána benevolentněji byla pak úspěšnost dokonce 95%. Tyto hodnoty platí pro rozlišení 224×224 px všech obrázků v Datasetu. Pro výrazně nižší rozlišení 96×96 px je pak úspěšnost modelů zhruba o 6% nižší.

Závěr Dosažené výsledky ukazují, že i relativně jednoduchý model konvoluční neuronové sítě dokáže detekovat jógové pozice vcelku spolehlivě, což znamená, že cíl práce lze směle označit za splněný. Přenost modelu je však závislá na kvalitě daného datasetu. Aktuálně navržený Dataset stále nelze označit jako komplexní, ba dokonce dokonalý, a je třeba jej dále rozšiřovat. I přes to však zůstává, společně s nástroji pro jeho tvorbu, hlavním přínosem této práce.

Detection of a Yoga Poses in Image

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author Jiří Kutálek under the supervision of prof. Ing. Adam Herout, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Jiří Kutálek
May 12, 2021

Acknowledgements

I would like to thank my supervisor prof. Ing. Adam Herout, Ph.D., for guidance, motivation, the project idea itself, and, finally, the contributions to the Dataset forming process. I would also like to thank my brother Adam Kutálek for assistance with recording the Yoga videos. Computational resources were supplied by the project "e-Infrastruktura CZ" (e-INFRA LM2018140) provided within the program Projects of Large Research, Development and Innovations Infrastructures.

Contents

1 Introduction **2**

2 Motivation for Yoga Pose Detection **3**
2.1 Concept of a Yoga Application 3
2.2 Yoga Pose Detection Solutions 5

3 Convolutional Neural Networks **8**
3.1 Feature Extraction 8
3.2 Classification 11
3.3 Significant CNN Model Architectures 13

4 Image Classification Process **16**
4.1 Introduction to the Neural Network Training Process 16
4.2 Image Classification on the TensorFlow platform 18

5 Forming the Dataset **22**
5.1 Collecting Videos and Annotating them in a Custom Annotation Application 22
5.2 Data Shape and Volume 24

6 Training Convolutional Neural Network Model for Yoga Pose Recognition **26**
6.1 Dataset Loading, Configuration and Arrangement 26
6.2 CNN Model Training Process 28

7 Experimental Results **31**
7.1 Findings, Advancements and Model Tuning 31
7.2 Achieved Results 36

8 Conclusion **40**

Bibliography **41**

A Contents of the included storage media **44**

Chapter 1

Introduction

In general, computer vision methods of detecting human figures in image and video (pose estimation techniques) are primarily based on variations of Convolutional Neural Networks. The are several approaches of how they can be used to recognize Yoga poses. However, some of them have a limited applicability, as their architecture is too heavy to be implemented in mobile devices, such as smartphones.

Yoga exercising becomes more and more popular throughout the years. This study introduces an idea of a smartphone application providing a Yoga workout feedback, by displaying a few Yoga pose frames from a recorded workout session. To achieve it, a Convolutional Neural Network model is trained to detect the individual Yoga poses.

The objective of this thesis is to find and get familiar with the existing human pose estimation techniques, collect enough data and form a dataset from them, build and train a CNN model, and make experiments in order to tune it as well as possible. It should be pointed out that the goal of this work is the proof-of-concept (verifying the potential of detecting Yoga poses with a simple CNN model), not designing the application itself.

As for the structure of the thesis, Chapter 2 introduces the reader into Yoga exercising, describes the existing Yoga pose detection solutions, and presents the concept of an application serving as the motivation for this work. After the explanation of how Convolutional Neural Networks work and mentioning some of the significant architectures in Chapter 3, a workflow of the image classification process on a corresponding machine learning platform is described in Chapter 4. The concept of building an input pipeline and a CNN model itself, as well as the commonly used approach of training and validating one, are covered.

The following chapters are related to the actual work done. Chapter 5 presents methods used for forming the Dataset, including a custom annotation application, and the shape of the Dataset itself. After that, Chapter 6 describes how is the Dataset loaded, configured and integrated into the training process, how is the CNN Model being built and trained, and how are the results being evaluated. Lastly, the experiments and findings made, and the actual results achieved are presented in Chapter 7.

Chapter 2

Motivation for Yoga Pose Detection

Before digging deeper into the topic of Yoga Pose Detection, it should be stated, why is this problem tackled, and how it can possibly be solved. In this Chapter, Yoga exercising is briefly explained, some widely used Yoga smartphone apps are introduced, and the motivation for designing a custom one is presented. In addition, the existing pose estimation techniques capable of detecting Yoga poses are listed, and the reason why it is worth exploring a custom solution is explained.

2.1 Concept of a Yoga Application

In the beginning of this Section, Yoga exercising is briefly explained. Subsequently, some of the popular smartphone applications, aiding a user to practice Yoga are presented. In the end, an idea of designing a custom smartphone application, which serves as the motivation for this thesis, is presented.

Brief Introduction to Yoga Exercising "Yoga exercising" describes a physical activity, during which a person performs a few postures, which are often organized into a single sequence. For instance, the work presented in this thesis covers three different Yoga sequences, each of them consisting of several Yoga poses. An example of the **Sun Salutation** sequence, and its associated poses, is shown in Figure 2.1.

Yoga Smartphone Apps Currently Available Over the last few years, Yoga popularity is rapidly increasing. Due to this, there are plenty of instructional videos and Yoga teaching smartphone applications available on the internet.

For instance, the **Daily Yoga** app[1], available on both of the main smartphone platforms, Android and iOS, aims to turn a Yoga practice into a user's daily habit. The app provides various guided classes, helping beginners to learn the basics, and advanced users to follow the world's class Yoga teachers. A global community of active users shares their ideas and motivation with each other. An example of a beginner class is shown in Figure 2.2.

Classes provided by the **Yoga Studio** smartphone app[2] often focus on specific goals such as strength, balance or flexibility, and some of them feature unusual themes such as

[1] https://play.google.com/store/apps/details?id=com.dailyyoga.inc&hl=cs&gl=US

[2] https://play.google.com/store/apps/details?id=com.gaiam.yogastudio&hl=cs&gl=US

Figure 2.1: Twelve different Yoga poses are included in this variation of the Sun Salutation Yoga sequence. Each of the poses is identified by a name. In fact, this Sun Salutation sequence consists of 24 poses in total, meaning that most of the poses presented here are performed twice during the sequence. The **Standing Forward Fold** pose, portrayed in the fourth picture in the top line, is performed even four times. Only the **Low Cobra** and **Upward Facing Dog** Yoga poses, shown in the last two pictures in the bottom line, are performed just once during the sequence. These images are samples from the Dataset, which is presented later in Section 5.2.

Yoga for runners or *Yoga for back pain*. In addition, users can schedule and track the Yoga classes, create a custom flow through the app's special feature, and design a music playlist for it.

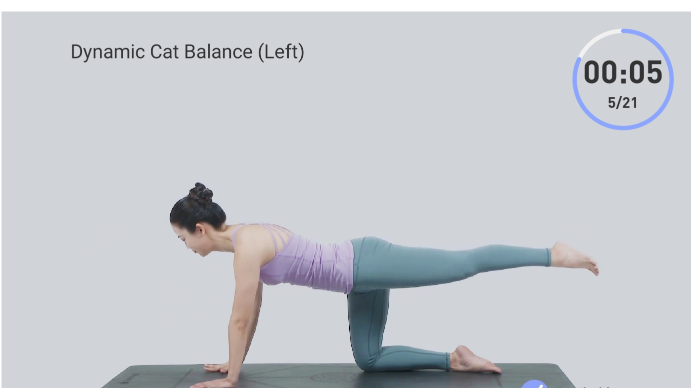

Figure 2.2: An example of a beginner workout class in the Daily Yoga smartphone application[3]. The subject in the picture demonstrates the exercise of the current posture (the *Dynamic Cat Balance*), in order to show the user the correct way of doing it. The picture was taken as a screenshot when using the application.

However, the smartphone applications available (such as these two) hardly ever provide a **visual workout feedback**. A visual workout feedback can be obtained either by exercising in front of a mirror, or by recording the Yoga session on a camera and then exploring the video. But, both may be quite inconvenient.

[3] https://play.google.com/store/apps/details?id=com.dailyyoga.inc&hl=cs&gl=US

Idea of a Custom Application This thesis introduces an idea of a smartphone app providing the workout feedback easily, specifically by choosing and showing just a few frames, representing the performed Yoga poses, from the recorded workout session. Thanks to that, a user does not need to manually seek frames in the video. Reviewing a dozen photos is much more accessible than watching and rewinding a whole Yoga session video. Furthermore, a person is able to either evaluate images by their eyes, or save them to gallery and consult with their Yoga instructor later, which may both lead to the exercise progression.

The main problem obviously is, how would an app like this know, which of the many frames in the video to choose and display to the user? In the following Section, the Yoga pose detection methods, which are capable of solving this kind of problem are presented.

2.2 Yoga Pose Detection Solutions

In general, the majority of pose estimation techniques (computer vision methods of detecting human figures in image and video) are based on variations of Convolutional Neural Networks (which are described in Chapter 3 in detail). There are many different approaches of how they can be used to accurately recognize various Yoga poses.

2.2.1 Existing Solutions

Convolutional Pose Machines In 2016, **Convolutional Pose Machines (CPMs)** [1] presented an innovative systematic design for how Convolutional Neural Networks can be incorporated into the *pose machine* framework [2] for learning image features and image-dependent spatial models for the task of pose estimation. The way of how *image features* are extracted from an image, is explained in the following Chapter 3.

Pose machine use random forest learning methods for classification and self-made functions for extracting features from images. Their sequential architecture, consisting of individual stages, produces more accurate predictions at the end of each stage.

A CPM proposes an architecture composed of CNNs directly operating on belief maps (heatmaps indicating probabilities of pixels belonging to the particular body joints) from previous stages, producing increasingly refined predictions for part locations, without the need for explicit graphical model-style inference. The CPM architecture is shown in Figure 2.3.

Real Time Yoga Pose Detection Using Deep Learning An approach to accurately recognize various Yoga poses using deep learning algorithms [3] has been presented in 2019. A hybrid deep learning model is proposed using CNN and long short-term memory (LSTM) [4] for Yoga recognition on real-time videos, where convolutional layer is used to extract features from keypoints of each frame obtained from OpenPose [5], and is followed by LSTM to give temporal predictions.

The researchers proposed a custom dataset for this purpose, consisting of more than a hundred thousand of Yoga images from six different Yoga poses.

Unfortunately, these methods are not suitable for a smartphone app as the architecture is too heavy for the processing units currently in use.

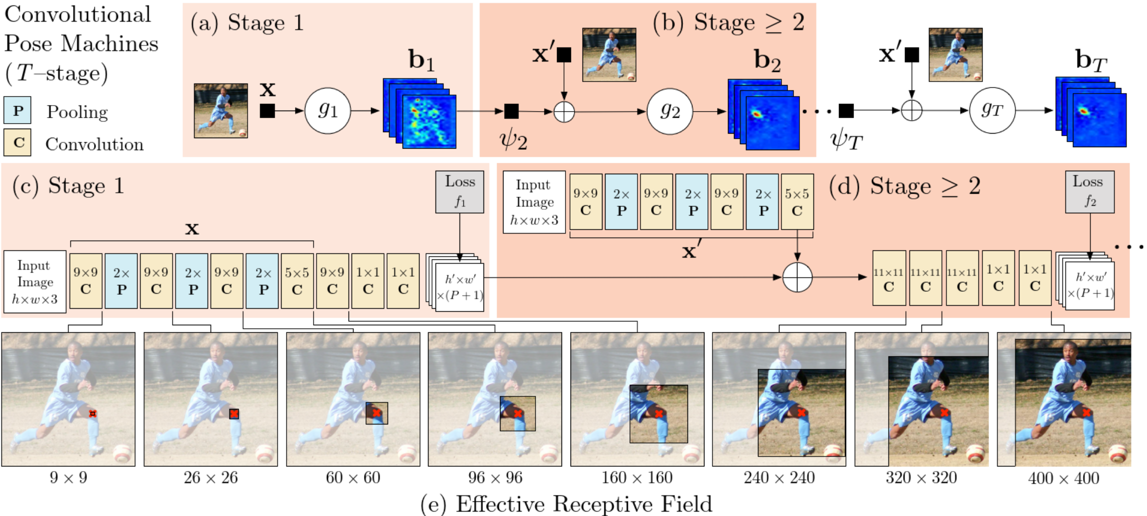

Figure 2.3: Convolutional Pose Machines flowchart from the original paper [1]. The pose machine is shown in insets **(a)** and **(b)**, and the corresponding CNNs are shown in insets **(c)** and **(d)**. Insets **(a)** and **(c)** show the architecture operating only on image evidence in the first stage. Insets **(b)** and **(d)** show the architecture for subsequent stages, which operate both on image evidence, as well as belief maps from preceding stages. The architectures in **(b)** and **(d)** are repeated for all subsequent stages (2 to T). The network is locally supervised after each stage using an intermediate loss layer that prevents vanishing gradients during training. Inset **(e)** shows the effective receptive field on an image (centered at left knee) of the architecture, where the large receptive field enables the model to capture long-range spatial dependencies, such as those between head and knees.

