BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

KEYBOARD AND KEYS IMAGE RECOGNITION

ROZPOZNÁNÍ KLÁVESNICE A KLÁVES V OBRAZE

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. JAN LORENC**
AUTOR PRÁCE

SUPERVISOR **Ing. JAN PLUSKAL**
VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

145570

Institut:	Department of Information Systems (UIFS)
Student:	**Lorenc Jan, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Information Systems and Databases
Title:	**Keyboard and Keys Image Recognition**
Category:	Computer vision
Academic year:	2022/23

Assignment:

1. Study machine learning methods for image recognition. Select relevant methods for the detection and recognition of keyboards and their keys in an image.
2. Create a suitable dataset containing image data with different types of keyboards, both physical and digital, and respective keyboard characters/symbols.
3. Design an application capable of creating an ML model for keyboard and individual key recognition based on the dataset from point 2.
4. Implement the application and create an ML model according to the proposal from point 3.
5. Test the application and the model using the dataset from point 2. Evaluate the results achieved.

Literature:

- Liu, Zongyi & Ferry, Bruce & Lacasse, Simon. (2019). A Deep Neural Network to Detect Keyboard Regions and Recognize Isolated Characters. 10.1109/ICDARW.2019.40095.

Requirements for the semestral defence:

- points 1, 2, and 3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Pluskal Jan, Ing.**
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	1.11.2022
Submission deadline:	17.5.2023
Approval date:	26.10.2022

Abstract

The goal of this thesis is to create a solution for keyboard keys recognition to automate robotic writing on keyboards. Datasets for keyboard detection in an image, character detection in an image and post-processing correction of the character detection based on various keyboard layouts were created as prerequisites for this work. This research presents several approaches towards keyboard keys detection problem and selects the most suitable one. The chosen strategy is to split the problem into 3 phases which correspond to the prepared datasets. First of all, a separate keyboard detection is run. After that, characters are recognized in the detected keyboard region. These tasks are accomplished using neural networks and Canny edge detection technique. The last phase is the post-processing of the detection results (character correction, autocompletion of undetected characters, special keys distinction etc.). The results of each phase are evaluated. The contribution of the thesis lies in the creation of the datasets for keyboard and keys detection, and novel modular and extensible solution for the recognition process that yields very promising results.

Abstrakt

Cílem práce je vytvoření řešení pro rozpoznání kláves na klávesnici za účelem automatizace robotického psaní na klávesnici. V rámci práce jsou vytvořeny datasety pro detekci klávesnice v obraze, rozpoznání znaků v obraze a dodatečnou korekci detekovaných znaků na základě různých rozložení klávesnic. Práce předkládá různé přístupy k řešení problému rozpoznání znaků na klávesnici a vybírá ten nejvhodnější. Navržený postup je rozdělen do 3 fází, kterým odpovídají připravené datasety. Pomocí neuronových sítí a Cannyho metody detekce hran se nejprve rozpozná klávesnice v obraze a následně se v nalezené klávesnici detekují jednotlivé znaky. V poslední fázi dochází k dodatečnému zpracování výsledků (oprava znaků, doplnění nerozpoznaných znaků, nalezení speciálních kláves apod.). Pro každou část jsou vyhodnoceny výsledky. Přínos práce spočívá ve vytvoření datasetů pro detekci klávesnice a jejích kláves a především modulárního a rozšiřitelného řešení pro detekční proces se slibnými výsledky.

Keywords

machine learning, computer vision, object detection, recognition, neural networks, Canny edge detector, data augmentation, keyboard detection, character recognition

Klíčová slova

strojové učení, počítačové vidění, detekce objektů, rozpoznávání, neuronové sítě, Cannyho detektor hran, augmentace dat, detekce klávesnice, rozpoznání znaků

Reference

LORENC, Jan. *Keyboard and Keys Image Recognition.* Brno, 2023. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Jan Pluskal

Rozšířený abstrakt

Práce se zabývá rozpoznáním klávesnice a jejích kláves v obraze. Námět vzešel z firmy Y Soft vyvýjející robotické řešení k ovládání hardwarových zařízení. To lze využít například k automatizaci testování daných přístrojů. Aktuálním cílem je naučit robota autonomně psát na klávesnici, k čemuž je potřeba dokázat klávesnici s jejím obsahem rozpoznat. Úloha je rozdělena na 3 úkoly. Nejprve je třeba detekovat samostatnou klávesnici v obraze. Dále se naleznou jednotlivé klávesy s cílem rozpoznat především alfanumerické znaky. Na závěr se dodatečně zpracují výsledky na základě podporovaných rozložní klávesnic za účelem oprav chyb, dopočítání nedetekovaných kláves, nalezení klíčových slov apod. Pro každou podúlohu jsou připraveny patřičné datové sady.

Při průzkumu existujících řešení bylo nalezeno pouze jedno a v práci je dále popsáno. Taktéž rozpoznává klávesnici s jejími znaky a posloužilo k inspiraci. Nicméně, nebere v potaz speciální znaky a nedokáže rozpoznat speciální klávesy, pokud na nich není ikona. Navíc, ani kód ani datové sady se nepodařilo veřejně dohledat. Z toho vyplývá, že dle průzkumu této práce neexistuje jiné dostupné řešení daného problému ani data k jeho realizaci.

Datové sady byly vytvořeny celkem tři. První se týká klávesnic a slouží k trénování a validaci neuronové sítě k detekci objektů v obraze, kde jedinou třídou je právě klávesnice. Z odlišných zařízení (telefony, televize, tiskárny, infotainmenty aut...) bylo dohromady nasbíráno 615 klávesnic různých typů (qwerty/abecední/numerické, fyzické/digitální). Tyto byly augmentovány na 20 000 obrázků, v rámci čehož byly generovány na různá pozadí. Těmi byly buď náhodná scéna ze sady COCO17, nebo shodné pozadí s pozadím klávesnice. K augmentaci byly použity metody jako škálování, rozostření, změna světla či průhlednosti nebo přidání různých šumů a moiré efektů. Další datová sada obsahuje 99 různých znaků či ikon speciálních kláves a taktéž slouží k trénování detektoru. Tato sada je realizována v šedotónové barevné paletě. Důvodem pro tuto volbu je vysoký kontrast mezi pozadím klávesnice a samotnými znaky. Převedením do šedotónových odstínů lze minimalizovat potenciální vizuální rušivé faktory, spojené s různými barevnými designy uživatelských rozhraní. Znaky jsou generovány na jednobarevné pozadí náhodného odstínu šedi, což odpovídá pozadí klávesnic. Metody augmentace jsou použity stejné jako u klávesnic a celkem bylo vytvořeno 40 000 obrázků. Poslední datová sada je ryze validační a slouží ke kontrole výstupů algoritmu následného zpracování výstupů detekce.

Pro rozpoznání klávesnice v obraze byla použita zmenšená verze současného state-of-the-art detektoru pro rozpoznání objektů v obraze YOLOv7. Tato zmenšená verze dosáhla srovnatelné přesnosti jako originální, přičemž je však násobně rychlejší a využívá mnohem méně výpočetních prostředků. Velikost vstupních obrázků detektoru je 640x640 pixelů, avšak pro jinou velikost proběhne automaticky škálování či výplň. Výstup detektoru klávesnic, tedy rozpoznaný region klávesnice, je pak použit jako vstup pro detektor znaků. Zde je znovu použit YOLOv7 detektor a byly opět trénovány různé verze. Zmenšená verze tu již nedosahovala stejných výsledků jako originální, nicméně výsledky algoritmu následného zpracování prokázaly, že obě verze jsou funkční a prakticky použitelné. Nicméně, ku příkladu mezerník je velmi často prázdná klávesa a zatímco v obecnosti platí, že znak reprezentuje klávesu, není tomu tak vždy. Z toho důvodu je paralelně provedena i detekce hran pomocí Cannyho metody. Výsledky detekce hran jsou pak použity algoritmem následného zpracování. Cílem tohoto algoritmu je rozpoznání rozložení klávesnice a dle toho korekce chyb. Je schopen doplnit nedetekované znaky, opravit pozice detekovaných, opravit velká a malá písmena těžko rozpoznatelných znaků jako x vs. X, o vs. O atd. Dále, na některých klávesách se může vyskytovat více znaků platících pro jiný mód klávesnice.

Algoritmus se pak snaží určit, o kterou klávesu se jedná. Významnou součástí je také rozpoznání klíčových slov, tedy sloučení korespondujících a sousedících znaků do slova a označení správnou třídou. Na závěr, nejsou-li nalezené vybrané speciální klávesy, dojde k pokusu je uhodnout dle výstupu Cannyho detektoru hran.

Všechny 3 úlohy byly vyhodnoceny samostatně na svých datových sadách. Výborného výsledku dosáhl detektor klávesnic, který korektně rozpoznal každou klávesnici v testovacích datech. Detektor znaků již tak přesný nebyl, nicméně to se ani neočekávalo z několika důvodů. Rozpoznává i velmi těžké speciální znaky jako jsou tečky, čárky středníky apod. Dále existuje již zmíněná problematika podobných velkých a malých písmen (x vs. X). Navzdory tomu však taktéž dosahuje velice kvalitních výsledků a to nejen v originální verzi modelu, ale i ve zmenšené. Oba modely mají přesnost alespoň 95 %, nicméně úplnost menšího modelu značně zaostává. Rozdíl mezi zmenšenou a originální verzí modelu je však smazán použitím algoritmu následného zpracování. Jelikož je schopen dopočítat vynechané znaky, nižší úplnost menšího modelu nevadí, dokud rozpozná alespoň něco. Naopak, tím, že zmenšený model nabízí při stejném prahu jistoty méně znaků, produkuje i méně chybných pozitivních detekcí a tím i méně mate daný algoritmus. Proto dokonce po aplikaci algoritmu následného zpracování produkuje na konečné validační datové sadě i lehce lepší výsledky, než-li originální model.

Výstupem práce je pak modulární funkční řešení problému rozpoznání klávesnice a kláves v obraze. Libovolný model je možné přetrénovat či vyměnit za jiný. Podobně lze snadno přidat či modifikovat následné zpracování výsledků. Řešení bude integrováno do produkční verze systému Y Soft AIVA. Navíc nabízí datové sady klávesnic a znaků použitelné i pro jiné modely či úkoly. Práce se také účastnila konference Excel@FIT 2023, kde byla oceněna odbornou veřejností cenou Jiřího Kunovského.

Keyboard and Keys Image Recognition

Declaration

I hereby declare that this thesis was prepared as an original work by the author under the supervision of Ing. Jan Pluskal. The supplementary information was provided by the members of Y Soft AIVA team. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Jan Lorenc
May 9, 2023

Acknowledgements

I would like to give my sincere thanks to the supervisor of the thesis Ing. Jan Pluskal for his expert guidance. Furthermore, I thank the Y Soft Corporation a.s. and especially the AIVA team for making this thesis possible and for useful advice.

Contents

Chapter 1

Introduction

Object detection is one of the most commonly used techniques in the image processing field. Its aim is to locate objects such as people, cars, animals etc. in an image or a video, construct bounding boxes for such objects and correctly label them. As of today, the application of object detection is used in various areas. For self-driving cars, it is of utmost importance to recognize obstacles. To gather information about customers, people counting systems are used. In agriculture, animals can be monitored thanks to object detection. Countless other domains including anomaly detection, security or medicine benefit from it. In this work, object detection is employed for automation, namely for automatic keyboard typing.

1.1 Current state of keyboard and keys detection

At the time of writing the thesis, only one solution for keyboard and keys detection seems to have been published. Researchers from Amazon have a robotic framework for testing mobile devices and they need to recognize keyboards and their keys so that their robots can automatically type [1]. This is essentially the same problem as mine and they solve it in another paper [2]. However, their solution focuses primarily on mobile Android and iOS devices while there are other devices such as printers, car infotainments, e-book readers etc. which might not even have a digital keyboard but a physical one. Moreover, neither the code nor the dataset can be easily found, if they are even publicly available, so it doesn't look to be a reusable solution.

Should we split the problems into just keyboard detection and keys detection, more options can be found. Keyboards can be recognized by standard object detection methods and even cameras on most modern smartphones can detect keyboards. There even exist some libraries [3, 4] which can detect a keyboard on the screen, but rather than image recognition they usually just use the UI API of the device operating system to get information about the layout of the UI elements.

The problem of keys detection can be viewed as a single-character recognition. There exist a huge amount of research and available methods for OCR technologies which might solve this issue. Nevertheless, after testing several popular OCRs both open-source[1] and commercial[2], the OCR were insufficient for the task. The main problem is that they try to join the characters to words as the major focus of most OCRs is to detect text in scenes

[1] Tesseract, EAST, CRAFT
[2] Google, Microsoft

and not single characters. Even the CRAFT OCR [5] which doesn't search for words but for letters doesn't produce satisfying results. The reason for this is that the letters are further merged into words based on an affinity score which is nearly impossible to find so it is general. To the same conclusion, that the OCR is not the way to solve this problem, came even the above-mentioned Amazon researchers in their paper [2].

The final issue with keyboard detection and its keys is a non-existent dataset. There cannot be found any set of keyboards to be used for keyboard detection, let alone annotated. The Amazon researchers struggled with this as well and were forced to download various keyboards from internet search engines and manually label them [2]. However, as already mentioned, this dataset doesn't seem to be publicly available either.

To summarize, a general public solution for keyboard and keys detection task is nowhere to be found. Existing available methods are not suitable either. In addition, according to this work's research there is no dataset that can be used for solving this problem and comparing against it.

1.2 Motivation

While the creation of an open-source solution for keyboards and keys detection along with a dataset is an incentive of its own, it is not the fundamental reason for this work. Apart from the Amazon robotic testing system, there exists another. Y Soft Corporation has developed a robotic system AIVA for testing and remote-controlling devices. The idea for this work originated from the need for learning AIVA to write on a keyboard by itself to further automate the testing processes.