Solutions Utilizing Microsoft Kinect Besides, another paper [6] proposes a system monitoring body parts movement and accuracy of different Yoga poses, which aids the user to practice Yoga.

Apart from that, a research [7] proposes an interactive system capable of recognizing Yoga poses, which can track up to 6 people at the same time. However, these solutions use Microsoft Kinect for human body parts real-time detection, an expensive device most people do not have at hand.

Segmentation and Detection of Yoga Poses in Video A methodology for automatic segmentation and detection of Yoga poses from video is presented in another paper [8]. The proposed segmentation approach is based on the Otsu's thresholding method [9].

Similarly to this thesis, the dataset forming approach presented in the paper is realised by extracting frames from Yoga videos. But, the work introduces only about 40 source videos and does not mention how many images does the dataset contain. In addition, the paper states that the Otsu's thresholding method is more accurate than some other ones, but it is not clear from the text how is the proposed recognition system capable of detecting the individual Yoga poses.

2.2.2 The Proposed Solution

In this work, I design a Convolutional Neural Network (CNN) classifying the Yoga images into categories based on the corresponding Yoga poses.

Currently, there are frameworks that enable implementing a CNN model into a smartphone. For instance, the TensorFlow Lite[4] deep learning framework allows to convert a pre-trained TensorFlow [10, 11] model to a format that can be optimized for speed or storage, and then deployed on smartphones.

The aim of this project is proving that even a simple CNN model architecture can recognize the individual Yoga poses, represented by Yoga images in a custom-built dataset, and thus, verifying the potential of the Yoga app concept (presented in the previous Section 2.1).

[4] https://www.tensorflow.org/lite/

Chapter 3

Convolutional Neural Networks

Using a conventional Artificial Neural Network (ANN), image classification problems become difficult, because 2D images need to be converted to one-dimensional vectors. This increases the number of parameters rapidly, which takes storage and processing capability. Convolutional Neural Network, a class of neural networks, convolves learned parameters with input data, and uses 2D convolutional layers, making this architecture well suited to processing 2D data, such as images.

A CNN typically consists of several convolutional layers dealing with feature extraction followed by fully connected layers managing the classification itself (Figure 3.1). An activation function plays an important role in feature extraction allowing to classify even non-linearly separable data.

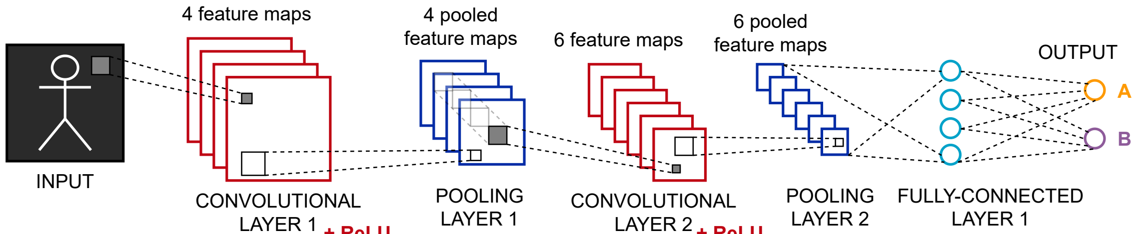

Figure 3.1: Schematic of a Convolutional Neural Network. The feature maps typically gradually decrease their spatial resolution, while increasing the number of channels. Ultimately, the CNN extracts more and more abstract information and produces a decision with typically very small number of outputs.

3.1 Feature Extraction

The task of a CNN is to take an image on the input, process it and classify it into a proper category. Each image has its resolution and exists in a certain color space. For example, a 256×256 RGB (Red, Green, Blue) image is represented as an $256 \times 256 \times 3$ array of matrix of RGB (3 refers to the number of channels: red, green and blue).

If there were large quantities of high-resolution images on the input, things would get computationally very intensive. Therefore, it is necessary to reduce the images into a form which is easier to process, without losing key features (attributes of the image) which are crucial for getting a good prediction. The complexity of the features detected typically

grows with the layer depth (the first CNN layers detect basic features such as edges or corners while the further ones detect objects, faces and other more complex features).

3.1.1 Convolutional Layers

The operation used to extract features from images is called **convolution** (the mathematical definition is presented in the following Equation (3.1)).

$$(f * g)(t) := \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau \tag{3.1}$$

Convolution is performed on an input image using a *kernel* or a *filter* (small matrix of values), sliding over the image, multiplying its values with the image pixel values and summing them (see Figure 3.2 for more detailed explanation). These outputs are also referred to as **feature maps**.

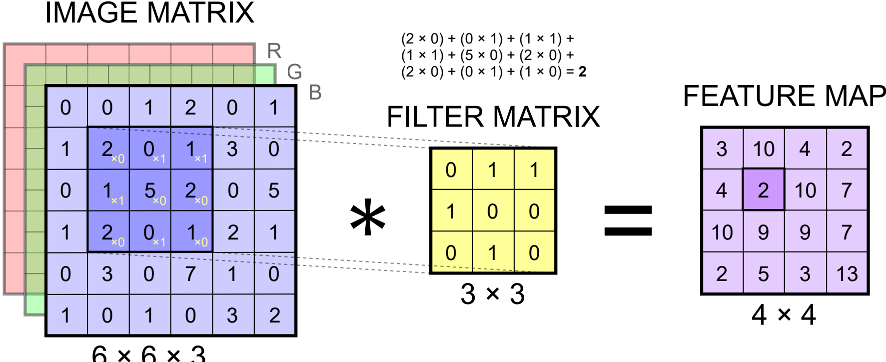

Figure 3.2: Schematic of the convolution operation performed on a blue channel of an RGB image matrix. The overlapping values of the image are multiplied by the filter values, which slides over it (one pixel at a time starting from the top left) and adds all of them up. There is a single output value for each overlap. The convolved features are usually called **feature maps**. The 3×3 is a frequently used filter size, as well as the 1×1 or the 5×5.

The filter moving from the top left of the image to the bottom right does not necessarily need to shift by one pixel at a time. The **strides** parameter specifies the step (number of pixels) the filter is moved across the image. Since the step of the filter is larger, increasing the stride value helps to reduce the output image dimensions. Another way of reducing the volume size is using a larger **filter size**, for instance, 5×5.

After performing the convolution as shown in Fig. 3.2, the convolved feature is reduced in dimensionality compared to the input. However, when working with small images, this limits the number of convolutional layers that can be used. Or sometimes the filter simply does not fit the input image. To deal with these problems, a **padding** can be applied. "Same" padding preserves the spatial dimensions of the volume by padding the picture with zeros. The "same" padding is often used to keep the image dimensions unchanged, while the volume size is then reduced by pooling layers.

3.1.2 Non-Linearity and Activation Functions

After sliding a filter over an image, another mathematical operation is performed. The given output is passed through an **activation function**, which adds non-linearity to the network. Thanks to the activation function, it is possible to classify even non-linearly separable data. Thus, the network can predict the class of a function divided by a non-linear decision boundary.

Sigmoid The **sigmoid** activation function (3.2) is one of the widely used ones in neural networks. The input to the sigmoid function is transformed into a value between 0 and 1. Very large inputs are transformed to a value close to 1.0 and similarly, values much smaller than zero are snapped to 0.0. According to the (0, 1) range, sigmoid is especially used for networks predicting a probability as an output.

$$S(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1} \tag{3.2}$$

ReLU An activation function commonly used for CNN feature extraction is the **ReLU** (Rectified Linear Unit). The output of ReLU is defined as $f(x) = max(0, x)$, which means that the function converts all the negative values on the input into 0 and keeps the positive ones the same (as shown in Fig. 3.3). ReLU is the default choice of activation functions, typically used in hidden (middle) layers of a CNN, because of its fast computation and high performance.

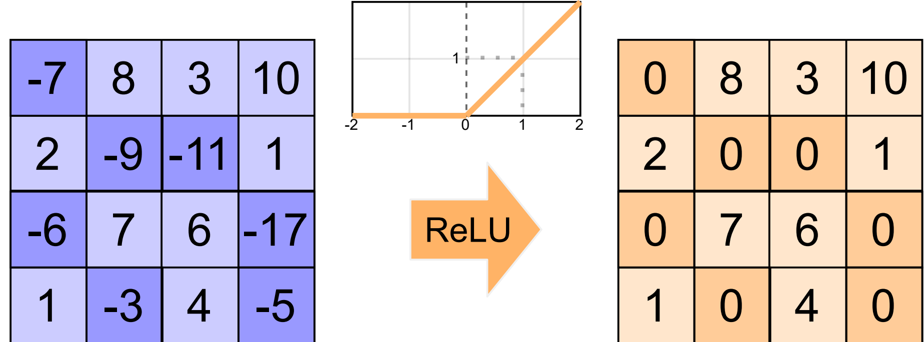

Figure 3.3: Schematic of the ReLU activation function transforming the negative input values into zeros and keeping the positive values the same. An activation introduces non-linearity to a model making it possible to classify even non-linearly separable data.

Softmax The **softmax** activation function is commonly used in the output layers of neural network models, dealing with a multi-class classification problem (when there are more than two class labels requiring a class membership). Similarly to the sigmoid function, the softmax gives an output in (0, 1) range. However, it ensures that outputs along channels sum to 1, which means that softmax units naturally represent a probability distribution. Nevertheless, it is also possible to use the sigmoid activation function for multi-class classification. Softmax is defined in the following Equation (3.3).

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_i}} \tag{3.3}$$

3.1.3 Pooling Layers

A **pooling** layer decreases the feature map size to reduce the number of computational parameters in the network. Spatial pooling (sub-sampling) can be of different types: *max pooling*, *average pooling* or *sum pooling*. Max pooling, the most frequent type of pooling, takes the maximum value from the portion of a feature map (as shown in Fig 3.4). Average pooling returns the average of all the values in a specified window, while sum pooling adds them up.

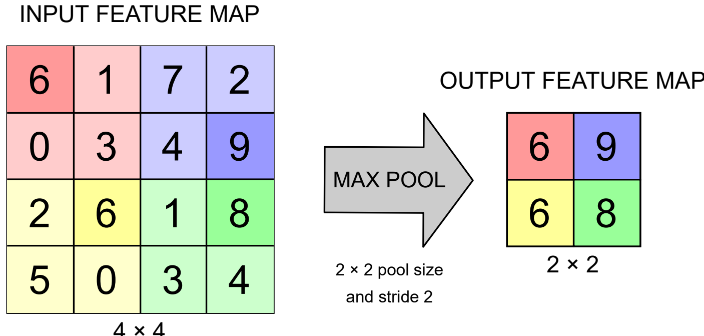

Figure 3.4: Schematic of the **max pooling** technique reducing the dimensionality of a 4×4 feature map. Pool size is set to 2×2 and stride (pool shifting step) is 2, which means that the size of the output map will be half the size of the input one.

The pooling windows are quite similar to the convolutional filters sliding over the image. They decrease the feature map size while keeping the significant information at the same time. Identically, the window is shifted by strides in each dimension, and its size can vary too.

Convolutional and pooling layers together form the part of Convolutional Neural Networks dealing with feature extraction. After this process, a CNN model understands the features and is ready to fed the output to a regular ANN for classification purposes.

3.2 Classification

Just before feeding the final output into the input layer of a regular Artificial Neural Network, an operation called **flattening** must be done. The pooled feature maps (each represented by a matrix) are flattened to a long 1D vector so that it can be passed through an ANN. The process of flattening a feature map into a vector is shown in Figure 3.5.

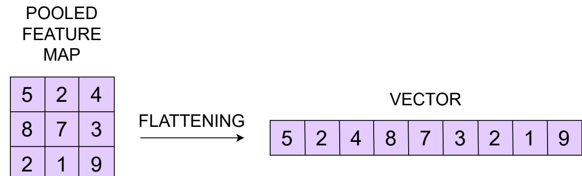

Figure 3.5: Schematic of the flattening operation. The final output from the convolutional and pooling layers must be flattened before it is fed into an Artificial Neural Network. The pooled feature maps are simply transformed into a 1D vector.

3.2.1 Dense Layers

Once the feature extraction is done and all the feature maps are flattened into one vector, a regular Artificial Neural Network (presented in Figure 3.6) handles the classification process itself. The fully connected layers of an ANN (also known as **dense** layers) combine the features into a wider range of attributes which make the CNN more capable of predicting a class that the images belong to.

For instance, the network classifies an image figure as a duck with a probability of 70%, but in fact the image portrays a fish, not a duck. In that case, a *loss function* calculates an error providing a feedback about how accurate the network is, which is then used in the optimization process. The general purpose of how a CNN is trained to classify images, including these error calculations and loss functions, is discussed in Section 4.1.

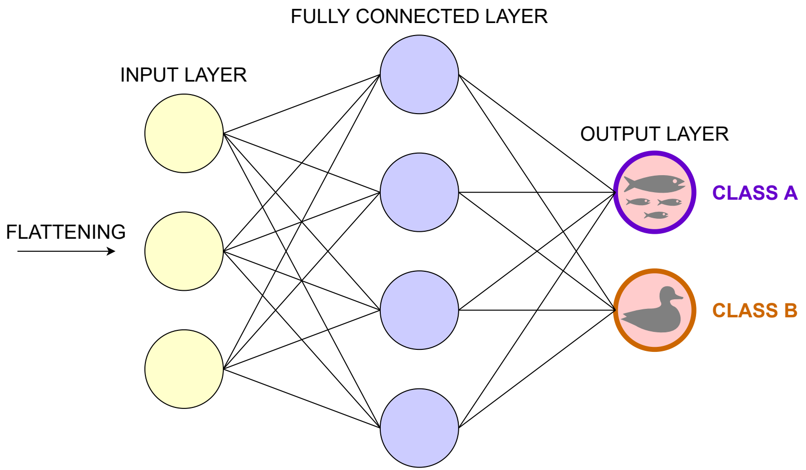

Figure 3.6: Schematic of an Artificial Neural Network, which classifies the output from convolutional and pooling layers. The neurons in the fully connected layer detect and preserve the value of a certain feature, and the output classes then check whether the feature is relevant to them. In fact, a term “dense” layer (or a fully connected layer) describes a linear layer whose every neuron is connected to every neuron in the next layer, which means that the input layer could be portrayed as a fully connected layer too.

3.2.2 Dropout

Deep neural networks, such as CNNs, often face the problem of overfitting. Overfitting happens when a neural network model learns from the training data too well, which causes its inability to correctly model new data (it does not generalize properly). A **Dropout** regularization method [12] reduces model overfitting by randomly dropping out nodes (layer outputs) during training. In CNN model architectures, Dropout is commonly used along with the fully connected layers, but it may be implemented on any layers of a network, excluding the output layer. One of the first significant CNN architectures implementing the Dropout method was the *AlexNet*, presented in the following Section.

3.3 Significant CNN Model Architectures

Over the years, many different variants of CNN model architectures have been designed to solve computer vision problems, leading to huge advances in the sphere of deep learning. At the moment, even the biggest tech companies such as Google, Facebook or Microsoft explore new architectures of CNNs through their research groups. Some of the innovative architectures developed so far are presented in this Section.

3.3.1 LeNet-5

The **LeNet-5** CNN model [13], presented already in 1998, was used for handwritten and machine-printed character recognition. Taking a 32×32 px grayscale image on the input (typically a handwritten digit), two couples of convolutional and pooling layers are designed to generate 16 feature maps of the 5×5 size. These are then fed into the fully connected part of the network ended with the softmax activation function making the predictions. The LeNet-5 architecture is described in detail in Figure 3.7.

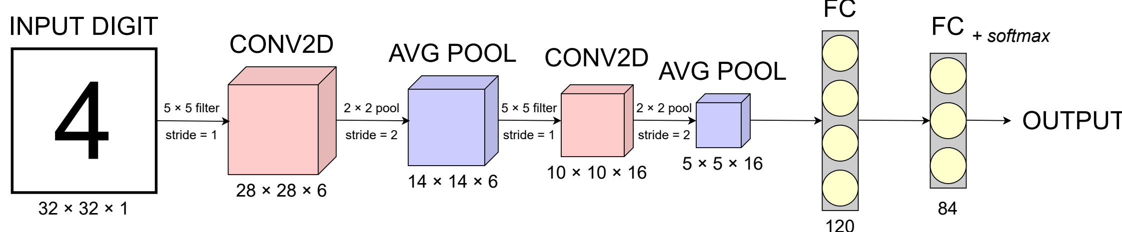

Figure 3.7: Schematic of the LeNet-5 CNN model architecture [13] primarily designed for digit recognition. It takes a 32×32 px grayscale image on the input, which is passed through two combinations of a convolutional and a pooling layer, each of them reducing its dimensions. The convolutional layers use 5×5 filters performing convolution while sliding over the image one pixel at a time. An average pooling, returning average values from 2×2 windows, reduces the feature map size into a half. 16 feature maps of the 5×5 size are then flattened into 400 nodes. The first fully connected layer connects all of these 400 nodes with every one of 120 neurons, which are then similarly connected to every of 84 nodes of the second layer. The softmax activation function is used at the end predicting the output digit.

3.3.2 AlexNet

In 2012, the **AlexNet** architecture [14] dominated the ImageNet ILSVRC challange [15] and started a wave of interest in Convolutional Neural Networks. It is one of the first models stacking convolutional layers directly on top of each other without inserting pooling layers between them. In addition, AlexNet implements the Dropout regularization method to reduce overfitting in the fully connected layers. The network consists of 5 convolutional layers, supplemented by 3 max pooling layers, and 3 fully connected layers, as shown in Figure 3.8. Thus, AlexNet is quite similar to LeNet-5, even though much larger (it has about 56 M trainable parameters compared to the 60 k in case of LeNet-5).

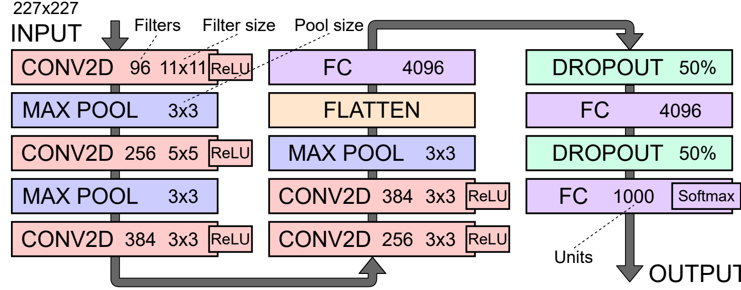

Figure 3.8: Schematic of the AlexNet Convolutional Neural Network architecture [14]. The model contains five convolutional layers, whose output is passed through the ReLU activation function, combined with three max pooling layers together taking care of feature extraction. During this process, the image resolution is reduced from 227×227 px to 6×6 px. Then the output is flattened and passed to the second part of the network managing the classification by three fully connected layers, coupled with two Dropout layers randomly dropping 50 % of outputs each. The softmax activation function makes prediction at the output layer.

3.3.3 VGG-16

Since the international success of the AlexNet, CNNs were starting to get deeper, and the simplest way of improving the deep networks performance is by increasing their size. Visual Geometry Group (VGG) presented the **VGG-16** [16], consisting of 16 weighted layers (convolutional and fully connected layers) and 136 M parameters in total. Due to this, the model deserves about 500 MB of storage space.

The remarkable thing about VGG-16 is despite it is a really deep network, it is easy to understand, because all of its convolutional layers use filters of the same size, "same" padding and stride 1, and all of the pooling layers use 2×2 pool windows with a stride of 2 (see Figure 3.9 for the architecture visualization). Besides, VGG designed an alternative to VGG-16, called VGG-19, which is even deeper, containing 19 weighted layers in total.

3.3.4 ResNet

Last but not least, in 2015, Microsoft Research proposed another successful model, an extremely deep network called **ResNet** [17]. ResNet is composed of 152 layers and it is well-known for the introduction of *residual blocks*, which are stacked on top of each other to form the network. Training such a deep neural network is possible thanks to the *skip connections* allowing to take the activation from one layer and feed it to another one, which can be placed even much deeper in the network.

ResNet dominated the 2015 ImageNet ILSVRC challenge [15] by reaching an amazing $3,57\%$ top-5 error rate. By comparison, AlexNet won this competition in 2012 with a top-5 error rate over 15 %, and the results presented in a research [18] mention that human top-5 classification error rate on the large scale ImageNet dataset [19] is only 5.1 %.

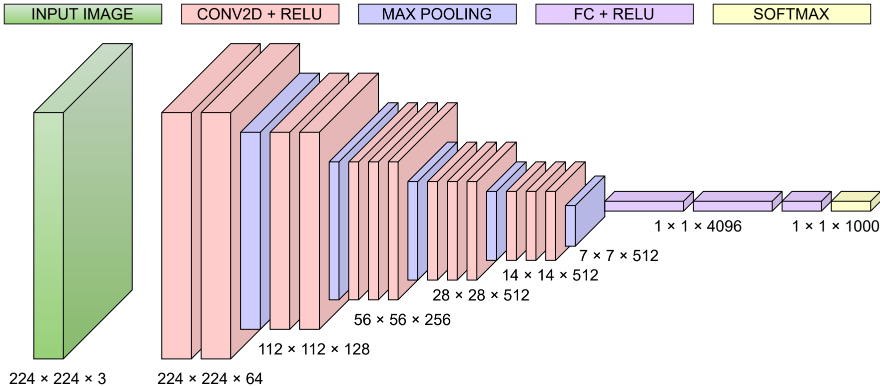

Figure 3.9: Schematic of the VGG-16 Convolutional Neural Network architecture [16] consisting of 13 convolutional layers, 5 max pooling layers and 3 fully connected layers ended with the softmax activation function. A 224×224 px input image is gradually transformed into 512 7×7 feature maps. All of them must be flattened into one long vector before feeding into the first fully connected layer, which clearly demonstrates the hugeness of the network. The model achieved nearly 93 % test accuracy when classifying more than 14 M images to 1000 classes (at the ImageNet ILSVRC challange [15]).

Chapter 4

Image Classification Process

Having the knowledge of how CNNs work and how does their architecture look like is only a half of the information required to understand the issues dealt with in this thesis. In the first half of this Chapter, a general workflow of the neural network training process, including the explanation of some fundamental terms related to it, is presented. An insight into how are the neural networks trained is essential, in order to understand the image classification process as a whole, which is described in the second half of this Chapter.

4.1 Introduction to the Neural Network Training Process

Before describing of how to classify images using the TensorFlow platform, it is important to know what does it mean "to train" a neural network. In this Section terms as a *loss function*, *cross-entropy*, the *Gradient Descent* or a *learning rate* are briefly explained. Most of them are discussed in the following Chapters and some are even involved in the experiments made, therefore, their meaning must be clarified.

4.1.1 Neurons and Weights in a Neural Network

Artificial Neural Networks consist of units, called **neurons** (by inspiration of biological processes in a human brain). These neurons are just mathematical functions taking some values on the input and giving a certain output. The important thing is that a neuron assigns a **weight** for each input value. The weights are the parameters that change during the model training process (while during testing they keep fixed values) in order to tune the network.

More specifically, a neuron is a linear combination of inputs and weights with an activation function on top of it.

$$f(x_1, x_2) = w_1 x_1 + w_2 x_2 \tag{4.1}$$

The Equation (4.1) above illustrates the linear combination of two inputs x_1 and x_1 which are multiplied by corresponding weights w_1, w_1 and summed up.

A neural network is just a bunch of neurons (linear functions) connected to each other, but when stacking a linear function on top of each other, the output is still a linear function. In other words, without adding non-linearity to the network it would not be more powerful that a single neuron. A non-linearity is applied through an activation function which was already explained in Section 3.1. For the last layer of neurons it could be sigmoid, for example, and for the previous layers it is typically ReLU.

4.1.2 Loss Functions

Unfortunately, it is not possible to calculate the weights perfectly. Therefore, the neural network learning process is rather an optimization problem of finding an algorithm discovering the possible sets of weights that the model could use to make predictions. For each prediction made by a model, an error is calculated, specifying the deviation between the predicted and the actual output. The error values are further used to penalize the model for wrong predictions. Since the aim is to minimize the error value, it is generally referred to as a **loss**, and the function calculating it is called a **loss function**.

Cross-entropy Typically, the **cross-entropy** loss function is used for classification problems (when the outputs are members of a certain class with a given probability) together with the sigmoid, or the softmax activation function at the model output layer. The penalty, provided by the cross-entropy loss function, is logarithmic, meaning that for small differences between the actual and the predicted values, for instance, 0.1, the score is quite small, but for large differences, such as 0.8 or 0.9, the penalization score is very high.

Cross-entropy can be calculated for both binary classification problems and multi-class classification problems. In the case of binary classification problems (when there are only two possible classes an output can belong to), cross-entropy is calculated as the average cross-entropy across all samples. The **binary cross-entropy** loss function, defined in the Equation (4.2), is typically used together with the sigmoid activation function.

$$Loss = -\frac{1}{m}\sum_{i=1}^{m} y_i \cdot \log \hat{y}_i + (1 - y_i) \cdot \log(1 - \hat{y}_i) \tag{4.2}$$

When doing the multi-class classification, the classes labels are often one-hot encoded (as shown in Fig. 4.1), meaning that each class is represented by a binary feature (either 1 or 0). In this case, the **categorical cross-entropy** loss function is used. The only activation function recommended to use together with it, is softmax. A separate loss for each class label per observation is calculated and the results are summed up. Categorical cross-entropy is defined in the following Equation (4.3).

$$Loss = -\sum_{i=1}^{M} y_i \cdot \log \hat{y}_i \tag{4.3}$$

In both of the Equations: (4.2) and (4.3), the M symbol stands for the number of classes, the $\hat{y}_i$ is the i-th predicted output value and the y_i is the i-th actual target value.