The current process of writing on a keyboard in the AIVA system is as follows. There are images of known target device screens in the system. Screens with the possibility of a keyboard needs to have a second variant with the keyboard displayed. Furthermore, each screen must have defined actionable elements such as buttons or input fields. In the case of a keyboard screen, every keyboard key has its own button element defined. In a test scenario, all of this is specified and a test flow of filling a login „uSer“ can look like this:

1. Go to screen “Login screen”
2. Go to screen “Login screen with keyboard”
3. Tap element “u”
4. Tap element “shift”
5. Tap element “s”
6. Tap element “shift”
7. Tap element “e”
8. Tap element “r”

The system naturally provides the user with a set of flow actions which simplify the text writing so that one doesn't need to specify each letter individually. Nevertheless, this set of actions is what it is translated to on the backend eventually. Moreover, the usage of special keys like “shift” often forces the explicit writing of letters anyway. Consequently, automatic recognition of keyboard keys would allow the creation of a single flow action capable of distinguishing keyboard modes, switching them using special keys and hence writing any text without the need of specifying any key elements. Furthermore, duplicate screen definitions just with a keyboard on them would no longer be required owing to the keyboard detection.

1.3 Expectations

The AIVA system already has several features which can help with the task specification in more depth and definition of what exactly is expected of the thesis. Firstly, full-HD cameras are used in all AIVA units so high-quality images can be presumed. Secondly, AIVA units always operate in a lab or an office environment. Therefore, image noise caused by natural elements such as rain doesn't need to be taken into account. Lastly, a camera calibration system is in place which means that it is possible to obtain just the device screen (zoomed without any background) still in full-HD quality and without any rotation or distortion. With these system properties being defined, solution requirements can be further described.

One of the key requirements is the very ability to detect a keyboard in an image. This task is needed quite often both for validation of test steps and navigation on and between the screens. To solve this problem, research into object detection algorithms has been done and further described in chapter 2. Then, a dataset for training a model must have been created about which refers chapter 3.1. Final model architecture is illustrated in chapter 4.1.

The expected thesis outcome is the means for typing automation though. For that reason another detection model is required, now for the keyboard keys with the objective to recognize alphanumeric characters and basic special keys (space, enter, shift...). Optional inclusion of special characters would be a benefit, however, problematic keys can still be defined explicitly the old way. More about the dataset creation can be found in chapters 3.2 and 3.3. Solution design is described in chapter 4.

1.4 Contributions

There are two areas which benefit from this work, as the previous sections might have already hinted. Regarding the public, open-source neural network models for both keyboard region detection and keyboard keys recognition in an image are provided. Along with that, an algorithm for keys correction based on keyboard layout is devised. Furthermore, datasets for all three tasks are created and available. Being split into several parts, the solution can be used for individual problems as well as for the whole keyboard automation issue.

Concerning Y Soft company, the thesis provides a new set of features for automatic keyboard typing. Compared to the old approach, it removes the need for keys definition completely. Moreover, a duplicate screen definition for the same screen, just with the keyboard opened, is no longer required. This also leads to simpler navigation between screens. The automatic key detection can be used to switch between the keyboard modes and find a target character elsewhere. Thanks to all of that, the test case scenario can be significantly reduced. At the end of the day, it makes AIVA one step smarter and the job of a quality assurance engineer easier.

Chapter 2

Image recognition algorithms

This chapter presents several techniques for object detection in an image. In the first section 2.1 are introduced two representatives of classical computer vision algorithms. The second section 2.2 focuses on the neural network approach. Not all of the described algorithms are used in the final solution, however, all of them are at least options and provide context for the selected ones.

2.1 Classical computer vision techniques

While using classical computer vision algorithms might seem irrelevant in the era of neural networks, it is not entirely so. They do not require any training and the computation is very fast. Moreover, keyboards are usually quite contrastive (dark text on a light background and vice versa) which presents opportunities for detectors based on edges or colors.

2.1.1 Canny edge detector

This detector was developed by John Francis Canny in 1986. Since the original paper [6], the algorithm has undergone slight modifications. Currently, one of the most used implementations is the one from OpenCV library [7], which consists of several steps:

1. *Noise reduction*
   Edge detection is very susceptible to noise. For that reason, the image is blurred by a Gaussian filter. As a consequence, the true edges will stand out.
2. *Gradient calculation*
   The Sobel filter is applied to the image to obtain horizontal and vertical derivatives 2.1.

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} * image, \quad G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * image \tag{2.1}$$

   The magnitude 2.2 and angle 2.3 of the directional gradients can be computed as follows:

$$|G| = \sqrt{G_x^2 + G_y^2} \tag{2.2}$$

$$Angle(G) = arctan(G_x^2/G_y^2) \tag{2.3}$$

3. *Non-maximum suppression*
   This stage tests each pixel for being a local extreme in the direction of the gradient. If not, it is suppressed.

4. *Hysteresis thresholding*
   The detector accepts 2 parameters, high and low thresholds. Edges with an intensity gradient higher than the upper threshold are considered "sure edges" and are accepted by the algorithm. On the other hand, edges with gradients lower than the lower threshold are suppressed. Edges with gradients in between are kept only if they are connected to a "sure-edge".

The result of the algorithm is a binary image. Bounding boxes can be constructed by computing contours. The following images 2.1, 2.2, 2.3 demonstrate the keyboard keys detection using the canny detector.

Figure 2.1: Canny detector creates a binary image with discovered edges.

Figure 2.2: Contours and their bounding boxes can be computed from the binary image.

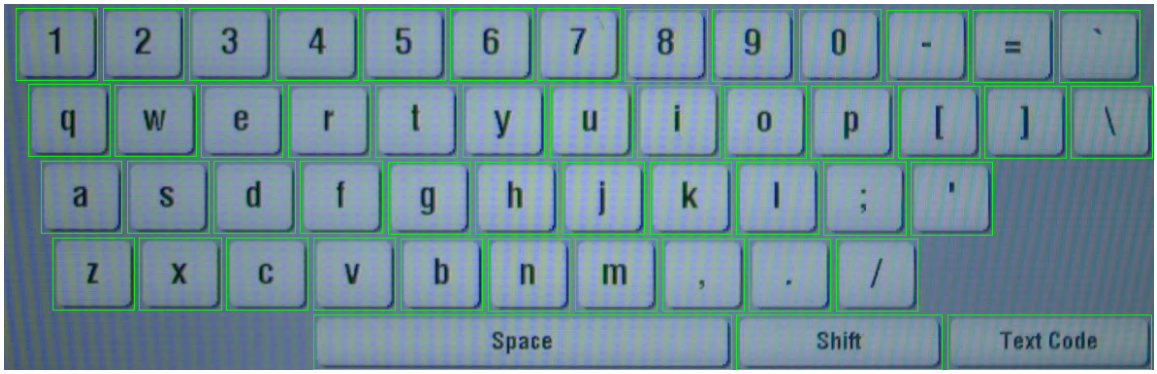


Figure 2.3: Canny algorithm is capable of detecting key regions on a keyboard.

2.1.2 Thresholding

Thresholding is a method which converts a polychromatic image to a binary one (usually black-and-white). To obtain pixel intensities, the image is typically converted to grayscale representation. Then a threshold is chosen and pixels with intensities above this threshold are set to one value (white) and the ones below to another (black). The computation is described by formula 2.4, where $I(x, y)$ is pixel intensity on position $[x, y]$, T is a threshold and $G(x, y)$ is the resulting pixel value.

$$G(x,y) = \begin{cases} 255 & \text{if } I(x,y) \geq \mathrm{T} \\ 0 & \text{else} \end{cases} \tag{2.4}$$

However, this method is quite trivial and doesn't work well with images with different light conditions [7]. In such cases, a different approach such as adaptive thresholding should be used which computes the threshold from the neighborhood of the given pixel. Figure 2.4 shows the different results from global and local thresholding.

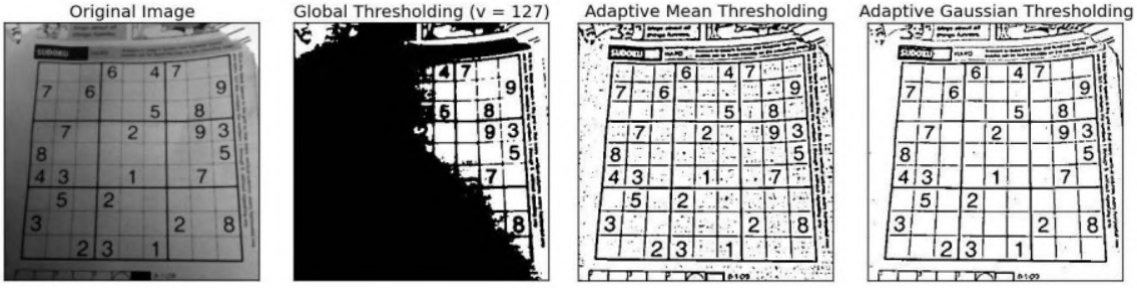


Figure 2.4: Difference between global and adaptive local thresholding (adapted from [7])

Another popular thresholding method is Otsu's binarization. It is effective and does not require a threshold specification. This method computes probabilities of pixel intensity values and constructs a histogram. The histogram usually contains 2 major peaks (classes), which represent background and foreground [7]. By iterating over all possible threshold values within the intensity range, the algorithm selects a threshold with minimum intra-class variance [8]. Following figure 2.5 depicts a result of Otsu's thresholding.

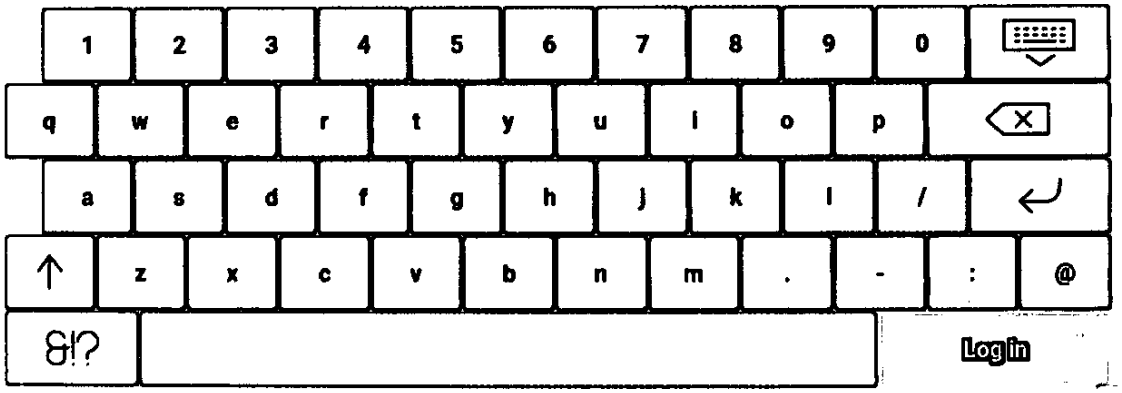


Figure 2.5: Keyboard keys can be detected using Otsu's binarization and bounding boxes acquired by contours computation same as in Canny detector approach 2.1.1.

2.2 Neural networks

Since the success of AlexNet in the ILSVRC competition in 2012, convolutional neural networks have become the backbone of computer vision. There exist many sub-domains of computer vision such as classification, segmentation or image restoration and new ones emerge every day. This section takes a closer look at object detection sub-domain. The main problem in object detection is the localization of the target objects. The objects can be anywhere and by using classification, a sliding window would have to be used which would classify the object in it or not. The issue with this approach is the variability in the size of the objects and the number of classifications necessary. Therefore, new algorithms had to be devised and the next sections describe the most iconic ones.

2.2.1 R-CNN

To solve the issue of a large amount of sliding window positions for classification, R-CNN selects only 2000 region proposals using Selective Search algorithm [9] which can be summed up to 3 phases [10]:

1. Generate candidate regions using segmentation.
2. Merge similar regions using a greedy algorithm.
3. Create final region proposals.

Once the regions are obtained, each of them goes into a ConvNet [11] which extracts a 4096-long feature vector [12]. This vector is then classified by the SVM [13] algorithm. Apart from the actual detection, 4 offset values are predicted to improve the bounding box precision (e.g. part of the object may be cut, so to correct it) [14]. Figure 2.6 demonstrates the whole detection process.

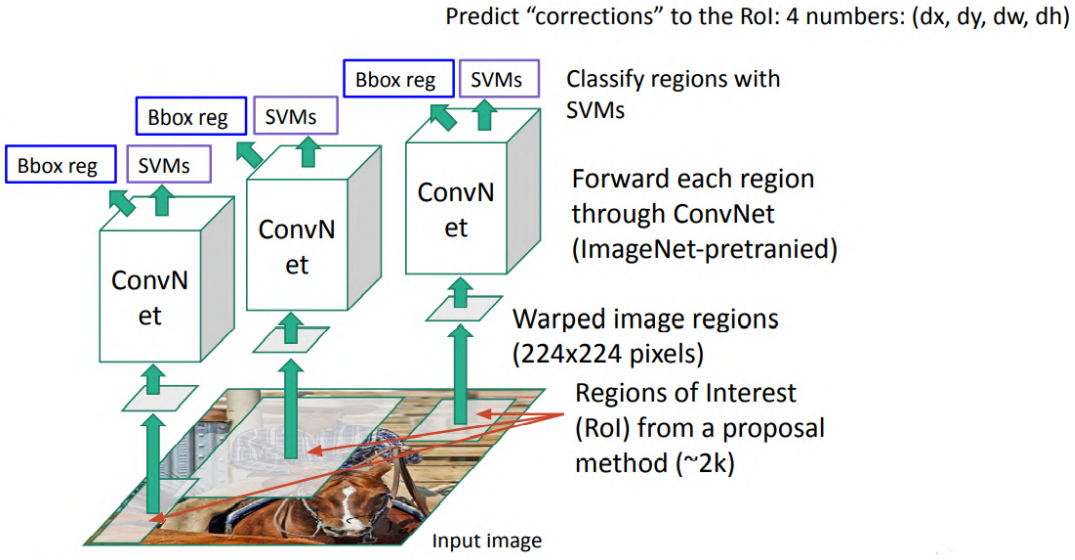


Figure 2.6: Object detection process used by R-CNN (adapted from [14])

R-CNN achieved remarkable accuracy for its time and significantly contributed to the development of specialized architectures for object detection. For instance, on PASCAL VOC 2012 it improved the accuracy by 30 % in comparison to the best previous result [12].