4.1.3 Gradient Descent

As mentioned, training a neural network is an optimization problem, requiring an algorithm that optimizes functions. Gradient-based algorithms use the gradient of the information provided by a function. In machine learning, a gradient is a vector pointing to the direction of the steepest increase of a loss function. Since the goal is to minimize the loss function, a meaningful approach is to take a step in the opposite direction of the gradient.

Model weights represent the variables with respect to which the derivatives are computed, because the weights are the values whose changing could improve the network performance. Computing the gradient of a loss function with respect to weights and taking tiny steps in the opposite direction of the gradient, gradually decreases the loss. This is the

	INTEGER ENCODING	ONE-HOT ENCODING
CLASS A	1	[1, 0, 0]
CLASS B	2	[0, 1, 0]
CLASS C	3	[0, 0, 1]

Figure 4.1: There are two options of how class labels can be represented in code. One-hot encoding means that each category value is converted into a column and a 1 or 0 value is assigned to the column. The alternative is to use the classic integer encoding.

principle of the **Gradient Descent** algorithm. The Gradient Descent updates the model weights by the following rule:

$$w_i = w_i - L \cdot \delta \frac{l}{\delta w_i} \tag{4.4}$$

For each weight w_i: the derivative with respect to it $\delta \frac{l}{\delta w_i}$, multiplied by the learning rate L, is subtracted from it. The **learning rate** is an important model hyperparameter, controlling the size of a step which is taken each iteration.

However, computing the derivatives deserves going through each dataset sample, and doing this every iteration of the Gradient Descent would be extremely inefficient. For instance, the *Mini-Batch Gradient Descent* algorithm approximately estimates the derivative on a small batch of samples from the dataset. The **batch size**, defining the number of samples taken from a dataset, is another hyperparameter used for model tuning. Generally, many different optimization algorithms are used to to train a neural network model, but most of them are just variations of the Gradient Descent.

4.2 Image Classification on the TensorFlow platform

With the understanding of how Convolutional Neural Networks process and classify images and with the insight into the neural network training process, the only thing left to describe is the workflow of the image classification process itself. As all the machine learning done in this project is handled by the TensorFlow platform and the Keras API (Application Programming Interface), the general image classification principles are combined with some Keras implementation examples in this Section.

4.2.1 Work Tools

There are various machine learning platforms handling the image recognition and classification. In this work, the image classification process is in a full control of the TensorFlow platform [10, 11] (v2.3 and higher) and its associated deep learning API Keras [20]. I use Keras for dataset loading and configuration, as well as for the CNN model definition and its subsequent training. Thanks to the TensorFlow's `tf.summary` module I am able to

write summary data and visualize it through the TensorBoard toolkit[1] (an example of a visualization is shown in Figure 4.2).

Apart from that, I capture video frames through the OpenCV library [21, 22] and form a dataset from them. Building a dataset using OpenCV is described in Chapter 5 in detail.

All of the scripts are written in the Python[2] programming language.

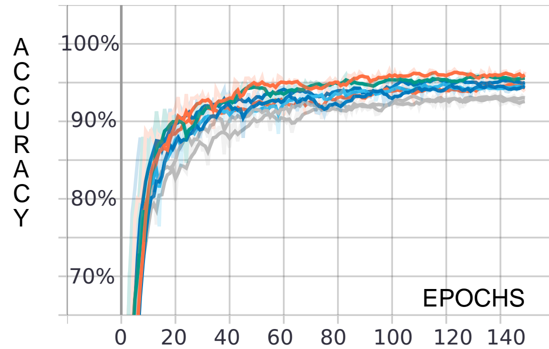

Figure 4.2: Example of the training process visualization in TensorBoard. This particular diagram shows eight validation accuracy curves, each representing a single model training process. The x-axis represents the training epochs, while the y-axis shows the accuracy. The TensorBoard's Scalars Dashboard allows a user to visualize many different accuracy or loss curves at once, filter out the unwanted ones, compare the wanted ones and find out additional information about each epoch, such as its duration or the actual accuracy value. Moreover, the curves are updated continuously during the training process, making it possible to monitor the training process and react to changes.

4.2.2 Data Preparations

Loading After obtaining and organizing data, the images are loaded off disk. In TensorFlow, an input pipeline is typically build through the `tf.data` API and a sequence of elements (a dataset, typically) is represented by the `tf.data.Dataset` abstraction.

The important thing is that some data is used for training a neural network model and some is used for its validation. The model is initially fit on a training dataset, and then used to predict the responses for the observations in a validation dataset. Apart from the training and validation data, sometimes there is a test dataset providing a final evaluation of the model. The data is usually split into training and validation parts right during the loading procedure. Besides that, the image resolution and the batch size, defining a number of images utilized in one training iteration, is specified when loading a dataset.

Configuration Before fitting a model on a data, the loaded dataset is usually slightly configured for a better performance. The `Dataset.prefetch()` transformation method enables to create an overlap between the data being pre-processed and the model execution while training, allowing the later pre-processed data to be prepared while some other data is being processed. Moreover, the `Dataset.cache()` method ensures that during the first training epoch, the data is kept in memory (cached) for the subsequent iterations to use it, in order to prevent the dataset becoming a bottleneck during the training process. A

[1] https://github.com/tensorflow/tensorboard
[2] https://docs.python.org/3/reference/

common approach is also normalizing the data to ensure that each image pixel has a similar data distribution, which makes the network a bit faster.

Augmentation Lastly, augmentation techniques might be applied to the data. When there is a small number of training examples, or when there are a lot of similar images in a training dataset, overfitting usually occurs. Data augmentation generates additional training data from the existing samples by using various transformations, such as rotating or cropping the image, or changing its color representation. A model can then expose more aspects of the image and thus, generalize better.

4.2.3 Building a Model through the Keras API

Once the data is prepared, it is essential to build a neural network model. The Keras API integrated within TensorFlow provides the `tf.keras.Seuquential` class grouping a linear stack of layers into a `tf.keras.Model` (an example is shown in Fig. 4.3). A CNN model implemented through the `tf.keras.Seuquential` class typically consists of several combinations of convolutional and pooling layers followed by some dense layers (which corresponds to the CNN architecture described in Chapter 3). In each of these layers, parameters, such as filter size and strides for convolutional and pooling layers, or number of units in fully connected layers, are specified. Activation functions can be either defined in separate layers, or as a parameter of convolutional or dense layers.

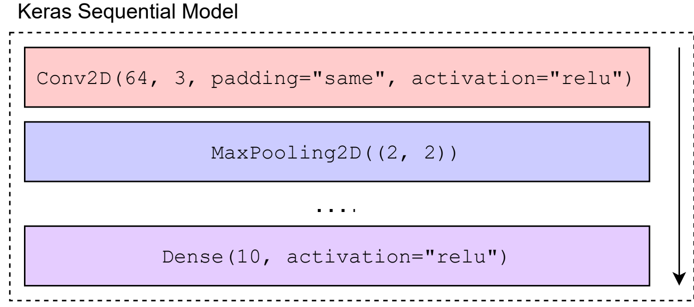

Figure 4.3: An example of several layers grouped into the Keras Sequential Model. The convolutional layer defines 64 filters of size 3×3 and "same" padding, the max pooling layer uses the pool size 2×2, and the dense layer consists of 10 units/neurons. In this example, the activation functions are not defined as separate layers, instead they are set as parameters of the other layers.

4.2.4 Loss, Optimizer and Performance Metrics

Right before the training process starts, a loss function, an optimizer and a metric are specified. In case of the image classification, a **loss function**, calculating the loss and penalizing the model, is typically some kind of cross-entropy. When doing the binary classification, the *binary cross-entropy* is used, and for the multi-class classification, it is usually the *categorical cross-entropy*, or the *sparse categorical cross-entropy*.

Both, categorical cross-entropy and sparse categorical cross-entropy have the same loss function. The difference between them is that in case of the categorical cross-entropy, the

class labels must be one-hot encoded, while when using the sparse categorical cross-entropy, the class labels are represented as integers.

The term **optimizer** refers to an algorithm used to update the attributes of a neural network, such as weights or the learning rate, in order to reduce loss. The Gradient Descent, presented in Section 4.1, is the most basic optimization algorithm. Perhaps the most trending optimization algorithm used for deep learning applications is called Adam [23], which stands for the Adaptive Moment Estimation. Adam is a faster and more efficient extension of the basic Gradient Descent algorithm.

The metrics evaluate the neural network model performance. One of the commonly used performance metric for classification problems is the **accuracy** metric calculating how often predictions match the labels.

4.2.5 Model Training and Evaluation

The training process itself happens in **epochs** (an epoch is a single pass of all data to the network). The batches of training data are fed to the network one-by-one, followed by the validation data batches, every epoch. For each batch, its samples are used to estimate the error gradient, which is subsequently used to update the model weights. At the end of each epoch, the learning rate, specifying how much are the model weights being updated, is recalculated.

The `tf.keras.Model` class provides several methods for model training and evaluation, such as `Model.fit()` for training the model in a fixed number of epochs, `Model.predict()` for generating output predictions, or `Model.evaluate()` for predicting outputs and even computing the evaluation metrics. However, despite these methods are quite practical and allow to train and evaluate a model in just a few lines of code, they do not provide a full control of the training process. Thankfully, it is also possible to write a custom training loop in TensorFlow.

Chapter 5

Forming the Dataset

With the help of my supervisor, I collected Yoga videos and built a custom dataset from them. I designed an application in which the videos are manually annotated, and the annotations are then used to form the Dataset. The process of building the Dataset, including the presentation of the annotation application, as well as the shape of the Dataset itself are described in this Chapter.

5.1 Collecting Videos and Annotating them in a Custom Annotation Application

I collected 162 Yoga videos containing three different Yoga sequences. Unfortunately, these videos come from only two people, which is a too small sample size to form a truly comprehensive dataset. But for this work, the goal is the proof-of-concept (verifying the very potential of the idea) and that purpose the Dataset serves well.

5.1.1 Collecting the Yoga Videos

Despite the 162 videos come from only two subjects, and the fact that each subject always exercises in a same room, each of the videos is unique. Most of the Yoga sessions are recorded by multiple cameras (up to four) positioned in various angles. Furthermore, almost for every session the subjects wear clothes of different colors and usually change the part of a room where practicing, so that the video background is altered. The light conditions for some sessions vary too, but care should be taken to ensure enough light and not much shadow in a footage. Thanks to all these aspects the data is quite varied, although the number of people producing the Yoga videos is very low.

As mentioned, three predefined Yoga sequences are covered: namely the **Cat Sequence**, the **Sun Salutation** and the **Warrior III**. As the Sun Salutation is the most complex one, because it consists of the most Yoga poses (12 from the total 22), nearly a half of the 162 videos are the Sun Salutation Yoga sequences. The Sun Salutation sequence, presented already in Figure 2.1, is unfortunately the only sequence performed by both of the executors. The **Mountain** pose is the only Yoga pose performed in two of these Yoga sequences (in both Sun Salutation and Warrior III).

5.1.2 Annotating the Yoga Videos in a Custom Application

The dataset for the training itself consists of images, not videos. Therefore, I created a tool (Fig. 5.1) where the videos are manually annotated. The annotations define exact frame numbers, at which are the training and validation frames are being captured.

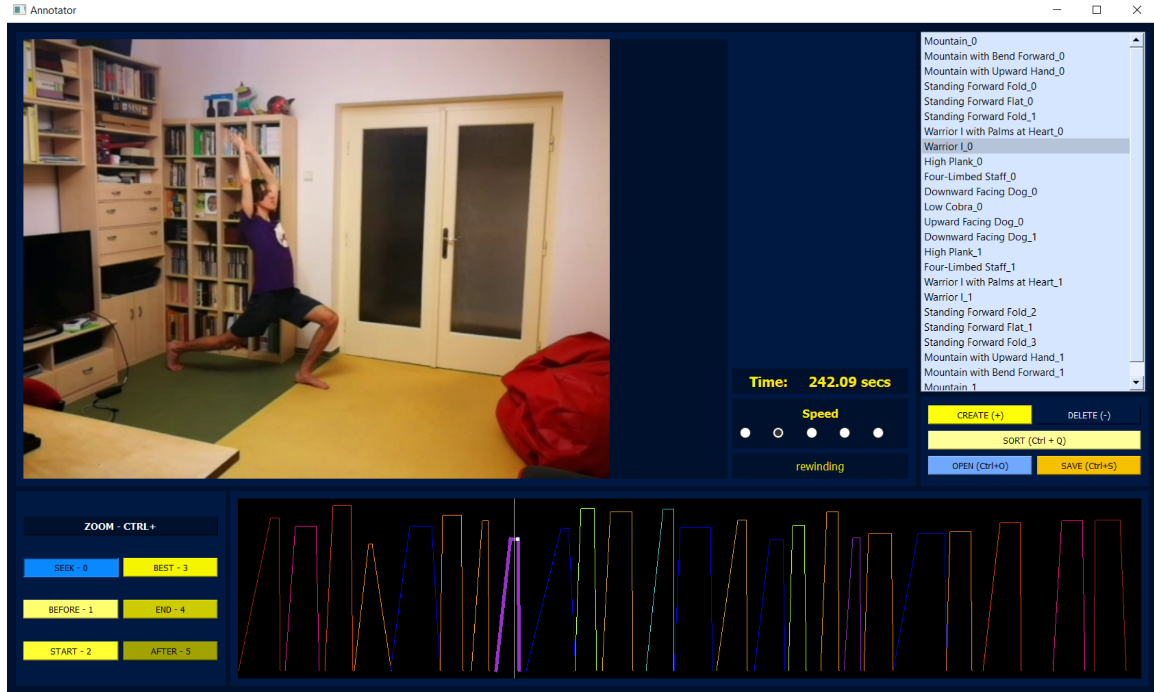

Figure 5.1: Screenshot of the annotation application workplace. In the top right corner there is a list of annotations belonging to the currently processed video. The annotations are created by the yellow buttons at the bottom left and visualized by graphs in the timeline. In addition to that, the application supports video rewinding and speed changes and provides several other features such as video seeking and timeline zooming.

In the video, which the application takes on the input, a person performs one of the predefined Yoga sequences, each of them containing several Yoga poses. And each of the poses is represented by an annotation. This means that when a pose is performed three times during its corresponding sequence, there is a single annotation for each of them.

The annotations (Fig. 5.2) consist of a pose identifier and five numerical values representing the video frames defining the time a person is performing the corresponding Yoga pose. In the app (Fig. 5.1), the frames are marked by the buttons in the bottom left corner and the annotations are then displayed in the video timeline.

Once done and the progression is saved, a `.json` file, named by the processed video file, is created. There is a single file for each annotated video, storing all of the annotations related.

5.1.3 Using the Annotations to Form the Dataset

The `.json` files storing the annotations are passed to a script randomly taking screenshots at the specified frames using the OpenCV library, which forms the Dataset. How is this done?

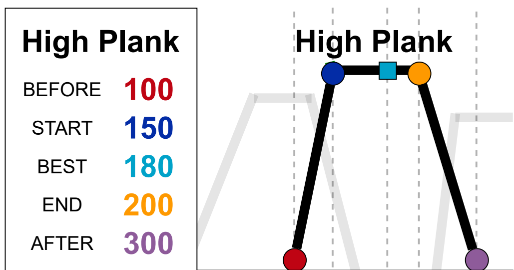

Figure 5.2: Example of an annotation referring to the **High Plank** Yoga pose. The graph on the right portrays its corresponding representation in the timeline. All of the frames between *START* and *END* symbolize the time a person is performing the pose, and therefore, the screenshots can be taken during this period. The frame intervals between *BEFORE* and *START* and between *END* and *AFTER*, as well as the *BEST* frame are not utilized for now, but could be useful in a future work.

Firstly, an integer value specifying the number of output image frames for each of the Yoga poses is defined (for each Yoga pose the same number of frames is extracted). After loading all the `.json` files and storing the data from the annotations in a memory, the script collects all the appropriate frames and groups them by videos and by the Yoga poses. The grouping allows the script to randomly choose the same (or almost same) number of frames for each video, in order to ensure a balanced representation of the videos in the Dataset. Lastly, the script iterates over all the Yoga videos (named equally to the `.json` files) and using the OpenCV library it generates image frames by the randomly chosen frame numbers. The complete Dataset forming process is then depicted in Figure 5.3.

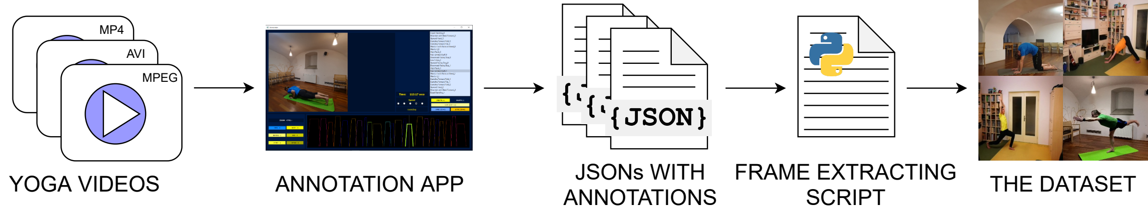

Figure 5.3: Workflow of the Dataset forming process. Yoga videos are annotated in the annotation application. The resulting annotations, stored in `.json` files, define possible frame numbers at which a script extracts the frames from videos. The script captures frames, randomly chosen from the defined frame numbers, which forms the Dataset.

5.2 Data Shape and Volume

Before annotating, the Yoga videos are split into two directories separating the training and the validation data by 3 : 1. This is done at this early level, because of the potential to manually choose the source of pictures used for model training (I am able to build both easy and hard datasets). See Section 7.1.1 for more information about "hard" and "easy" datasets.

It should be pointed out that every single video always serves as the source for either the training dataset, or the validation dataset (never for both of them). The frame capturing

script walks through the directory containing the video files iteratively, meaning that all of the classes hold the same amount of data.

I work with the Dataset containing 44 000 images (2 000 per class – Yoga pose), but thanks to the frame range defined in the annotations, I am able to create a Dataset carrying hundreds of thousands of pictures. The frames are captured with the fixed resolution 256×256 (squared) and resized at a later stage, when the Dataset is being configured for performance, into an appropriate shape. A few Dataset samples are shown in Figure 5.4.

However, not all of the 44 000 images are further used in the training process. The frame extraction script serves only as a giant image collection provider and this collection is updated only when new videos are being integrated into the dataset forming process, which is done occasionally. Moreover, I often alter the Dataset size and structure, and it takes more than an hour to walk through the videos and capture frames. Therefore, I truly work with 22 000 images randomly chosen from the 44 000 before every set of experiments.

Figure 5.4: Samples from the Dataset. The videos are squared (or trimmed to square if captured otherwise). The subject should be visible during the whole session. A defined set of Yoga sequences was performed when shooting the videos.

As mentioned above, the training and validation data is split by 3 : 1, which means that the training dataset contains 16 500 Yoga images (750 per pose) and the validation dataset consists of 5 500 Yoga images (250 per pose).

Chapter 6

Training Convolutional Neural Network Model for Yoga Pose Recognition

A complete workflow of how the Dataset (presented in the previous Chapter) is loaded, configured and integrated into the training process is described in Section 6.1. The actual augmentation techniques applied to the training data are presented too.

The other part of this Chapter explains how is the CNN Model being built, how does the actual training loop look like, and how are the training parameters set. Apart from that, a description of which data is being summarized and visualized during the training process and which utilities are used for it, is covered.

6.1 Dataset Loading, Configuration and Arrangement

6.1.1 Getting the Data

In preparation for model training, the images are loaded off disk using the *image_dataset_from_directory* utility, provided by the `tf.keras.preprocessing` module. The Dataset is already divided into training data and validation data, as the videos come from two different directories. The frame resolution remains at the 256×256 px and the image batch size is usually set to 32. The labels are encoded as a categorical vector (one-hot encoded).

6.1.2 Performance Configurations

After loading, the data is normalized to unify the data distribution of the pixels and then resized. The input in the $[0, 255]$ range is rescaled to fit the $[0, 1]$ range. The images are mostly resized to 96×96 px. I mainly operate with this resolution, because it provides a decent image quality and a reasonable size at the same time (training a network on 224×224 px data, for instance, is a compute-intensive task). However, a higher image resolutions bring out better results (which is demonstrated in the following Chapter 7.

In addition, the `Dataset.cache()` method caches the Dataset samples during the first training epoch and the input pipeline ends with a call to the `Dataset.prefetch()` transformation. Both of these methods are explained in Section 4.2.2.

6.1.3 Data Augmentation

To fight overfitting in the training process and increase the diversity of the Dataset, I apply various augmentation techniques to the images (see augmented samples in Fig. 6.1). These transformations cover picture rotating, horizontal flipping and several image color enhancements, namely contrast, brightness, saturation and hue changes. The central region of some pictures is cropped, too.

Most of the techniques are realized using the `tf.image` module. For frame rotations I utilize the `tfa.image` one belonging to the TensorFlow Addons repository. All of these techniques are combined together and applied to each image batch of the training data just before the first training epoch starts (for each epoch there are different transformations applied to the training data). The validation data is not being augmented.

Figure 6.1: Augmented image samples. Each of the pictures is taken from a different batch, as all the transformations applied are same for the images in a single batch. The augmentation techniques used include color changes, rotations, flipping, and a central crop.

Data augmentation proved itself to be very useful in fighting the model overfitting. Due to this, I widely experimented with all the individual transformation techniques (rotations, color changes, and cropping), in order to set their parameters as precisely as possible. The results of the experiments made with the augmentation techniques are presented in Section 7.1.5.

6.2 CNN Model Training Process

6.2.1 Model Building

I build a model through the `tf.keras.Sequential` class (presented already in Section 4.2.3). In the model training script, a custom method groups the individual layers together and sets their parameter values based on the input command line arguments.

The models I experiment with consist of six to ten convolutional layers combined with four or five max pooling layers, followed by no more than three fully connected layers. I try to design the architectures as simple as possible, although the highest priority is the model efficiency of course. As for the activation functions, I majorly use the Rectified Linear Unit (ReLU) at the hidden layers and switch between softmax and sigmoid at the output layer. An example of one of the most successful model architectures (according to the experiments made) is shown in Figure 6.2.

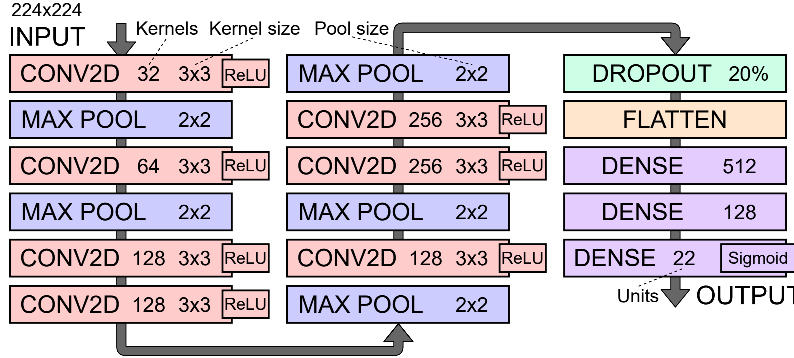

Figure 6.2: Schematic presenting one of the most successful CNN models I found during the experiments. It consists of seven convolutional layers, whose output is passed through the ReLU activation function, managing the feature extraction together with five max pooling layers, which consecutively reduce the image dimensionality from 224×224 px to 7×7 px. Before the final output is flattened and fed to the first fully connected layer, 20 % of the outputs are dropped (chosen randomly), in order to reduce overfitting. The Model contains three fully connected layers (called **Dense** in the Keras Sequential API). The sigmoid activation function is used at the output layer to make predictions. This Model operates with 7, 7 M trainable parameters (or 2, 5 M for the 96×96 px input image dimensions). The capability of this model to detect Yoga poses is shown in Fig. 7.6

6.2.2 Setting the Metric, the Loss Function and the Optimizer

For model evaluation I use the **categorical accuracy** performance metric calculating how often predictions match one-hot labels, as the data is represented as one-hot Tensors.

Since the labels are provided in the one-hot representation, I use the *categorical cross-entropy* loss function to compute cross-entropy loss between labels and predictions. However, sometimes I replace it with the *binary cross-entropy* loss function combined with the sigmoid activation function at the Model output layer. More details about experiments with cross-entropy loss functions are listed in Section 7.1.2.

On top of that, I use Adam [23] as the optimizer used to update the model weights and the learning rate.

6.2.3 Model Training

I design a custom training loop. As mentioned, the training process happens in epochs. Each epoch, these steps are done:

1. Passing a new learning rate value as an argument to the Adam optimizer.
2. One iteration over all the batches of training data.
3. One iteration over all the batches of validation data.
4. Recalculating the learning rate value.

Updating the Weights and Training the Model During each iteration over all the *training data* batches, the model is being trained. For each batch, the `tf.GradientTape` scope is opened, in which the model is called and loss is computed. Then, the gradients of the weights of the model with respect to the loss are computed (outside the scope already). Finally, the optimizer updates the model weights based on the gradients.

During each iteration over the batches of *validation data*, the model is called to make predictions and the loss is computed too. However, the gradients are not computed and the model weights are not being updated again, because the model has already been trained on the training data. Therefore, the `tf.GradientTape` scope is not needed when iterating over the validation data batches.

Updating the Metrics In parallel with the model training process, the metrics monitoring is happening. For each batch of the training data, the model accuracy is being updated based on the values of the actual and the predicted class labels. After iterating over all the training data batches, the metric state is cleared, so that the same accuracy updating (and then resetting) can be done for the validation data.

The loss values are also stored (in a variable) for each batch of the training and validation data. An average loss (per batch) is then computed and summarized.