2.2.2 Fast R-CNN

In spite of great object detection accuracy, R-CNN is very slow and computationally expensive. Therefore, the author of the R-CNN tackles the drawbacks in a new paper [15]. There are several issues he wanted to solve:

1. *Multi-stage pipeline training* - ConvNet, SVM and bounding box regressors are trained separately.
2. *Expensive training* - It takes time to classify 2000 region proposals per image and space to store features for each of them for both SVM and bounding box regressors.
3. *Slow detection* - Features must be extracted for each region proposal, which means running ConvNet forward pass, and there is no computation sharing.

To improve the detection speed, features are no longer extracted from the regions, but instead, the whole image is processed by the ConvNet. By doing that, the feature extraction is run only once and the whole image feature map is obtained. An ROI pooling layer is then used to create a feature vector for each region proposal. ROI is a window of rectangular shape which is divided into sub-windows and standard max-pooling is performed on each of them [15]. It is based on the spatial pyramid pooling layer in SPPnets [16] and the same size output is ensured for the subsequent fully-connected layers as is depicted in figure 2.7. After that, the softmax loss function is used to predict the object class and a bounding box regressor the offset. Due to 2 output layers and single-stage training goal, a multi-task loss is introduced which combines both layers' losses and hence classification and bounding box regression can be trained together [15]. Full Fast R-CNN architecture is shown in figure 2.8.

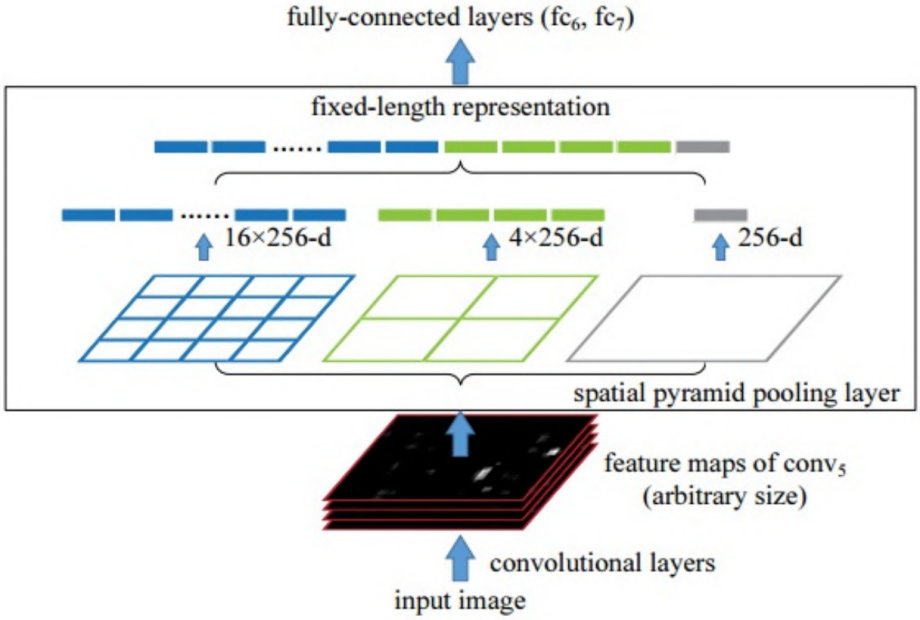


Figure 2.7: ROI pooling layer in Fast R-CNN (taken from [18])

Fast R-CNN managed to fix all of the pin-pointed drawbacks. The training became single-stage which can update all layers at once and there is no need to store features on disk anymore [15]. In addition, detection is much faster and even more accurate [15]. Compared to the R-CNN, training time got reduced almost 10 times and test time 23 times including the region proposals and nearly 147 times excluding them [14].

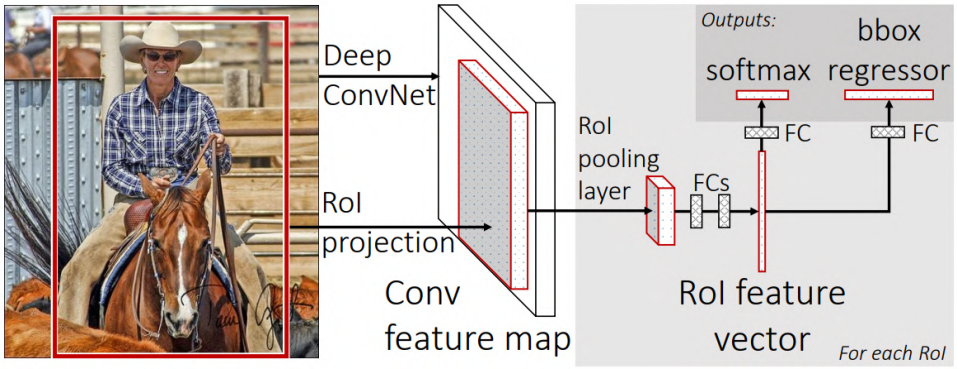


Figure 2.8: In Fast R-CNN, a feature map is extracted from the input image and converted to a feature vector by the ROI layer. The softmax then makes the classification and the bounding box regressor finds the offset (taken from [15]).

2.2.3 Faster R-CNN

R-CNN and Fast R-CNN share a common problem which is the use of Selective Search algorithm for region proposals. Selective Search on its own is rather popular and effective, but in comparison with neural networks, it is slower by a margin as it takes around 2 seconds per image to find the regions [17]. It doesn't matter much to R-CNN since it is quite slow and the performance hit is not so noticeable. However, Fast R-CNN without the region proposals achieves almost real-time rates and the 2-second hit increases computation 7 times [14]. Faster R-CNN aims at replacing Selective Search with Region Proposal Network (RPN) as shown in figure 2.9.

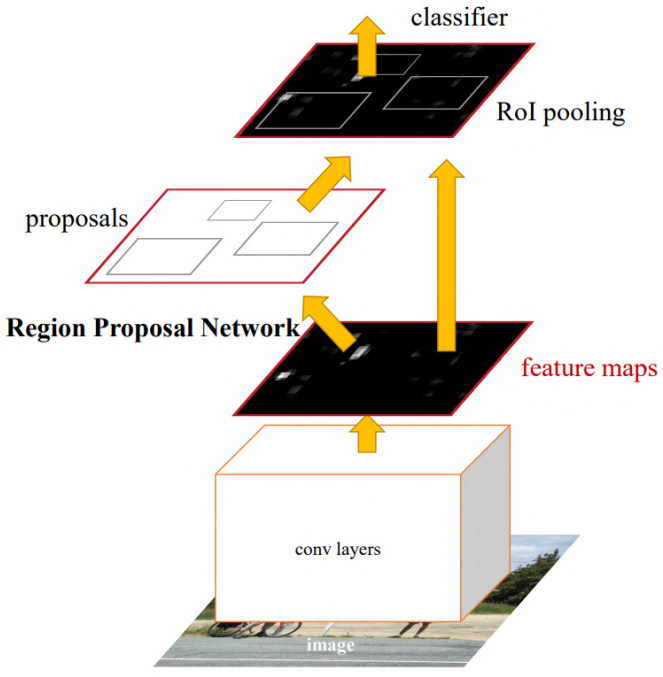


Figure 2.9: Faster R-CNN is essentially Fast R-CNN with a neural network in place of Selective Search (taken from [17]).

To understand better how RPN works and how it is included in the model architecture, the original paper [17] describes it quite clearly. Still, I will provide a short summary. As the idea is to train the whole model together, RPN shares the convolutional layers with the Fast R-CNN and works on the output of the last convolutional layer, the feature map. It passes a rectangular sliding window through the map and generates several region proposals for each window. The proposals are then parameterized relative to reference boxes called anchors. The anchors lie at the center of the sliding window and have different scales, so the detection is size-indifferent. A feature vector is extracted for each proposal and a sibling two-layer network predicts the region coordinates and an objectness score defining, how sure it is that an object is in the region. This is visualised in figure 2.10.

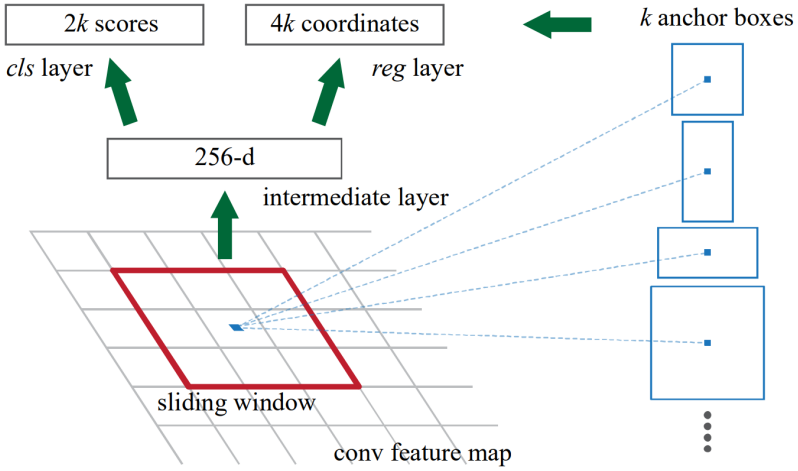


Figure 2.10: Region extraction in Region Proposal Network (taken from [17])

During the training, the intersection-over-union of anchors and ground-truth boxes is computed. If an anchor is eighter the highest scoring or scores over 0.7 with any ground-truth box, it is assigned a positive label for being an object. Having removed the Selection Search algorithm, it takes only 0.2 seconds to process an image which makes it a real-time object detector and 11 times faster than the Fast R-CNN [14].

2.2.4 YOLO

YOLO (You only look once) brought a novel approach to the object detection field. As the authors of this algorithm [19] claim, the detection systems at that time reused classifiers for the evaluation of certain windows in an image. In spite of having methods for selecting such windows, the R-CNN family is no different. What is more, the whole process is a pipeline of several actions and all the R-CNN algorithms need to look at the image twice, once for region proposal generation, and once for object detection of the proposals. YOLO, on the other hand, needs only one look and is capable of making the classification and bounding box regression simultaneously. Instead of a single region, it sees the whole image which allows it to encode context information and as a result, it makes significantly fewer background errors than R-CNN based algorithms. Owing to all of this, YOLO is extremely fast. Nevertheless, it lacked in accuracy in comparison with the state-of-the-art approaches at the time [19].

The process of finding the bounding boxes is described by the paper [19] as follows. The image is divided into a S x S grid which portrays figure 2.11 and a grid cell, which is

a center of an object, is responsible for its detection. Each grid cell predicts B bounding boxes and their confidence scores, which reflects the intersection-over-uniou (IoU) with the ground-truth. To predict a bounding box, the neural network uses features from the entire image. However, excess bounding boxes can be produced by neighboring grid cells for the same object. This is solved by non-maximal suppression. Apart from the object location, the grid cell also predicts the object class probabilities.

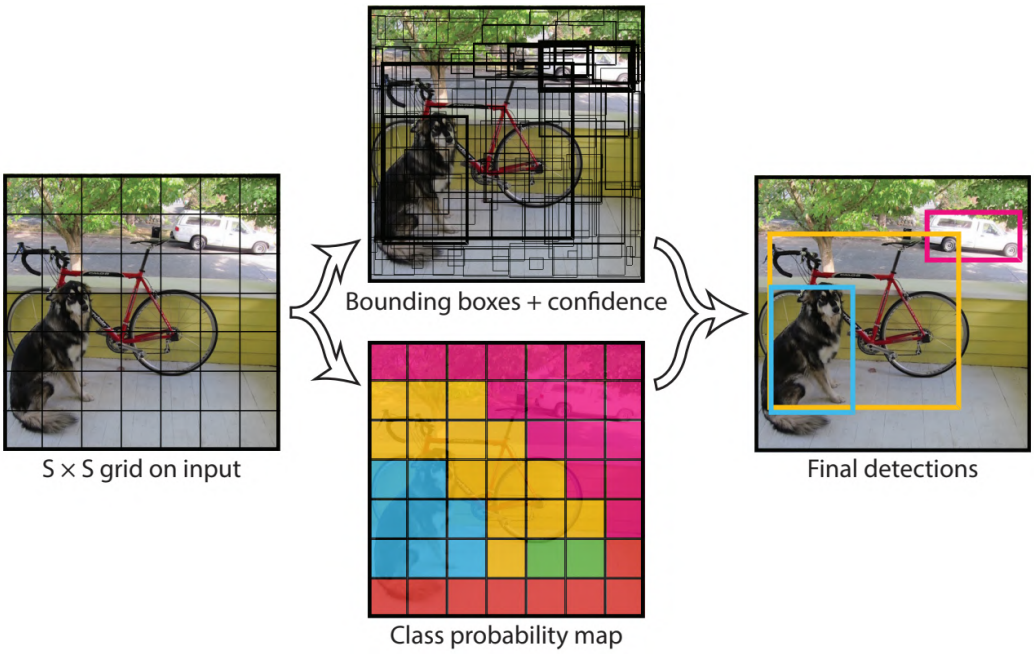


Figure 2.11: YOLO uses a grid to predict bounding boxes and object classes (taken from [19]).

The architecture of YOLO is based upon the GoogLeNet [20] with the modification of using 1x1 reduction layers followed by 3x3 convolutional layers instead of GoogLeNet's inception modules [19]. Consequently, the network has 24 convolutional layers followed by 2 fully connected layers as figure 2.12 shows. The features are extracted by the initial convolutional layers and the fully connected layers then produce the bounding boxes and class predictions [19].

During the training, leaky ReLU activations are used in the hidden layers and normal ReLU in the final layer [19]. For optimization, sum-squared error is utilized for its simplicity, although it weights classification and localization errors equally, which is a downside [19]. The loss function is multi-part and consists of 5 terms. These terms are for the object's center coordinates, bounding box dimensions, object class, a class in case of the object's absence and probabilities of finding the same object respectively [21].

However, not even the novel approach of YOLO is without limitations and the authors are well aware of them and mention them in the paper [19] as well. As each grid cell predicts only a few bounding boxes and a single class, it imposes a spatial constraint which causes difficulties in prediction of small objects like flocks of birds. Furthermore, it struggles with generalization of unusual aspect ratios. It has problems with localization errors. Still, it worked extremely well among other real-time object detectors and new versions fixing those issues have been being developed due to its popularity ever since.

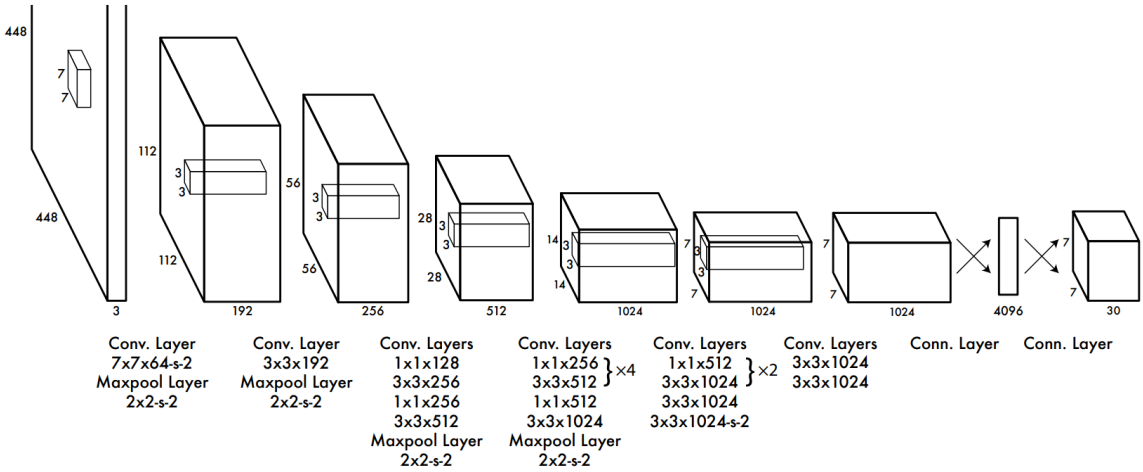


Figure 2.12: Architecture of YOLO network (taken from [19])

YOLOv2

YOLOv2 [22] brought several improvements. These are very well summarized by article [21] from which the most important ones are listed:

- Batch normalization is used instead of dropout layers.
- Resolution detail is increased by the removal of a max-pool layer, which leads to better accuracy.
- Location is predicted with respect to a grid cell instead of an anchor box. This means better stabilization during the training.
- Feature map is more fine-grained (13x13).
- Fully-connected layers are replaced with convolutional ones, which allows for multi-scale training.
- VGG-16 backbone is replaced by Darknet-19.
- Hierarchical classification is introduced. This allows for assigning several classes (e.g. dog and their breed) to objects whereas before they were mutually exclusive.

As a result, YOLOv2 improved especially in accuracy, recognition of smaller objects, but also in speed. It became state-of-the-art in both speed and accuracy without any obvious limitations.

YOLOv3

YOLOv3 focuses mainly on improving speed and accuracy, rather than fixing some known issues as the name of paper [23] "YOLOv3: An Incremental Improvement" also suggests. This paper introduces the following changes. The objectness score is calculated using logistic regression and the sigmoid function is used to predict the bounding box center. In class prediction, a switch to cross-entropy from the sum-squared error was made. Moreover, a new deeper backbone Darknet-53 is used.

YOLOv4

This version of YOLO is very important because the creator of YOLO, Joseph Redmon, retired from computer vision and new researchers continued with YOLO advancements. They came to the conclusion, that a general object detector consists of several parts, which are input, backbone, neck and head [24]. They decided to map YOLO into this decomposition as figure 2.13 depicts.

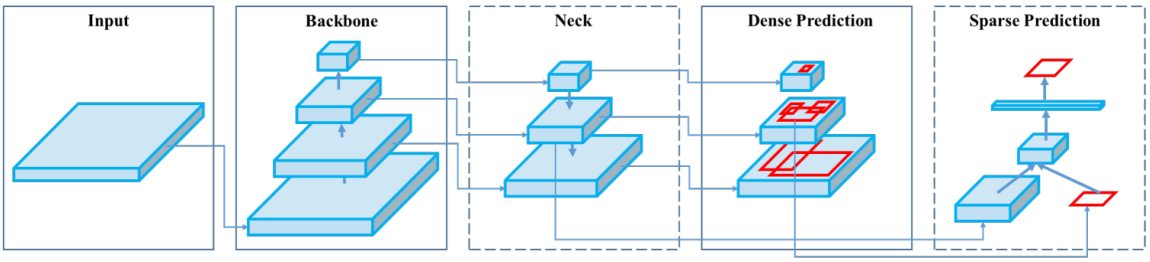


Figure 2.13: In YOLOv4, input is the input image. Backbone is the feature extraction network which was updated from Darknet53 in YOLOv3 to CSPDarknet53. The neck is supposed to be the region selection part, the spatial pyramid pooling, which underwent subtle modification in YOLOv4 to improve accuracy. Lastly, the head takes care of the class and location prediction which in this figure are the dense and sparse prediction blocks (adapted from [24]).

The new authors also introduce bag of freebies and bag of specials. Bag of freebies are methods that improve performance without increasing inference time in production [24]. They can be viewed as some plugin modules which one can try to improve a model. There is a huge amount of them and they are usually normalization or regularization methods, data augmentation, data imbalance etc. From those that YOLOv4 uses can be named e.g. Mosaic data augmentation, DropBlock regularization, class label smoothing and many othes [24]. Concerning the bag of specials, those are post-processing methods which do have a small impact on inference time but bring a significant accuracy increase. Out of these, mish activations, Cross-Stage Partial (CSP) Connection or SPP blocks for instance are applied in YOLOv4 [24].

YOLOv5

YOLOv5 was developed by different authors than YOLOv4 and was released just a month after the previous version. Research on both of them started relatively at the same time and was conducted simultaneously. It was named YOLOv5 simply to avoid a collision [25]. As a result, due to the parallel work and application of the same state-of-the art innovations in the field, the architectures of both do not differ much [25]. YOLOv4 paper even acknowledges Glenn Jocher, the author of YOLOv5, for the Mosaic data augmentation [24]. Consequently, both models perform very similarly, however, YOLOv5 appears to be a bit faster and YOLOv4 slightly more accurate when more custom configuration is applied [26]. Owing to the naming, timing, lack of outstanding improvements in comparison with the previous version and most importantly no research paper, YOLOv5 was a source of quite a controversy in the computer vision community. Nevertheless, it got accepted eventually.

YOLOv6

YOLOv6 further advances both speed and accuracy of YOLO algorithm and beats all other real-time detectors at the time [27]. It further improves the backbone which is now named EfficientRep consisting of RepBlocks [27]. These RepBlocks also replace the CSP blocks in the neck [27]. Main architectural change brings the head, though. Instead of 1 head, 3 heads are used as figures 2.14 and 2.15 show.

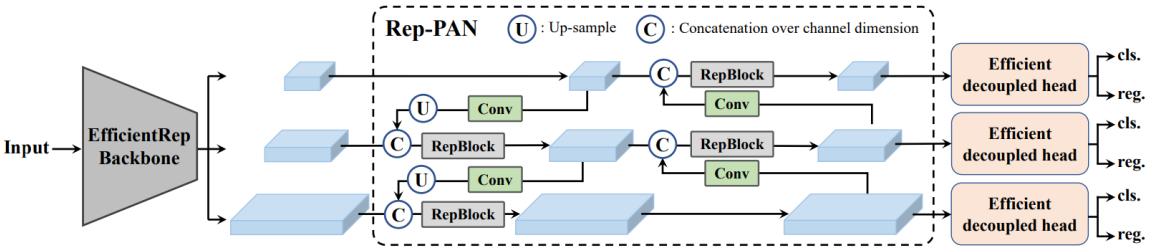


Figure 2.14: In YOLOv6, neck outputs go to 3 decoupled heads. In the previous versions, localization and classification heads were coupled, which means that they shared the same features and more computation was needed [27] (taken from [27]).

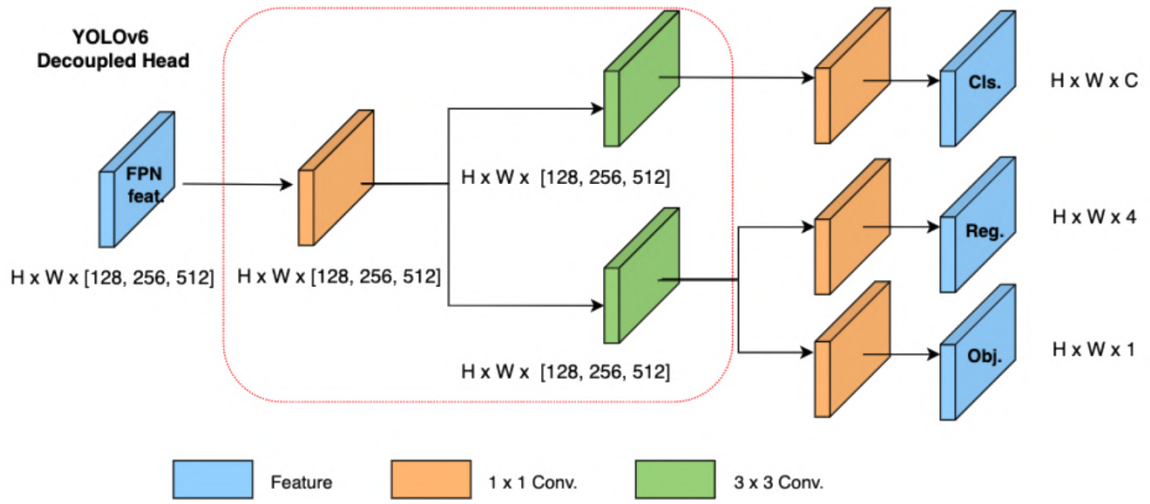


Figure 2.15: In a head, 3 loss functions are computed. Cross-entropy for classification and L1 for regression were left the same from the previous versions. A new object loss is computed, which aims at reducing the score of low-quality bounding boxes but unfortunately does not have many effects [27] (taken from [28]).

YOLOv7

Similarly to YOLOv4 and YOLOv5, the release of YOLOv6 and YOLOv7 was very close. There are still debates about which one is better but more benchmarks found on the internet favour the latter. Moreover, the YOLOv7 paper claims better results than YOLOv6 on the COCO dataset [29]. According to some [30], it is even expected of YOLOv7 to become the new industry standard. It might also be worth mentioning that YOLOv7 come from the same authors as YOLOv4. The most important contributions of YOLOv7 are the following: [29]

- *Extended efficient layer aggregation* – Efficiency of the convolutional layers is the key to the inference speed. Hence, building heavily on research on this topic, using group convolution to increase feature cardinality and combining features of different groups leads to an improved feature learning process.
- *Model scaling for concatenation-based models* – While other models usually scale the width or depth of the network, YOLOv7 scales them together which keeps the architecture optimal for different sizes.
- *Planned re-parameterized convolution* – Re-parameterization means averaging weights to create a more robust model. Which network modules to re-parameterize is determined by gradient flow propagation paths.
- *Auxiliary heads* – Network heads making the prediction are quite far. Other heads placed in the middle of the network help with the training.

Others

Apart from the main versions of YOLO, other YOLO mini-series such as YOLOX, YOLOR or PP-YOLO have emerged. They (and their newer versions) usually build upon a current YOLO version and improve it in certain ways. YOLOX for example removes the anchor boxes and YOLOR combines explicit and implicit knowledge to perform several tasks. However, covering all these YOLO offspring is out of scope of this thesis.

2.2.5 Single Shot Detector

Similarly to YOLO, Single Shot Detector (SSD) needs only one look at an image to process it. At the time, it outperformed both YOLOv1 and R-CNN-based algorithms in speed and accuracy alike [31]. Same as the original YOLO, it uses the VGG-16 net as a backbone on which it builds additional convolutional layers as depicted in figure 2.16.