Updating the Learning Rate At the end of each epoch, after the accuracy metric state is reset for the second time, the learning rate value is being updated. I update the learning rate by the following pattern:

$$L^{i+1} = L^i \cdot \frac{1}{\frac{L^i}{E} \cdot i + C} \tag{6.1}$$

where L^i stands for the learning rate in the ith epoch (i is the epoch number), the E represents the total number of epochs, and the C is the coefficient. The coefficient value ranges from 1.00 to 1.08. Experiments finding the optimal value are presented in Section 7.1.4.

After the new learning rate value is calculated, one training epoch ends and another begins (by passing the new calculated value to the Adam optimizer).

6.2.4 Summarizing and Visualizations

Through the `tf.summary` module, I write summary data to disk using the file writer, and visualize them in TensorBoard. I use to visualize Yoga images from preferred batches, in

order to both explore the augmented training data and see the correct and wrong label predictions on validation data.

Apart from that, scalar Tensor values (accuracy and loss) are written down every epoch, right after the accuracy metric state is updated for the last time. By doing so, I am able to visualize these metrics through TensorBoard, and therefore, track the model training process effectively.

Chapter 7

Experimental Results

I experimented with CNN model architectures (including the AlexNet and VGG-16), data augmentation techniques, and training parameters such as the learning rate, loss functions or setting the input image dimensions. Some of those led to interesting findings and helped to reach great results. In this Chapter, the experiments made and the actual results achieved are presented.

It should be noted that in some of the graphs that show a model accuracy in this Chapter, the training accuracy curves are omitted. This is because the training accuracy commonly reaches the 100% value in the experiments and, therefore, the curves are excluded from the graphs in order not to mix with the validation accuracy curves, which are more important to show.

7.1 Findings, Advancements and Model Tuning

The results, presented in the following Section 7.2, could not be reached without experimenting with the Dataset and CNN models and their parameters. In this Section, a selection of the most significant experimental findings is presented.

7.1.1 Hard and Easy Datasets

Firstly, it should be pointed out that all of the results presented in this Chapter were achieved when the videos forming the Dataset were selected precisely, in order to make the model predictions as **hard** as possible.

How is this done? For instance, all of the videos where the exercising subject wears a yellow shirt serve as the validation data source, so that the model cannot be trained on them (then, the training dataset consists of frames extracted from none of these videos).

Similarly, some experiments were done using an "easy" Dataset, containing frames from videos selected with an opposite intention (ensuring that the validation dataset source videos are all similar to the videos serving as the training dataset source, if possible). However, results reached by experimenting with this kind of a dataset are clearly **not valid**. The comparison of results reached using a "hard" and an "easy" dataset is briefly mentioned in Section 7.2.

7.1.2 Activation Functions and Loss Functions

Perhaps the most important finding I have made is related to activation and loss functions. Since I do the multi-class classification, I used to work with the *softmax* activation function, normalizing the network output to a probability distribution over predicted classes, together with the *categorical cross-entropy* loss function. But, after a consultation with my supervisor, I found out that a combination of the *sigmoid* activation function and the *binary cross-entropy* loss function is more efficient.

This finding was confirmed across many CNN models, where all the networks implementing the sigmoid activation function had about 4% higher validation accuracy, than the models using the softmax. From that moment, most of the models tested use the sigmoid activation function at the output layer instead of softmax.

7.1.3 Model Architectures

I experimented with more than a hundred CNN model architectures, and despite many of the experimental results are not much useful, it is not possible to present all of the useful ones. Nevertheless, some of the interesting results are listed below.

Convolutional and Pooling Layers Firstly, I tested the part of a CNN model dealing with feature extraction (convolutional a pooling layers). Working with the 96×96 px image resolution, I found out that the Model should contain at least four consecutive combinations of convolutional and pooling layers to make decent predictions. The validation accuracy of a model architecture not meeting these criteria is shown in Fig. 7.1.

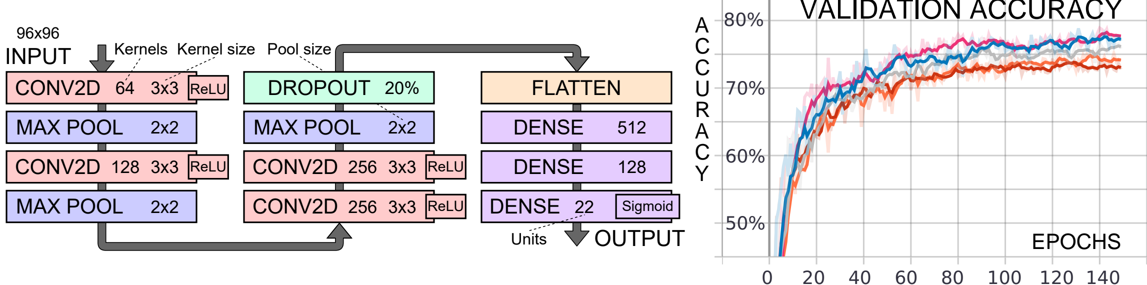

Figure 7.1: Schematic of a CNN model architecture taking 96×96 px images on the input, and containing only four convolutional layers and three pooling layers. Its validation accuracy is shown in the graph on the right. The x-axis shows the training epoch numbers and the y-axis shows the model validation accuracy. The validation accuracy reaches 75% for some of the experiments. However, compared to the accuracy of the previously presented Model (see Fig. 6.2 for its architecture and Fig. 7.6 for its accuracy), it is still quite low accuracy. The size of the feature maps, before flattening and fed them to the first fully connected layer, is 12×12 px, compared to the initial 96×96 px image dimensions. The output feature map resolution might be still too high, as the convolutional layers might have not managed to extract even the more complex features from the image. The model operates with 19,9 M parameters when using the 96×96 px input image resolution.

Besides, I ascertained that a too low number of convolutional filters leads to a worse validation accuracy. An example of this kind of model is presented in Fig. 7.2.

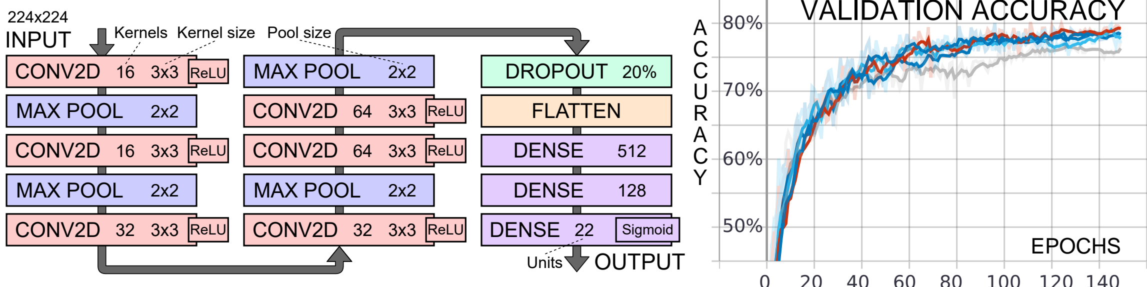

Figure 7.2: Schematic of a lightweight CNN model architecture. The low number of convolutional filters/kernels leads to a low number of trainable parameters in the network (6, 5 M for the 224 × 224 px image resolution, and only 1, 3 M for 96 × 96 px). However, this fact might be a reason of approximately 5 % worse model accuracy than a similar model with a quadrupled number of filters in each layer (64, 64, 128, 128, 256). Therefore, using such a low number of kernels may be inappropriate.

Dense Layers After that, I experimented with the number of fully connected layers and their units. Unfortunately, the outcomes were quite similar to each other (the best ones did not stand out much), meaning that this testing brought out hardly thrilling results. I only found out that two or three dense layers might be a reasonable number of layers for the CNN Model.

AlexNet and VGG-16 Apart from all those relatively basic models, AlexNet and VGG-16, both presented in Section 3.3, were tested too. The VGG-16 model architecture turned out to be probably too complicated for my unvaried data, because the network was not able to learn anything.

The AlexNet, having about 80 M parameters less than VGG-16, showed itself in a better light, as the validation accuracy reaches up to 80 % (see Figure 7.3 for the results). Nevertheless, these results are still worse than the best ones achieved by some less robust custom-build models.

7.1.4 Learning Rate

I slightly experimented with the learning rate too. Originally, I used constant values for every epoch (approximately in the range of 2×10^{-4} to 7×10^{-3}), but then I started operating with a partly custom-built algorithm updating the learning rate dynamically at the end of each training epoch. These experiments brought out better results compared to the measurements made with the static values.

The experiments lied in tuning the value of the coefficient, presented along with the learning rate updating formula in (6.2.3), as well as in finding the optimal starting learning rate value. It turned out through the experiments that the optimal values might be 1.02 for the coefficient and 1×10^{-3} for the initial value. From that moment, all of the experiments update the learning rate by the following pattern:

$$L^{i+1} = L^i \cdot \frac{1}{\frac{L^i}{E} \cdot i + 1.02} \tag{7.1}$$

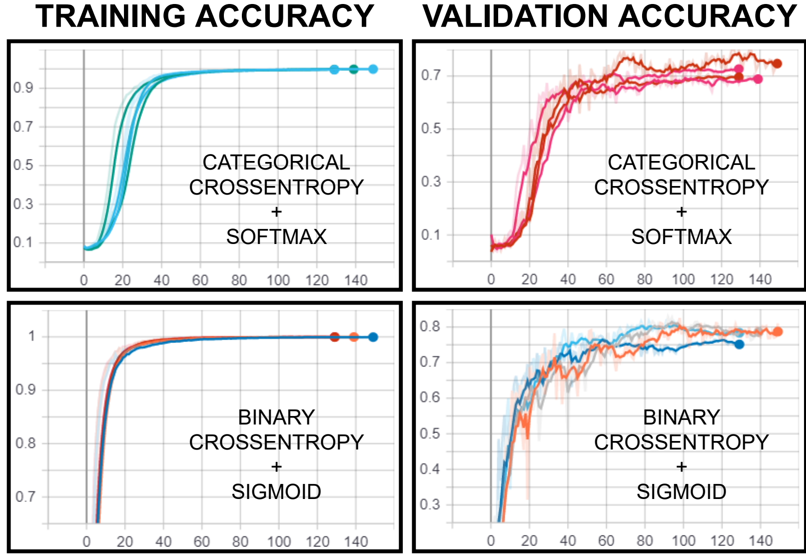

Figure 7.3: Visualization of experiments made with the AlexNet CNN model. Each diagram groups results of a few measurements together. The horizontal axis represents training epochs, while the vertical axis represents training or validation accuracy. The AlexNet originally uses the softmax activation function at the output layer, but sigmoid (combined with the binary cross-entropy loss function) was tested out of curiosity too. The model using the **sigmoid** variant reaches slightly better results. It is clear to see that the network is trained perfectly in just about 50 epochs, but the validation accuracy does not exceed 80 %. The input image dimensions were set to 96×96 px. Apparently, it does not generalize well at all.

In the Equation (7.1.4) above, the L^i stands for the learning rate in the ith epoch (i is the epoch number) and the E represents the total number of epochs. The 1.02 is the obtained coefficient value. The Table 7.1 presents, how does this pattern updates the starting learning rate value 1×10^{-3} across 100 training epochs.

Epoch	Learning Rate Value
1	1.0×10^{-3}
20	6.9×10^{-4}
40	4.6×10^{-4}
60	3.1×10^{-4}
80	2.1×10^{-4}
100	1.4×10^{-4}

Table 7.1: A Table showing how is the learning rate value being updated across 100 training epochs. The starting value 1×10^{-3} is updated by the pattern defined in the Equation (7.1.4).

7.1.5 Data Augmentation Techniques

I experimented with these transformation techniques: image color changes (brightness, contrast, hue, saturation), frame rotation, and cropping the image in the central region

("central crop"). Firstly, I tested each of the techniques separately, and then I combined them together.

As mentioned in Section 6.1.3, these techniques are mixed and applied to each image batch of the training data just before a training epoch starts. Thus, there are slightly different transformations applied to the training data for each image batch, every epoch.

The parameters of the transformation functions are initialized randomly (from a given range), so that the training data is varied as much as possible. For instance, when setting the maximum brightness *delta* value to 0.1, which is either added to, or subtracted from the normalized image pixel values, the image brightness of all images in a batch is increased/decreased **maximally** by 10 % (by 0% – 10%).

The experiments are made in order to tune the values of the parameters passed to each of the transformation functions. In the model training script, a custom method sets the transformation technique parameter values according to the input command line arguments.

Rotating the Images The experimental results showed that the highest model validation accuracy is achieved when the images are **rotated** maximally by 10° to 20° degrees in both directions. A graph visualizing these experiments with image rotations is presented in Fig. 7.4.