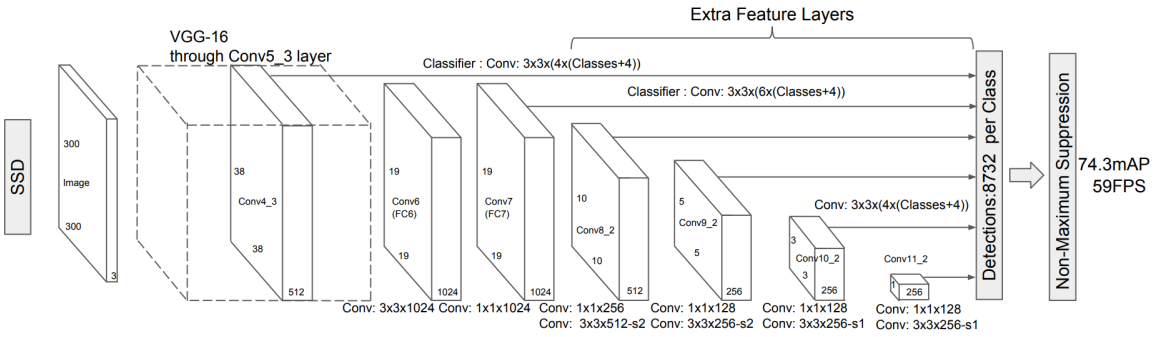
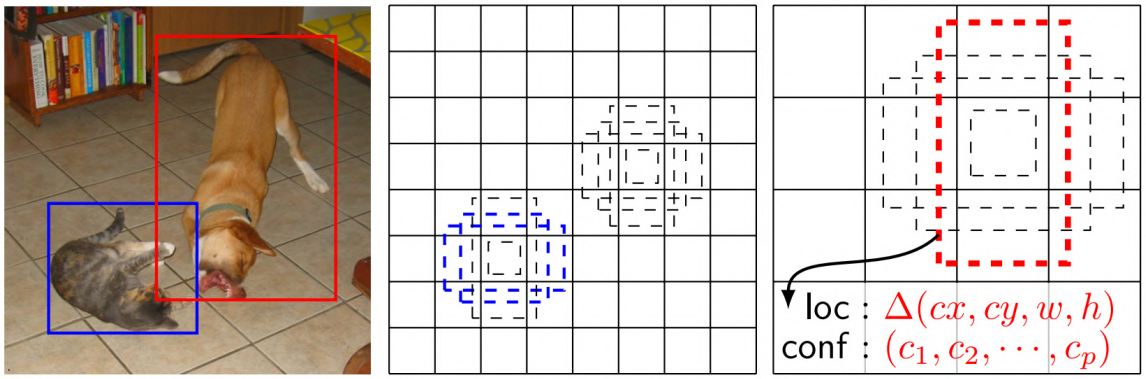


Figure 2.16: SSD architecture consists of VGG-16 backbone followed by additional convolutional layers. The classifiers in extra feature layers use 3x3 convolutions for k (4 or 6) bounding boxes for N classes + 4 offsets. Non-maximum suppression is used at the end to handle intersecting bounding boxes (adapted from [31]).

SSD can be also called Single Shot MultiBox Detector because it uses multiple boxes for localization. The original paper [31] describes the bounding box selection as follows. The added convolutional layers scale down so that multi-scale prediction can be done. Each of them contributes to the result with its set of detections. A certain feature map has size m x n which is a number of locations (feature map cells). A default set of bounding boxes is associated with each location as shown in figure 2.17.



(a) Image with GT boxes (b) 8×8 feature map (c) 4×4 feature map

Figure 2.17: Each feature map cell has k (usually 4 or 6) default bounding boxes. The idea is that a person, for instance, would need a vertical box whereas a car would need a horizontal one (taken from [31]).

Then we can easily get to the 8732 detections from figure 2.16 like this:

- Conv4_3: 38 x 38 x 4 = 5776
- Conv7: 19 x 19 x 6 = 2166
- Conv8_2: 10 x 10 x 6 = 600
- Conv9_2: 5 x 5 x 6 = 150
- Conv10_2: 3 x 3 x 4 = 36
- Conv11_2: 1 x 1 x 4 = 4

For an image of size 300x300 is done $5776+2166+600+150+36+4 = 8732$ detections. In comparison with YOLOv1 which makes only 98 detections for a 448x448 image, it is an extreme improvement and the main source of the accuracy boost [31].

Concerning the loss function, while YOLOv1 used 5 terms based on the sum of squares, SSD uses only two-term loss 2.5 [31]:

$$L(x, c, l, g) = \frac{1}{N}(L_{conf}(x, c) + \alpha L_{loc}(x, l, g)) \tag{2.5}$$

L_{conf} is the softmax loss function for class prediction which takes $x \in \{1, 0\}$ as an indicator if the box matches the ground-truth and c as the class confidence. L_{loc} is the Smooth L1 localization loss and l, g the predicted and ground-truth boxes respectively. N denotes the number of matched bounding boxes.

Then there is the question of how the default bounding boxes are determined. Suppose there are m feature layers, the scale for each layer's bounding boxes is determined by the following equation 2.6 [31]:

$$s_k = s_{min} + \frac{s_{max} - s_{min}}{m - 1}(k - 1), \qquad k \in \langle 1, m \rangle \tag{2.6}$$

In this equation $s_{min} = 0.2$ and $s_{max} = 0.9$, which makes the scale of the lowest layer 0.2, the highest layer 0.9 and other layers regularly spaced. Default boxes' aspect ratios are then $a_r \in \{1, 2, 3, \frac{1}{2}, \frac{1}{3}\}$ (excluding 3 and $\frac{1}{3}$ when only 4 boxes are used) and a box size is computed as $width_k^a = s_k\sqrt{a_r}$ and $height_k^a = \frac{s_k}{\sqrt{a_r}}$ [31].

The large amount of generated bounding boxes comes with a price, though. Most of the boxes are naturally negatives which creates an imbalance between positive and negative training samples [31]. To solve this issue, the hard negative mining technique is used. To achieve more stable training and faster optimization, only the negatives with the highest confidence score are picked. In addition to this, to improve the training and to make the model more robust, data augmentation is heavily employed.

In spite of achieving better results than any other object detector at the time, no further advancements or newer versions of SSD seem to be around. While it undoubtedly influenced many object detection architectures, SSD itself got soon dethroned as the state-of-the-art by newer versions of YOLO. Nevertheless, it can still be seen in many object detection applications amongst which the Amazon's keyboard detection [2] can be counted.

Chapter 3

Dataset

After a thorough internet search, no dataset suitable for keyboard let alone keys detection could be found at the time of writing this thesis. The Amazon research team struggled with the same issue and was forced to create its own dataset [2]. Nonetheless, not even their dataset seems to be published. Consequently, to achieve the goal of this work, I was compelled to create my own datasets for keyboard detection, keys detection and post-processing corrections. The contribution of my datasets is not only to keyboard typing automation tasks such as mine but also to others like single-character recognition for instance.

3.1 Keyboards

As it turned out, collecting a sufficient amount of keyboards was not an easy task. The primary source of keyboard images became Google and Bing search engines followed by the Pinterest platform. The main issue with the internet keyboard images is their low utility. As discussed in section 1.3, the images for the detection will be full-HD and not distorted nor rotated. Many of the keyboards found by the search engines are either of really low quality, from a perspective or with the user's fingers on them. However, the low-resolution images were still kept as long as the characters were readable. Moreover, these keyboards were usually set in lowercase mode as most of them, from my observation, come from android devices where it is the default mode. Uppercase mode keyboards were significantly harder to find and special characters were particularly difficult to come by. The secondary source was my own mobile phone on which several keyboard applications with different designs and key layouts were installed. From these, all keyboard modes were obtained. Other sources were personal devices such as tablets, e-book readers and smart TVs or company printers on which AIVA units already operate.

In the end, 615 keyboard images were gathered. While it might not sound as much, the Amazon working solution used only over a hundred different keyboard styles for the keyboard detection [2], which I significantly surpass. Even for the keys detection, the Amazon researches labelled keys from only 634 images [2]. This is roughly the same as me and it further proves the difficulty of obtaining a substantial amount of keyboards and also that this amount is sufficient for the task. The keyboards were collected from various device types (smartphones, tablets, desktop computers, laptops, car infotainments, e-book readers, printers, TVs and pin pads) in both digital and physical form. From the original images, only the keyboard regions were kept and placed on a background in a manner specified by section 3.1.1.

To prepare the dataset for training, it is split into train-validation-test in a 70:20:10 ratio. Therefore, 429 keyboards are selected for training, 124 for validation and 62 for testing. The training set is further augmented to 20 000 samples, same as done by the Amazon team [2], using the techniques described in section 3.1.2.

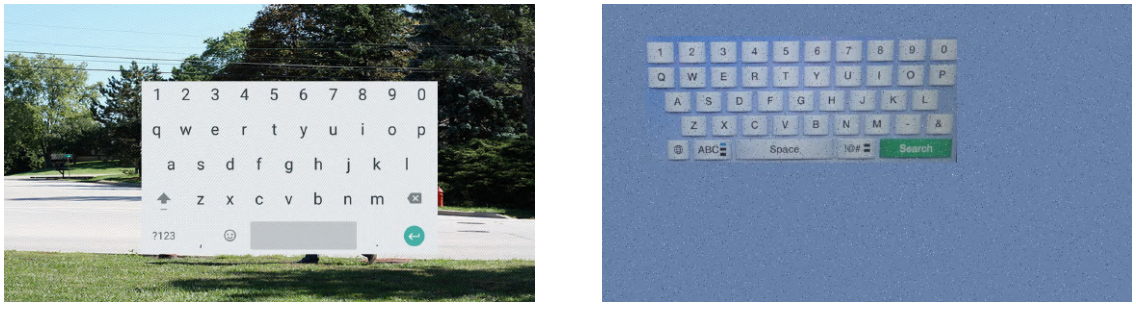
3.1.1 Dataset generation

Placing a keyboard region on a background comes with the benefit of no need for manual object annotation. Exactly one keyboard is placed in each image and an annotation JSON file with the format depicted by figure 3.1 is generated for each training/validation/testing image set.

```
"1.png": [0, 49, 1280, 574],
"2.png": [71, 470, 1017, 692],
"3.png": [237, 179, 1041, 407],
...
```

Figure 3.1: Keyboard annotation file is a JSON dictionary with a filename as a key and bounding box coordinations in format [x1, y1, x2, y2] as a value.

Despite the fact that the object detection input will be a full-HD image, I decided to use HD resolution for backgrounds. The first and main reason is the size. The resulting dataset would attack 100 GB of space and both the training and inference would take an unnecessarily long time. Moreover, the detected region can easily be scaled back so that the keyboard region from the original full-HD image is used later. The second reason is that the Amazon researchers used SSD300 model for the keyboard detection [2], so clearly HD image is still more than enough. In the Amazon paper, they placed the keyboard regions on random images from the COCO 2017 dataset [2]. Although I was inspired by this idea due to the large scale of scene backgrounds, I found it odd that there are hard edges where the keyboard starts in an image. In theory, just being a rectangle of different color might add a lot to the keyboard detection. For this reason, I decided to use the COCO dataset just for half of the images. For another 40 %, the average border color of the keyboard is used as the background so that the keyboard seemingly blends in. Hence, the keyboard keys and characters layout become more important. Figure 3.2 shows the difference. Concerning the rest, 5 % of keyboards are rendered on a completely random background color and 5 % on a random color gradient.



(a) COCO 2017 image as a background

(b) Average border color as a background

Figure 3.2: Comparison of a COCO image (a) and keyboard border color (b) as a background shows how the keyboard in a scene clearly stands out.

A keyboard is placed on a background completely randomly. The only constraint is that it cannot overflow. Conditionally, the keyboard can get resized or made transparent and to the whole image light conditions can be changed or noise added. Moreover, a random moiré effect is always computed as the images under detection will be screenshots from a camera. Section 3.1.2 referes about these modifications in more detail. The whole process summarizes algorithm 1.

Algorithm 1 Pseudocode for generation of single image for keyboards dataset

```
keyboard ← get_keyboard()
if random() ≤ 0.25 then
    keyboard ← resize(keyboard, random(0.5, 2))
if random() ≤ 1/3 then
    keyboard ← make_transparent(keyboard, random(0.5, 1))
bg ← get_background()
img ← put_keyboard_on_bg(keyboard, bg)
if random() ≤ 0.5 then
    img ← change_brightness(img)
if random() ≤ 0.9 then
    img ← add_random_noise(img)
img ← add_moire(img)
```

3.1.2 Data augmentation

Owing to the office or lab conditions in which AIVA operates, no natural elements like rain need to be taken into consideration. Moreover, thanks to the camera calibration system, other modifications such as rotation or perspective can be omitted. In the end, only various light conditions, noise and different moiré effects are the real problems. Apart from these, keyboard size and transparency with respect to the background are also subjected to the data augmentation process.

If we follow the algorithm 1, firstly the keyboard is resized. This happens with a 25 % probability. The image can be changed to any size in the range of twice as small to twice as big as the original. However, boundaries are in place, so that it cannot overflow the background. Next in line is the transparency which is applied with probability 1/3 and it is quite simple as the only thing necessary is adding the alpha channel. The alpha is chosen randomly from interval $\langle 0.5, 1\rangle$. Both of these modifications are applied directly on the keyboard region itself, whereas the following augmentation methods concern the whole image.

To simulate different light conditions, brightness change is applied to every other image. For that, gamma correction is used. Firstly, γ is picked from interval $\langle 0.5, 1.5\rangle$ where $\gamma < 1$ makes the image darker and $\gamma > 1$ makes it lighter. Secondly, γ is applied to the image using equation 3.1. Image must be normalized from $\langle 0, 255\rangle$ to $\langle 0, 1\rangle$ for the gamma application and than converted back. Figure 3.3 shows the difference between the dark, normal and light images.

$$Output = \left(\frac{Img}{255}\right)^{\frac{1}{\gamma}} * 255 \tag{3.1}$$

(a) $\gamma = 0.5$ (b) $\gamma = 1$ (c) $\gamma = 1.5$

Figure 3.3: Comparison of dark (a), original (b) and light (c) versions of an image. The resulting image of the brightness change application can be anywhere in between these values. Note that even brighter images would be viable for most images, however, for some, it could be too much, so these margins were selected for both generalization and equal dark/light distribution.

The next augmentation technique is noise addition. As noise in an image is quite common, it is generated in 90 % of them. Four types of noise with the same probability of occurrence were used. The first one is Gaussian noise. The Gaussian parameters are set to $\mu = 0$ and $\sigma \in \langle 0.5, 10 \rangle$ 3.2. The randomly generated noise is then added to the image.

$$Img = Img + N(0, \sigma^2) \qquad \sigma \in \langle 0.5, 10 \rangle \tag{3.2}$$

The next one is Poisson noise. This noise makes the image very bright though, so the noise values are randomly divided by a number from {2, 3, 4, 5, 6} to reduce its effect 3.3.

$$Img = Img + \frac{Pois(Img)}{r} \qquad r \in \{2, 3, 4, 5, 6\} \tag{3.3}$$

Another noise is salt and pepper. It iterates through every pixel in the image and with a certain probability it sets a pixel to white or black 3.4. This probability is again randomly chosen from interval $\langle 0.005, 0.01 \rangle$.

$$Img[i] = \begin{cases} 0 & \text{if } r < P \\ 255 & \text{if } r > 1 - P \\ Img[i] & \text{else} \end{cases} \qquad r \in \langle 0.5, 10 \rangle, P \in \langle 0.005, 0.01 \rangle \tag{3.4}$$

The last one is speckle noise. To generate such noise, the image is multiplied by a random map from the standard normal distribution and optionally can be multiplied by a variance. The variance is again randomly chosen from interval $\langle 0.05, 0.4 \rangle$ and it controls the noise intensity. In the end, it is added to the original image 3.5. A comparison of these 4 noises is shown in figure 3.4.

$$Img = Img + Img * N(0, 1) * var \qquad var \in \langle 0.05, 0.4 \rangle \tag{3.5}$$

The final data augmentation method is moiré effect generation. This is particularly important as the image subjected to the object detection is usually a target device screen taken by a camera which tends to exhibit moiré pattern behaviour. To create a moiré effect in an image, python `latticegen` library [32] and especially method `anylattice_gen(r_k, theta, order, symmetry)` was used. It is capable of creating parameterized moiré lattices. To set the lattice scale (how big or seemingly zoomed the lattice is) there is `r_k` parameter which I select randomly from interval $\langle 0.01, 0.25 \rangle$. Parameter `theta` stands for lattice

(a) Gaussian noise (b) Poisson noise

(c) Salt and pepper noise (d) Speckle noise

Figure 3.4: Comparison of Gaussian (a), Poisson (b), S&P (c) and speckle (d) noises

.

rotation and it can be anything between 0 and 90 degree angle. The `order` parameter is for simplicity described as "The higher the order, the more well-resolved the atoms are as single spots" [32] which can be seen in figure 3.5. One of the first 3 orders is randomly chosen for the generation.

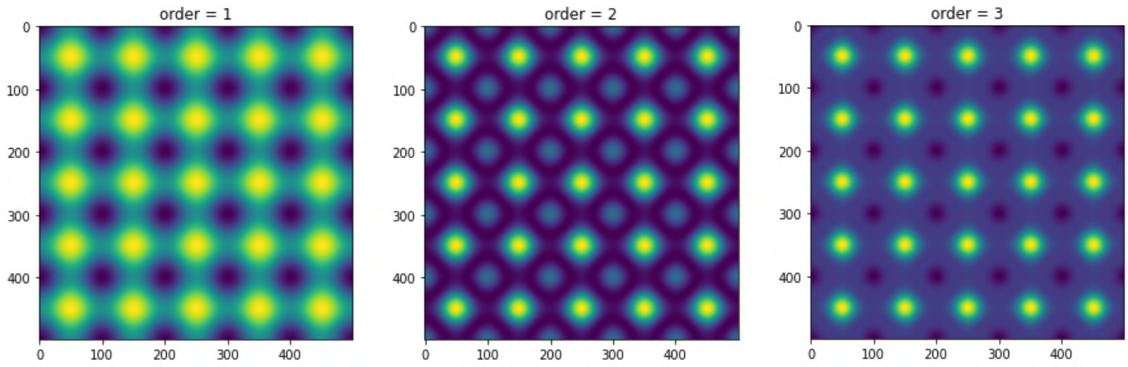


Figure 3.5: Different latticegen orders (adapted from [32])

The last parameter `symmetry` controls the shape of the spots. The most common moiré effect on screens seems to be lines which correspond to symmetry value 1. For that reason, the value 1 is chosen in 50 % of cases. The rest is equally distributed to values {2, 3, 4, 5}. Different symmetries depicts figure 3.6.

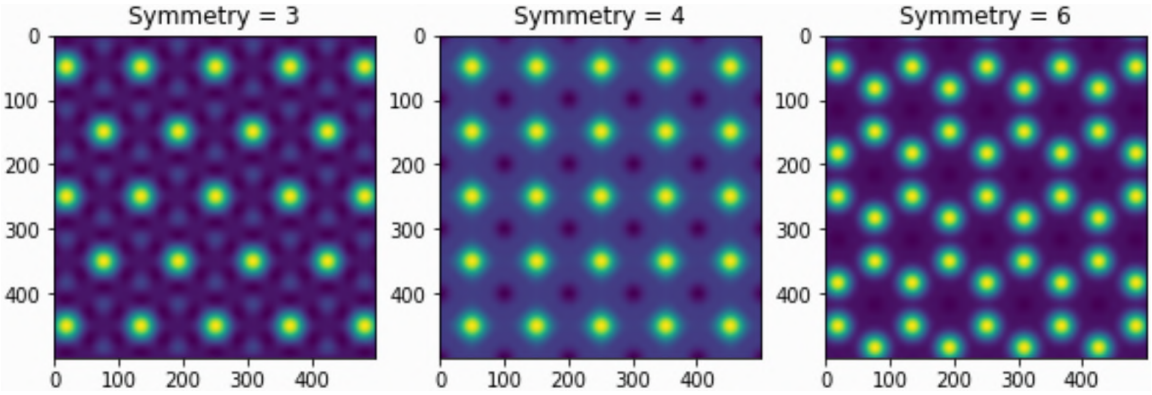


Figure 3.6: Different latticegen symmetries (taken from [32])

So that the lattice is not always regular, a shift taken from the library is applied. For the shift, no further randomization is done as it behaves differently on various lattice designs. To deform the regularity even more, several lattices can be combined together. In the end, either a single lattice or a combination of 2 lattices is used with equal probability.

3.2 Characters

In many ways, the generation of characters dataset is the same as the one for the keyboards. Some characters are selected, put on a background and then noise and moiré effect are added. Noise is created similarly as to the keyboard images. The only difference is that additional gaussian noise is added with 50 % probability. Concerning the moiré effect, the lattice is always of the first order and the symmetry is always 1 (lines) as the special patterns empirically did not influence the image much. Gamma correction is omitted because the characters dataset is in grayscale, so the brightness naturally differs simply by choosing different gray colors. There are several reasons for grayscale images. The most important one is that the keyboards are very contrastive. By converting to grayscale, the characters will always be light on a dark background or vice versa no matter the original color design. Another reason is that processing a grayscale image is easier than a colored one. In addition, to simulate bad-quality images, every third image is blurred by scaling down (randomly up to half) and up again. The algorithm 2 demonstrates the process.

Algorithm 2 Pseudocode for generation of single image for characters dataset

```
char ← select_character()
bg ← generate_background()
img ← put_characters_on_bg(char, bg)
if random() ≤ 0.5 then
    img ← add_gaussian_noise(img)
if random() ≤ 0.95 then
    img ← add_random_noise(img)
img ← add_moire(img)
if random() ≤ 1/3 then
    img ← blur(img)
```

Having described the high level differences to the keyboards dataset generation, the rest of this section focuses on the selection and placement of the characters on the background. The set of characters consists of alphanumeric ones (a-zA-Z0-9), special characters from the sequence .,::'„?!@#$£€%^&(){}[]<>/_+-*÷= and icons for backspace, shift, enter, space and tab. These are repeatedly iterated during the dataset generation. For further reference, an iteration's current character is considered to be the "main character". The background on which the characters are placed is created as follows. The resolution is set to 640x640 for several reasons. A keyboard usually takes around 1/2 of the screen's height so the target resolution 1920x1020 is covered. Moreover, object detectors take square images and the current state-of-the art YOLOv7 uses 640x640. Furthermore, the objective is to find characters of different but bounded sizes, so the background resolution does not play a significant role anyway. As for the background color, it is usually light or dark. The background color probability is shifted towards such pixel intensities on the gray scale according to the formula 3.6.

$$bg\_color = \begin{cases} randInt(0, t) & \text{if } r \leq \frac{1}{3} \\ randInt(255 - t, 255) & \text{if } r \in (\frac{1}{3}, \frac{2}{3}\rangle \\ randInt(0, 255) & \text{else} \end{cases} \qquad r \in \langle 0, 1\rangle, t = 64 \tag{3.6}$$

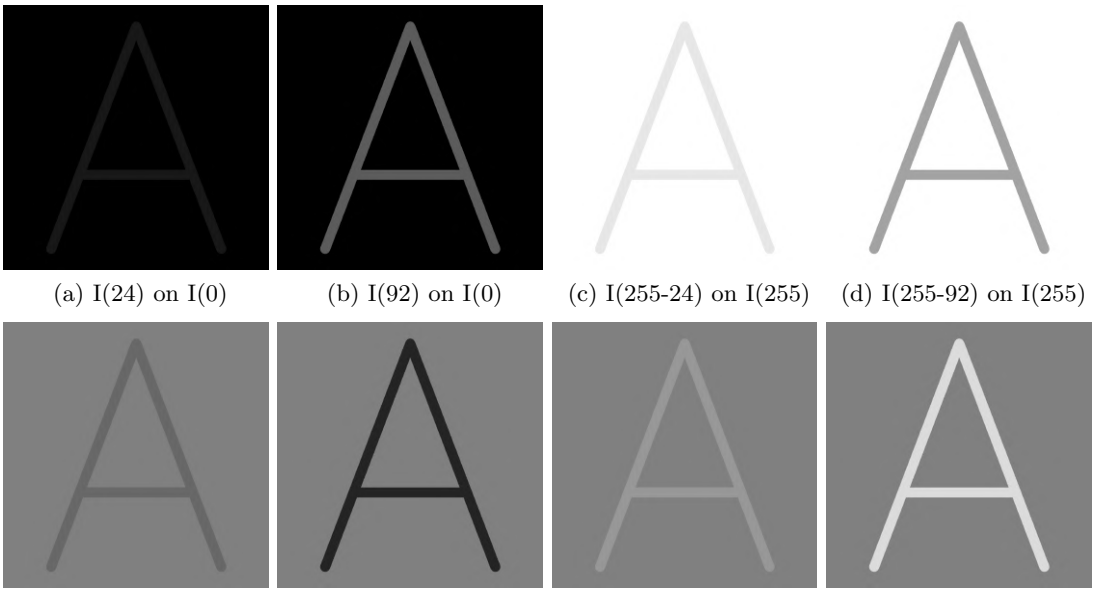
A convenient threshold t for the light/dark boundaries appeared to be value 64. The formula 3.6 basically says that with a 1/3 chance the background color will be dark, with another 1/3 chance that light or else it will be completely random.

The placement of characters on a background is a bit more complicated. The main character is rendered several times on the background and other random characters with it. Different position, color and font, which includes font family, size, weight and style, are selected for each character. To solve the position selection, an attempt to just randomly put characters on the background and check for intersections was made in the beginning. However, the intersections occurred so often that too few characters could be safely rendered which left a lot of free space. As a consequence, a grid system was devised. The largest font size used is 64 which means that after dividing the resolution, from which a random padding is subtracted so that the characters are not directly on a border, a 9x9 grid is created. In order to render not only individual characters but words as well, one or two lines are randomly left out of the grid for word generations. This is because a detector might otherwise not learn to recognize characters in the proximity of other characters. The generated words are either keyboard special words (space, shift, enter...), mode-changing sequences (abc, ?123, #+=...) or completely random character sequences. Out of the remaining (7|8)x9 = 63-72 grid cells, a quarter to half is randomly taken by the main character and another quarter to half by other random characters. As a result, there is no intersection and even the grid is not that apparent as not all grid cells are utilised and different characters of various fonts and sizes are rendered.

Concerning the color selection, the contrast between the characters and keyboard/key background is usually very high, often even black on white and vice versa. Thus, a pixel intensity threshold for the difference between the background and a character color can be set. The formula 3.7 demonstrates the calculation of available colors to be randomly chosen from, where I_{bg} is the intensity of the background pixels and t stands for the threshold. In 75 % of cases, the threshold value is 92 due to the high contrasts and 24 otherwise so that

even unusual low contrastive keyboards can be learned. The visual difference between the threshold pixel intensities with respect to different backgrounds provides figure 3.7.

$$available\_colors = \langle 0, 255 \rangle \setminus \langle I_{bg} - t, I_{bg} + t \rangle \quad t \in \{24, 92\} \tag{3.7}$$



(a) I(24) on I(0) (b) I(92) on I(0) (c) I(255-24) on I(255) (d) I(255-92) on I(255)

(e) I(128-24) on I(128) (f) I(128-92) on I(128) (g) I(128+24) on I(128) (h) I(128+92) on I(128)

Figure 3.7: The first row shows 24 and 92 pixel intensity I(x) differences on a black/white background. The second row depicts the same for a middle gray background bidirectionally.

With regards to the font selection, 46 different font families such as Calibri, Arial, Verdana etc. are used. For most of them, bold and italic versions are available. As italic characters are quite uncommon on keyboards, the normal to italic ratio is 95:5. On the other hand, bold characters are seen more often, so a bold font is used in 40 % of cases instead of the regular one. Font size is selected from interval $\langle 8, 64 \rangle$ with a 50 % chance of inclination towards the lower half.

As there are several characters in an image, the annotation file format needed to be changed. Like in the keyboards annotation, the dictionary key is a filename, however, the value is not a single bounding box for a single keyboard in an image, but another dictionary for characters. In the value dictionary, keys are individual characters and values are lists of bounding boxes as figure 3.8 shows.