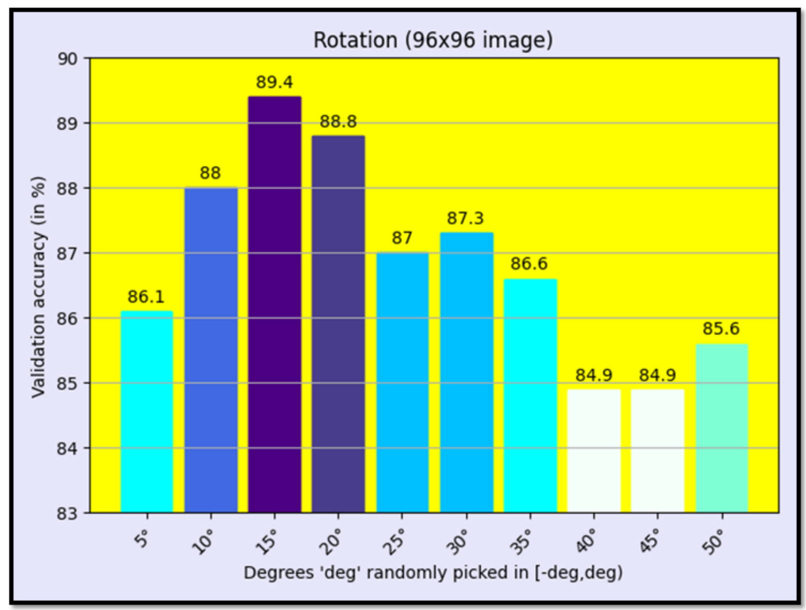

Figure 7.4: A bar chart presenting the validation accuracy reached when experimenting with the image rotation augmentation technique. In this case, the technique was tested separately, meaning that no other augmentation techniques were applied to the training data. The horizontal axis shows the value (in degrees) specifying how much are the images in a batch rotated in both directions. The vertical axis shows the average validation accuracy reached in the last 40 epochs (from the total 150). Five measurements were done for each of the values listed on the horizontal axis. The results show that rotating the training images by 0° to 15° degrees in both directions (chosen randomly for each batch) might be the optimal setting.

Brightness and Contrast Changes I found out that changing the image **brightness** just minimally (by 5 % in maximum, for instance) is the most effective setting. The validation accuracy of a model continuously decreases with increasing the delta value added to, or subtracted from the normalized image pixel values.

When the technique was tested separately, changing the brightness just by $0 - 5\,\%$ resulted in the 80 % average validation accuracy in the last 40 epochs (from the total 150), while for increasing/decreasing the brightness of the images by up to 70 %, the average validation accuracy reached only 69 %. These final values are averages of five independent experiments each. For this experiment, the input image dimensions were set to the 224×224 px.

As for the **contrast**, it can be stated that the highest validation accuracy is reached when the range, from which the random contrast values are generated for images in a batch, is set as wide as possible.

Saturation and Hue Changes Altering the **hue** value, which is done by converting an image to HSV (Hue, Saturation, Value) color model and rotating the hue channel by a delta, did not brought out thrilling results. It only turned out that switching the hue channels is effective, because the data is more varied with it.

Changing the **saturation** is done by converting an image to HSV (similarly to hue) and multiplying the saturation channel by a *saturation factor*. The experimental results showed that the range specifying the saturation factor value should just be wide enough, 0 to 30, for instance (similarly to the contrast defining range). However, the lower bound should always be a lower number than 1.0, otherwise the validation accuracy decreases quite rapidly.

Central Crop Lastly, a heatmap in Figure 7.5 shows how much and how often should be the images cropped in the central region.

7.2 Achieved Results

In the main part of this Section, the best results achieved (the highest model validation accuracy) are presented. In Figure 7.6, the advantage of a higher input image resolution is showed.

After that, a confusion matrix shows which of the Yoga poses classifies the network as similar to each other.

7.2.1 Best Results Achieved

The CNN Model, consisting of eight convolutional and five max pooling layers in total, together with three dense layers (which was already presented in Figure 6.2), reached the 100 % training accuracy in a few tens of epochs, and demonstrated its capability to detect Yoga poses by achieving the 91 % validation accuracy (as shown in Figure 7.6).

However, the 91 % were achieved using a **"hard"** dataset (the meaning of "hard" and "easy" datasets was explained in Section 7.1.1). When testing the Model on an **"easy"** validation dataset, into which videos were chosen to be similar to the videos chosen to form the training dataset, the validation accuracy reached 95 % .

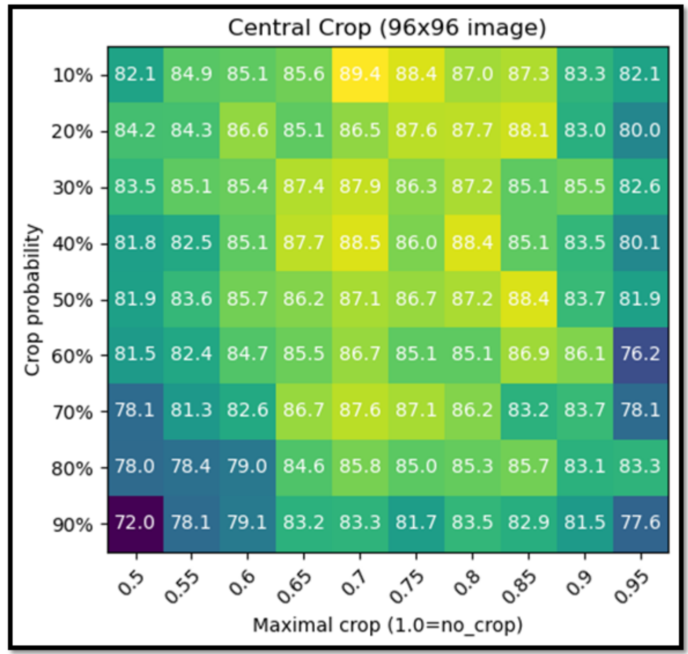

Figure 7.5: A heatmap visualizing the results of experiments with the **central crop** data augmentation technique. All images in a batch are cropped in the central region with a given probability. The individual heatmap values represent the average model validation accuracy in the last 40 epochs (from the total 150). Each heatmap value is the average value taken from five independent measurements. The values on the vertical axis represent the chance of cropping all images in a batch, while the horizontal axis shows the values specifying how much are the images being cropped (1.0 stands for no crop, while 0.5 means that the outer half of an image in cropped out). The heatmap shows that leaving about 70% of the original image (cropping out 30% of the outer region) brings out the highest model accuracy. Besides, it is clear to see that the crop probability does not affect the results much.

But, it is fair to say that the Model is able to detect Yoga poses with the 91 % accuracy when trained on this particular Dataset, because the results reached with the "easy" datasets may probably not be valid.

The sigmoid activation function coupled with the binary cross-entropy loss function, both of them involved in this experiment, confirm the fact of being such a powerful combination, at least for working with my Dataset.

7.2.2 Yoga Poses Similarity

The latest experiments done so far analyze which of the Yoga poses sees the network as similar to each other. A custom confusion matrix, showing only the wrong decisions for each pose, was designed for this purpose (see it in Figure 7.7).

The results show that all the three **"Warrior III..."** Yoga poses are often misclassified by each other. A similar trend can be observed by the **"Thunderbolt..."** poses. Both of the **High Plank** and **Low Cobra** are quite often classified as the **Four Limbed Staff** pose. Besides, the two **"Warrior I..."** Yoga poses seem interesting as they are quite often classified as each other, but hardly ever classified as the other poses.

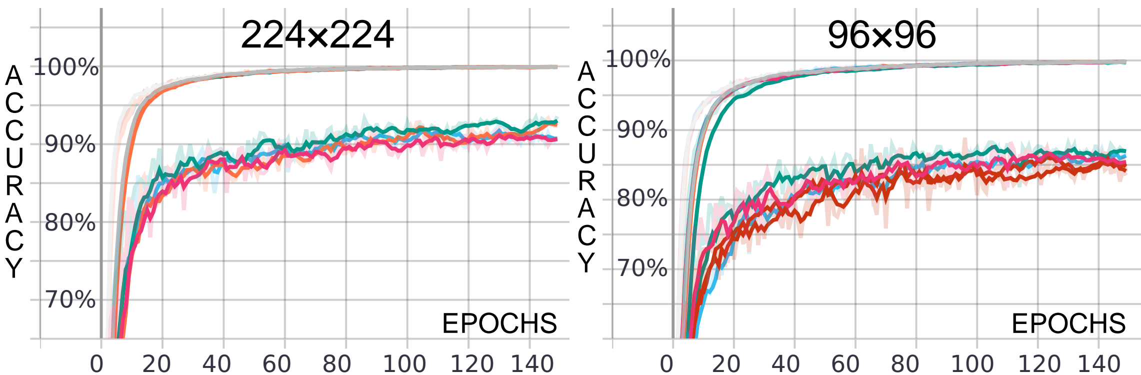

Figure 7.6: The training and validation accuracy of the CNN Model (6.2) visualized through the TensorBoard Scalar graphs. The graph on the right shows accuracy for a slightly modified version of the Model, where the input image dimensions are set to 96×96 px, instead of the original 224×224 px (graph on the left). Each of the graphs visualizes five independent measurements and for each of them, both the training and the validation accuracy are presented. Epoch numbers are showed on the horizontal axis and the accuracy is showed on the vertical one. The training accuracy, showed by the "smoother" curves, reaches the 100% value in both of the cases, meaning that the CNN Model learned perfectly on the training data. The validation accuracy presented reaches 91 % in the last 40 epochs for the original Model and 85 % for the lighter modification working with the 96×96 px input images. This Model was chosen as the best one among many others tested, because of the highest average validation accuracy during the last 40 epochs.

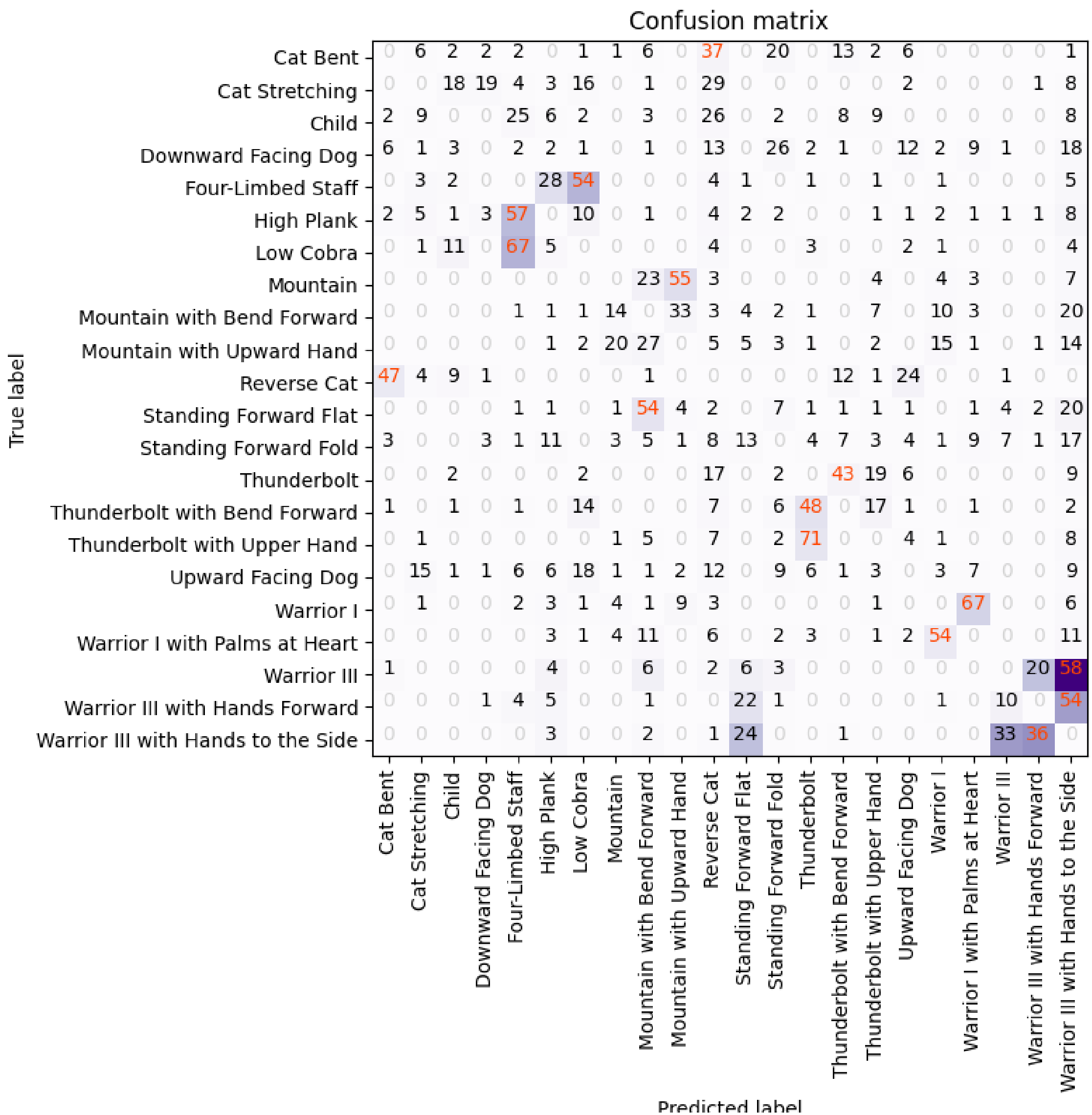

Figure 7.7: A confusion matrix, but is shows only the wrong decisions for each Yoga pose. Each value represents the number of how many times has a pose been misclassified by another one (the horizontal axis shows the predicted labels and the vertical axis shows the true labels). The aim of this visualization is only to show the possible similarity of the poses, not to measure how well are the poses being detected by a model. The results show that all the three **"Warrior III..."** poses are quite similar to each other, and the same holds for the **"Thunderbold..."** ones. **High Plank** and **Low Cobra** are often classified as **Four Limbed Staff**, while the two **"Warrior I..."** poses are similar to each other, but unlike the others. An interesting thing is that the matrix is not completely symmetric, as, for instance, the **Standing Forward Flat** Yoga pose is quite often classified as the **Mountain with Bend Forward** pose, but the **Mountain with Bend Forward** is hardly ever classified as the **Standing Forward Flat** pose.