Totally, 40 000 character images are generated with train-validation-test ratio 70:20:10 and an example is shown in figure 3.9. In the training set, there are 2 287 613 characters. As there are 99 of target classes, each one has on average 23 107 representations. This is significantly more than my Amazon counterparts who manually annotated characters on 634 keyboards [2]. As they work with 68 characters and 26 (a-z vs A-Z) are usually mutually exclusive, there could be a maximum of 42 characters in each image which results in 634x42=26 628 characters. If uniform distribution of characters is presumed, it makes 26 628/68=391 samples for each character. However, from my findings lowercase characters are more usual in internet keyboard images, so the uniform distribution is doubtful. The benefits of my dataset are the following:

- There is substantially more samples for each character.
- Any character can be generated, not just what is found in an internet image.
- Characters are generated uniformly, none has significantly more samples than others.
- There is no need for manual annotation.

```
"27.png": {
    "A": [
        [463, 193, 491, 223],
        [95, 105, 126, 132],
        [3, 382, 36, 415],
        ...
    ],
    "n": [
        [463, 115, 510, 161],
        [279, 380, 310, 427]
    ],
    ...
},
"28.png": {
    ...
},
...
```

Figure 3.8: Characters annotation file is a JSON dictionary with a filename as a key and a dictionary of character bounding boxes in format [x1, y1, x2, y2] as a value.

Figure 3.9: An image from characters dataset with W as the main character

3.3 Post-processing

In comparison with the previous ones, the post-processing dataset serves only for validation and is substantially smaller. There are 2 major issues which need to be solved after the actual detection. The first one is that a lot of times more than one character is written on a key. On a physical keyboard, the other character(s) is usually accessed by switching keyboard language or by pressing the shift key. On digital keyboards, they mostly just foreshadow what special character is in another keyboard mode. For my problem, only the main keyboard character is relevant as AIVA's robotic arm cannot press multiple buttons. Therefore, these other characters need to be recognized and discarded based on the keyboard layout. The second issue is that not all characters will necessarily be detected or recognized correctly e.g. 'o' vs 'O' vs '0'. This needs to be solved in post-processing and a proper validation dataset must be provided for such a task.

From the collected keyboard regions, 120 which are appropriate to test these issues are selected. These are further split 50:50 into 2 groups. The first group is called *layout* and it validates the correct detection of characters on different keyboard layouts, including the omission of unwanted characters. The second one is named *missing_chars* where not only single characters but also sequences of characters are removed to test if they can be automatically computed. Moreover, even characters in keywords (e.g. *space*) are left out to test partial recognition of keywords for special keys.

Chapter 4

Keyboard and keys detection design and implementation

Among the thesis requirements is not only the keys detection to allow automated typing but also individual keyboard detection so that a decision if an AIVA unit sees a keyboard screen can be made. For that reason, a similar approach to the one described in the Amazon paper [2] is appropriate. In this chapter, a three-phase strategy for automated keyboard typing using current SOTA methods is proposed. Firstly, a keyboard region is detected using the latest YOLO version as described in section 4.1. Making it a separate subtask, said requirement for keyboard screen recognition is satisfied. Secondly, keys are detected in the identified keyboard region which simplifies the recognition in comparison to finding the keys in the whole image. This process presented in sections 4.2 and 4.3 covers the second and third phases, character detection and post-processing corrections respectively. The whole recognition is reviewed and demonstrated in the last section 4.4. Unfortunately, this design cannot be tested against the Amazon's solution as neither the code nor the dataset is to the best of my knowledge available. This fact makes this work even more contributive, as no other seems to currently exist on the topic.

4.1 Keyboard detection

This task can be seen as a standard object detection with just a single class to recognize, a keyboard. The Amazon researchers used SSD300 neural network with great results. Since then, a newer, faster and more accurate architecture has been devised which is YOLOv7. While YOLOv7 was briefly introduced in section 2.2.4, as a selected model architecture, the following section 4.1.1 describes it in more depth. Section 4.1.2 demonstrates, how the model was trained and how it can be used.

4.1.1 Neural network design for keyboard detection

The YOLOv7 paper clearly states that it derives from the state-of-the-art (SOTA) methods and aims to improve loss function, label assignment and training [29]. Hence, the focus is directed on the incremental changes as the base architecture of SOTA YOLO at the time from figure 4.1 is already described in section 2.2.4. YOLOv7 introduces 4 major changes, two architectural (E-ELAN, Model scaling for concatenation-based models) and two trainable bag-of-freebies (Planned re-parameterized convolution, Coarse for auxiliary and fine for lead loss).

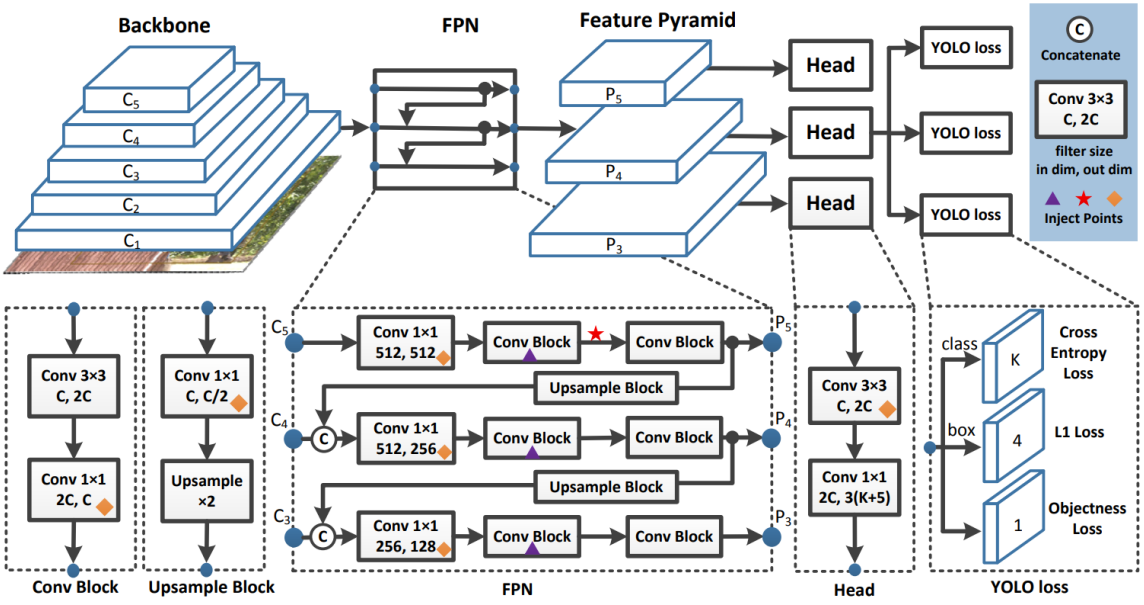
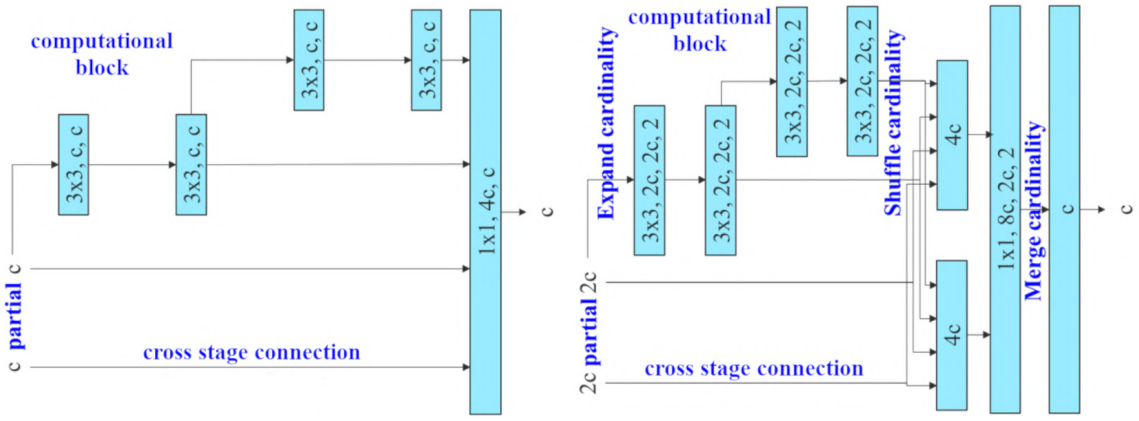


Figure 4.1: General YOLO architecture from which YOLOv7 derives (taken from [33]).

E-ELAN (Extended efficient layer aggregation network)

The backbone consists of E-ELAN computational blocks which depicts figure 4.2. These blocks can be stacked and the depth of the backbone can vary. There were implemented several models (YOLOv7-[X|W6|E6|D6|E6E]) of different depths which are more accurate but slower respectively [29]. The blocks are improvements of ELAN blocks about which not much is known as the paper is not released yet at the time of writing of this thesis. Nevertheless, they are supposed to help control the longest shortest gradient path to achieve more effective convergence [29]. However, they have problems with scaling and E-ELAN solves it using expand, shuffle and merge feature cardinality [29]. For the cardinality expansion, group convolution is used and for more diverse learning, features of different groups are shuffled and merged. Thus, more efficient learning is reached while only the inner block architecture is modified and not the outside so the gradient path is unchanged [29].



(a) ELAN (b) E-ELAN

Figure 4.2: Difference between the ELAN block upon which YOLOv7 builds and Extended ELAN with expand, shuffle and merge cardinality blocks (adapted from [29]).

Model scaling for concatenation-based models

When a concatenation-based model is scaled by depth, the output width increases as (a) and (b) in figure 4.3 demonstrate. Such changes usually affect the structure of a model. To maintain the initial properties of the original design, a novel scaling method as (c) in figure 4.3 is proposed which scales depth and width together [29].

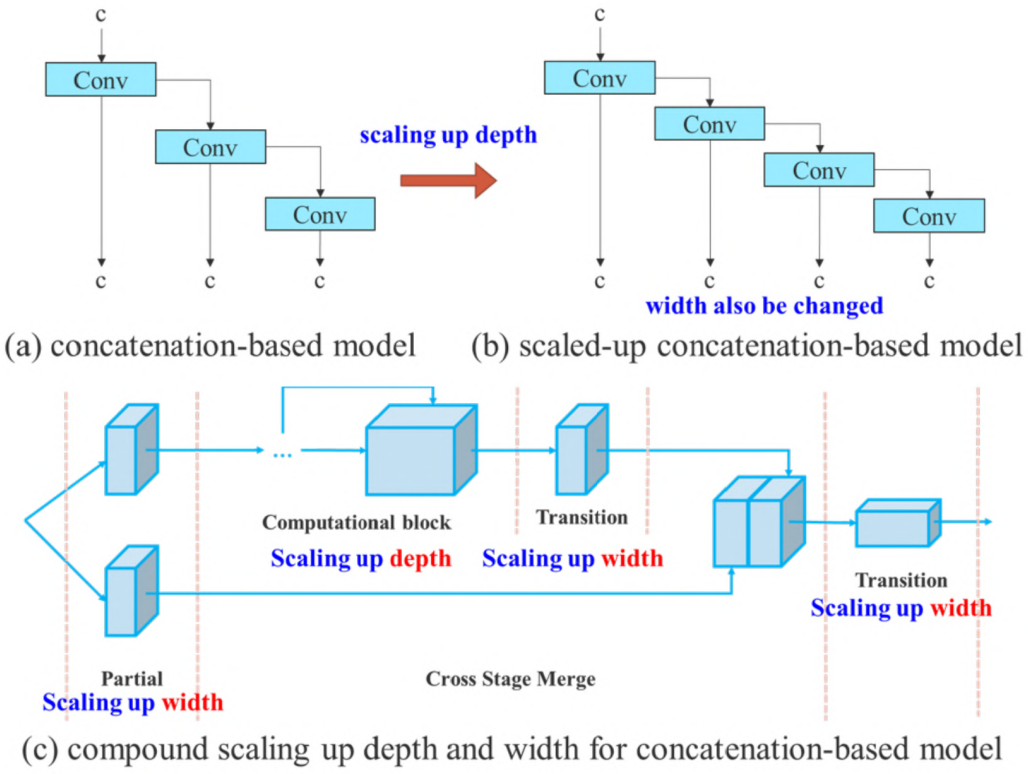


Figure 4.3: Depth scaling is required only inside of a computational block and then the transition layer width is scaled according to the output channel change [29] (adapted from [29]).

Planned re-parameterized convolution

The purpose of model re-parameterization is to merge several computational modules during the inference stage by averaging weights to get a more robust module [29]. The previous YOLO version used heavily RepConv [34] which is a novel highly accurate classification architecture. RepConv combines 3x3 and 1x1 convolutions with identity connection in a single convolutional layer [29]. However, the authors of YOLOv7 discovered that in combination with other architectures such as ResNet or DenseNet the accuracy is reduced [29]. They further analyzed re-parameterized convolutions using gradient flow propagation paths in order to find out how they should be combined with different networks. They discovered that RepConv doesn't perform well in layers with residual or concatenation connections. Figure 4.4 shows combinations which do or do not work. Hence, they proposed RepConvN (RepConv without the identity) to be used instead in such scenarios.

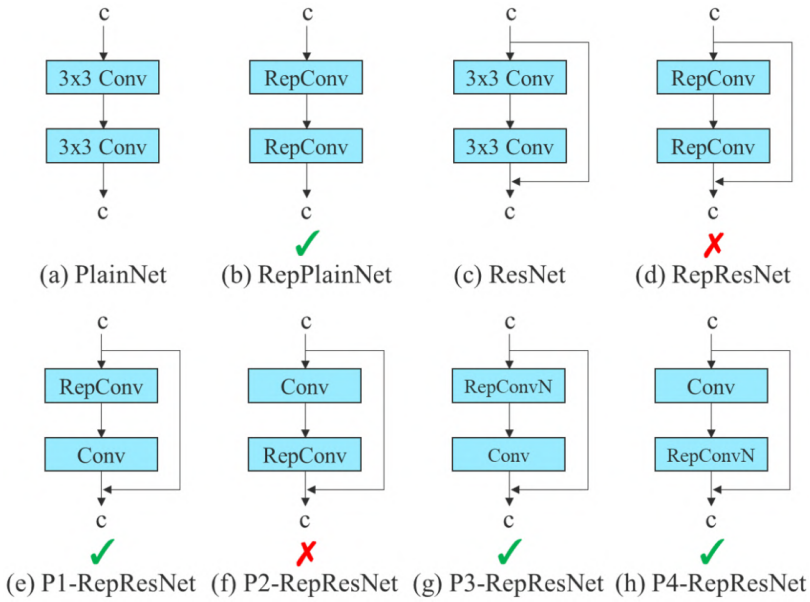


Figure 4.4: Architectures (not) working with identity connections (taken from [29]).