Chapter 8

Conclusion

In this thesis, the process of forming the Dataset, including a custom video annotation tool and a frame capturing script, was described, and the shape of the Dataset itself was demonstrated. In addition, the CNN model building and training process, along with the actual methods used for it, was analyzed, step by step. In the end, experimental results and findings, including the highest accuracy achieved by the CNN Model, were presented.

I experimented with more than a hundred CNN models, of which the most successful one (presented in Figure 6.2) detects Yoga poses with the 91% accuracy (as shown in Figure 7.6), when implementing the sigmoid activation function at the output layer, instead of the traditional softmax. However, the main contributions of this project are the tools for a dataset forming process, and the Dataset itself. I collected 162 Yoga videos, designed an annotation tool providing an efficient way of creating video annotations, and wrote a script capturing video frames based on these annotations. The Dataset contains 44 000 Yoga images of 22 different Yoga poses.

I believe that the annotation application is a practical and functional tool that anyone can use for video annotating, if it fits their purposes. In addition to that, some of the findings and experimental results might also be useful for somebody.

As for this project as a whole, the objective was definitely attained. The results achieved show that a relatively simple CNN model may be actually able to detect Yoga poses successfully. Apart from that, this work gave me a complete overview of the image classification process, and taught me how to build an image dataset from videos, how to train a neural network model using a certain machine learning platform, and how to interpret the actual training results.

In the future, I am planning to continue increasing the Dataset size and improving its diversity, because by now, it still cannot be labeled as a comprehensive and varied one. Along with that, I aim to experiment with more CNN models and their parameters. It is expected that with improving the Dataset complexity, making predictions will be harder for a CNN model.

All in all, anyone can use the results achieved and try to move them a bit further. The long-term goal is designing a model capable of detecting Yoga poses confidently even on a much more complex dataset. And once done, the last step would be the desired smartphone app development and the CNN model implementation.

A paper presenting this work at the Excel@FIT 2021 student conference was awarded by software engineers from the Thermo Fisher Scientific tech company.

Bibliography

[1] Wei, S., Ramakrishna, V., Kanade, T. et al. Convolutional Pose Machines. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 2016, p. 4724–4732. DOI: 10.1109/CVPR.2016.511. Available at: https://doi.org/10.1109/CVPR.2016.511.

[2] Ramakrishna, V., Munoz, D., Hebert, M. et al. Pose Machines: Articulated Pose Estimation via Inference Machines. In: Fleet, D., Pajdla, T., Schiele, B. and Tuytelaars, T., ed. *Computer Vision – ECCV 2014*. Cham: Springer International Publishing, 2014, p. 33–47.

[3] Yadav, S., Singh, A., Gupta, A. et al. Real-time Yoga recognition using deep learning. *Neural Computing and Applications*. december 2019, vol. 31, p. https://link.springer.com/article/10.1007/s00521–019. DOI: 10.1007/s00521-019-04232-7.

[4] Hochreiter, S. and Schmidhuber, J. Long Short-Term Memory. *Neural Comput.* Cambridge, MA, USA: MIT Press. november 1997, vol. 9, no. 8, p. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735. ISSN 0899-7667. Available at: https://doi.org/10.1162/neco.1997.9.8.1735.

[5] Cao, Z., Hidalgo, G., Simon, T. et al. OpenPose: Realtime Multi-Person 2D Pose Estimation Using Part Affinity Fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 2021, vol. 43, no. 1, p. 172–186. DOI: 10.1109/TPAMI.2019.2929257.

[6] Islam, M. U., Mahmud, H., Ashraf, F. B. et al. Yoga posture recognition by detecting human joint points in real time using microsoft kinect. In: *2017 IEEE Region 10 Humanitarian Technology Conference (R10-HTC)*. 2017, p. 668–673. DOI: 10.1109/R10-HTC.2017.8289047.

[7] Trejo, E. W. and Yuan, P. Recognition of Yoga Poses Through an Interactive System with Kinect Device. In: *2018 2nd International Conference on Robotics and Automation Sciences (ICRAS)*. 2018, p. 1–5. DOI: 10.1109/ICRAS.2018.8443267.

[8] Dhanyal, S. S. and Nandyal, S. S. SEGMENTATION AND DETECTION OF YOGA ASANA FROM VIDEO. *Studia Rosenthaliana (Journal for the Study of Reserach)*. Aiwan-E-Shahi Area, Kalaburgi-585102, Karnataka, India: Department of CSE, PDA College of Engineering. june 2020, vol. 12. ISSN 1781-7838.

[9] Otsu, N. A Threshold Selection Method from Gray-Level Histograms. *IEEE Transactions on Systems, Man, and Cybernetics.* 1979, vol. 9, no. 1, p. 62–66. DOI: 10.1109/TSMC.1979.4310076.

[10] Abadi, M., Agarwal, A., Barham, P. et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.* 2015. Software available from tensorflow.org. Available at: https://www.tensorflow.org/.

[11] Ramsundar, B. and Zadeh, R. B. *TensorFlow for Deep Learning: From Linear Regression to Reinforcement Learning.* 1stth ed. O'Reilly Media, Inc., 2018. ISBN 1491980451.

[12] Srivastava, N., Hinton, G., Krizhevsky, A. et al. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *J. Mach. Learn. Res.* JMLR.org. january 2014, vol. 15, no. 1, p. 1929–1958. ISSN 1532-4435.

[13] Lecun, Y., Bottou, L., Bengio, Y. et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE.* november 1998, vol. 86, no. 11, p. 2278–2324. DOI: 10.1109/5.726791.

[14] Krizhevsky, A., Sutskever, I. and Hinton, G. E. ImageNet Classification with Deep Convolutional Neural Networks. In: Pereira, F., Burges, C. J. C., Bottou, L. and Weinberger, K. Q., ed. *Advances in Neural Information Processing Systems 25.* Curran Associates, Inc., 2012, p. 1097–1105. Available at: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[15] Russakovsky, O., Deng, J., Su, H. et al. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV).* 2015, vol. 115, no. 3, p. 211–252. DOI: 10.1007/s11263-015-0816-y.

[16] Simonyan, K. and Zisserman, A. Very Deep Convolutional Networks for Large-Scale Image Recognition. In: Bengio, Y. and LeCun, Y., ed. *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings.* 2015. Available at: http://arxiv.org/abs/1409.1556.

[17] He, K., Zhang, X., Ren, S. et al. Deep Residual Learning for Image Recognition. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* 2016, p. 770–778. DOI: 10.1109/CVPR.2016.90.

[18] Dodge, S. and Karam, L. A Study and Comparison of Human and Deep Learning Recognition Performance under Visual Distortions. In: *2017 26th International Conference on Computer Communication and Networks (ICCCN).* 2017, p. 1–7. DOI: 10.1109/ICCCN.2017.8038465.

[19] Deng, J., Dong, W., Socher, R. et al. ImageNet: A Large-Scale Hierarchical Image Database. In: *CVPR09.* 2009.

[20] Chollet, F. et al. *Keras* [https://keras.io]. 2015.

[21] Bradski, G. The OpenCV Library. *Dr. Dobb's Journal of Software Tools.* 2000.

[22] BRADSKI, A. *Learning OpenCV, [Computer Vision with OpenCV Library ; software that sees].* 1. ed.th ed. O'Reilly Media, 2008. ISBN 0-596-51613-4. Gary Bradski and Adrian Kaehler.

[23] KINGMA, D. P. and BA, J. Adam: A Method for Stochastic Optimization. In: BENGIO, Y. and LECUN, Y., ed. *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings.* 2015. Available at: http://arxiv.org/abs/1412.6980.

Appendix A

Contents of the included storage media

In this attachment, the contents of the storage media (CD-R) are presented (see Figure A.1 for the schematic).

Not all the files included are listed down below. The `training_data` and `validation_data` folders include several Yoga image files, and the `doc` folder contains the LaTeX source files. For more detailed explanation of the storage media content, please, see the `readme.md` file in the root directory.

- `bin/annotator` – Folder containing the annotation application (both `.exe` and `.py` files) and its documentation files.
- `bin/frame_capture` – Folder containing the frame capturing script and its documentation `README.md` file.
- `bin/main_ml` – Folder containing the main machine learning/model training script and its documentation `README.md` file.
- `bin/renew_dataset` – Folder containing the secondary dataset forming script and its documentation `README.md` file.
- `data/DatasetSamples` – Image samples from the Dataset. Each of the `training_data` and the `validation_data` folders includes Yoga images representing each of the 22 different Yoga poses.
- `data/VideosAndAnnotations` – Example of two source Yoga videos and the corresponding `.json` files containting the annotations.
- `doc/...` – Bachelor's thesis LaTeX source files.
- `Kutalek_Jiri_Detection_of_a_Yoga_Poses_in_Image.pdf` – Bachelor's thesis PDF file.
- `Kutalek_Jiri_poster.pdf` – The attached poster.
- `Kutalek_Jiri_video.mp4` – The attached demonstration video.
- `readme.md` – The main manual file for all of these files in the `doc` directory.

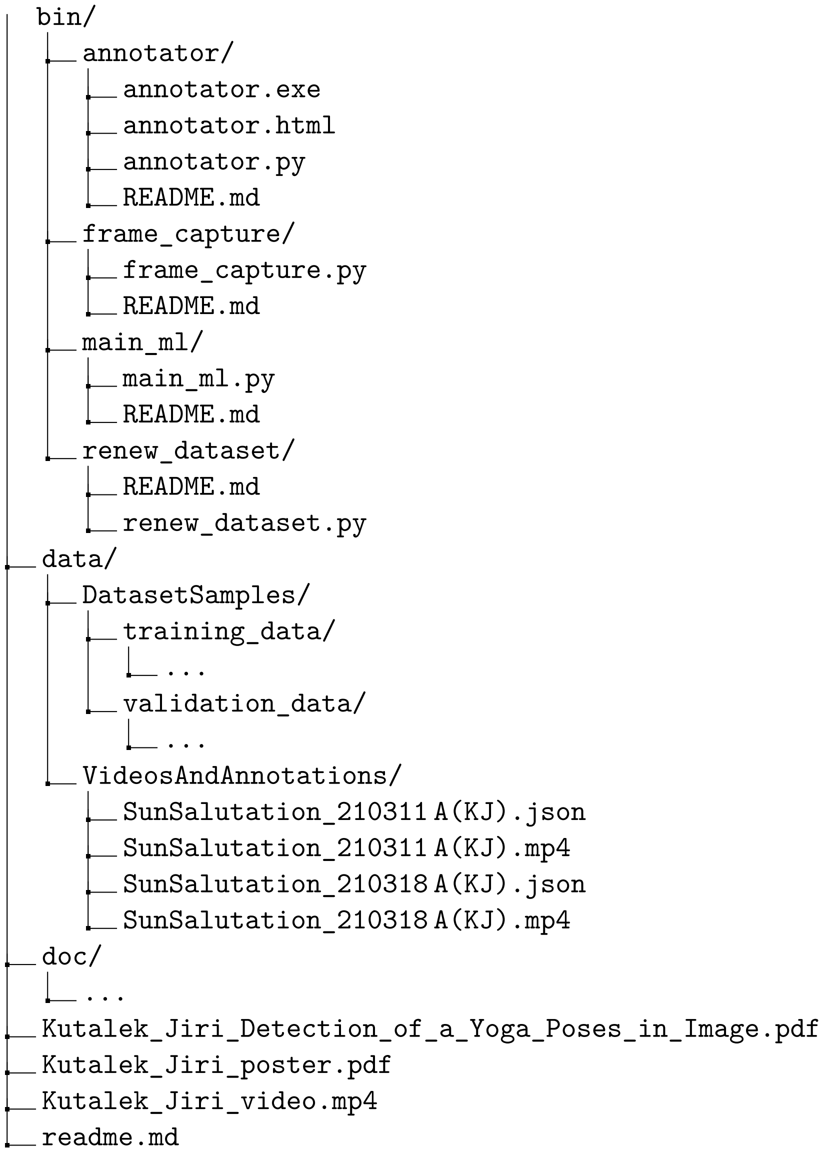

Figure A.1: Schematic of the storage media contents. More detailed information, including the description of how to run the individual scripts and where to find more information about them, is in the `readme.md` file.