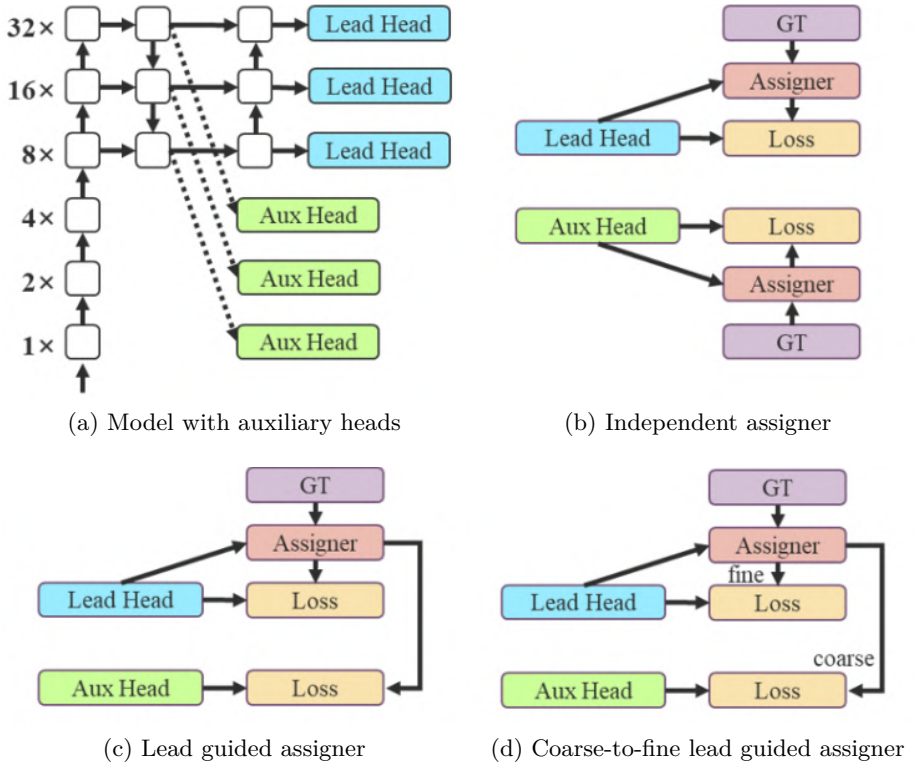
Coarse for auxiliary and fine for lead loss

YOLOv7 introduces deep supervision [35] to the YOLO architecture. Figure 4.5 (a) depicts how auxiliary heads are added to the model. In general, the lead heads are still responsible for the final prediction, the auxiliary heads just help with the training. While figure 4.5 (b) shows the popular approach at the time that both heads make their own prediction, the proposed method in YOLOv7 is to guide both heads by lead head prediction using (c) and (d) assigners [29]. The paper describes them followingly. Lead guided assigner takes ground truth and lead head predictions and generates soft labels which are used as targets for both lead and auxiliary heads during the training. This way, the shallower auxiliary head can learn the predictions from a more accurate lead head and the lead head can in turn focus on not yet learnt information. Coarse-to-fine lead head guided assigner also uses lead head predictions but creates 2 sets of labels. Fine labels are the same as soft labels in the previous assigner. Coarse labels are generated with lesser constraints for positive sample assignment as auxiliary heads have weaker learning capabilities and this can help to avoid information loss.

4.1.2 Training and usage of the detector

The YOLOv7 paper implementation is available on GitHub[1] and is written using PyTorch. Hence the implementation of the whole thesis is in Python as well. The usage of YOLOv7 is very straightforward and is shown by algorithm 3. Firstly, one must install Python requirements (dependent libraries) either globally or in a virtual environment, which is recommended. Secondly, a training run can optionally be initialized in the Weights & Biases platform which the YOLOv7 project uses for logging and metrics collection. In the code 3, this is done from Python while the other commands are for the command line, which is legit if using a Jupyter notebook for instance. Lastly, training itself can start.

[1] https://github.com/WongKinYiu/yolov7



(a) Model with auxiliary heads

(b) Independent assigner

(c) Lead guided assigner

(d) Coarse-to-fine lead guided assigner

Figure 4.5: Deep supervision adds auxiliary heads to the middle of a model (a). Label assigners are usually independent for both lead and auxiliary heads (b), but YOLOv7 introduces assigners for both heads (c, d) [29] (adapted from [29]).

Algorithm 3 Bash and Python code for running YOLOv7 training

```
# set virtual environment not to install dependencies globally
python3 -m venv yolov7-env
source yolov7-env/bin/activate
# install requirements
pip3 install -qr requirements.txt
# Weights & Biases connection - optional and in Python code
wandb.login(key="API KEY OF YOUR ACCOUNT")
wandb.init(project="yolov7", name="yolov7-keyboards")
# train
python3 train.py --img-size 640 640 --cfg cfg/training/yolov7.yaml
--hyp data/hyp.scratch.custom.yaml --data data/keyboards-local.yaml
--name yolov7-keyboards --weights '' --batch-size 16 --epochs 30
```

--img-size This parameter means that both training and testing images have input size of 640x640 pixels. The actual images can have any size, but they will be scaled or padded to this size as the model works with 640x640 images. There are network configurations even for 1280x1280 input images, which are more accurate, but also bigger, slower and unnecessary for this task. The base 640x640 is more than twice the resolution of the SSD300 which is proven to work well.

--cfg The authors prepared several network configurations. The *yolov7.yaml* file describes the paper architecture. Then another significantly smaller and faster variant, designed for microcontrollers and such, is called yolov7-tiny. The last one still for 640x640 input images is called yolov7-x, which is like the 1280x1280 models more accurate at the cost of size, speed and computation resources. These 3 were trained, tested and the most appropriate was selected. There are another 4 variants (W6, E6, D6, E6E) for 1280x1280 input images which were not considered for this task.

--hyp This parameter describes training hyperparameters such as learning rate, weight decay, momentum etc. These were not changed for the training.

--data This one specifies the training and testing data along with the number of classes and class names. The model is trained on the dataset described in chapter 3.1.

--name A name under which the training results are saved.

--weights Specification of pre-trained weights which can be used for finetuning or resuming a halted training.

--batch-size Size of the mini-batch that should be used during the training.

--epochs Number of epochs the training should run for.

To achieve optimal training results, several argument settings were examined. One was whether to use a pre-trained model on the COCO17 dataset, which includes keyboards in a scene. For a demonstration of the difference between finetuned and not finetuned training, chart 4.6 shows the comparison of the development of mAP@.95 metric during 30 epochs using batch size 32. After the first epoch, it is clear that the pre-trained model knows something already. However, the drop after the second epoch suggests that it forgot everything and started training anew. The beginning of the finetuning is quite volatile and after stabilization, it goes hand in hand with normal training. During the not finetuned training, on the contrary, the improvements are steady. A conclusion can be drawn, that the finetuning does not add any value to the training results, and hence can be omitted.

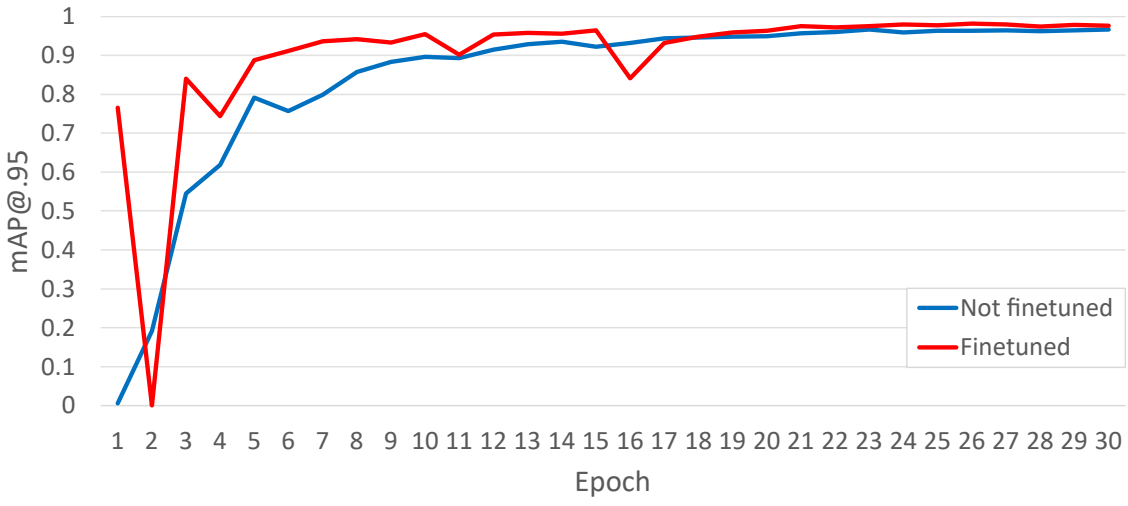


Figure 4.6: A comparison of mAP@.95 development on validation data during training a new model and finetuning an existing one proves that finetuning does not bring any benefit.

Another argument selection concerns the batch size. Usually, a power of 2 is selected, namely one of 32 or 64. The YOLOv7 default is set to 16, so it was compared to 32 if it

brings any improvements. Chart 4.7 displays the difference. Lower batch sizes generally slow down the training as the weights are updated more often and the vectorization potential is reduced. Conversely, the more frequent weight updates might (not a rule) improve accuracy.

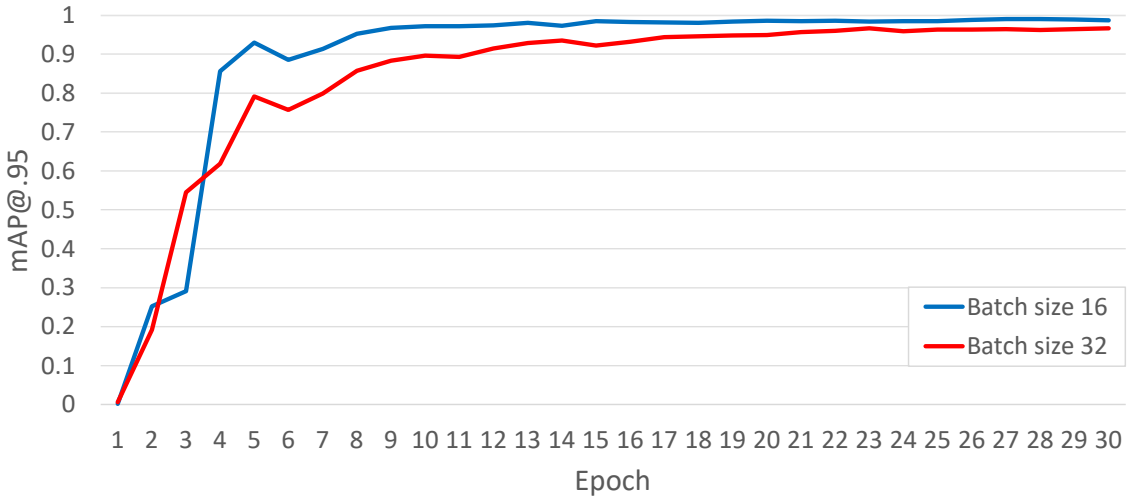


Figure 4.7: A comparison of mAP@.95 development on validation data during trainings with batch sizes 16 and 32 clearly shows that batch size 16 achieves better results.

The remaining parameter, the number of epochs, was set to 30 for visible reason in both charts 4.6 and 4.7. Most of the learning occurred in the first 10 epochs, in the next 10 there were significantly smaller improvements and in the last 10 it stabilized completely. In this manner were trained all 3 640x640 models (yolov7-tiny, yolov7, yolov7-x) and the results are further discussed in chapter 5. Full training results are available on the attached medium A.

Once the model is trained, the YOLOv7 project provides *detect.py* script for the actual detection. However, this is not usable in a custom solution. Therefore, Python pseudocode 4 demonstrates the model usage. Firstly, PyTorch loads the model and sets it from training to evaluation mode. Secondly, the given image is transformed into the detector's format, which means scaling and adding padding to 640x640 size and normalizing color values from 0-255 to 0-1. Then the model makes its predictions on which the non-maximum suppression method is applied. By default, only the detections with a higher confidence score than 0.5 and only the best from the overlapping ones (iou = intersection over union threshold) are selected. Lastly, the detected bounding boxes are scaled back to the original image size.

Algorithm 4 Python pseudocode for a detection function using YOLOv7 model

```
1  def detect(model_path, img, confidence=0.5, iou=0):
2      model = torch.load(model_path)
3      model.eval()
4      input_img = prepare_img(img)
5      predictions = model(input_img)
6      predictions = non_maximum_suppresion(predictions, confidence, iou)
7      predictions = scale_to_original(predictions, img)
8
9      return predictions
```

4.2 Keys detection

Recognition of keys on a keyboard is the most challenging part of this work. The Amazon research team used modified SSTD (Single Shot Text Detector) [36] architecture. SSTD serves for text detection in a scene image. However, it only finds text regions and does not recognize the text meaning [36]. The main modifications made were output change to support multiple character classes and addition of a non-maximum suppression layer so that it does not try to join characters into words [2]. Nonetheless, I found 2 issues with this approach. One of them is that I do not see the point in using a scene text detector and reducing its capabilities. Firstly, there is no real-world scene as keyboards are basically just characters on a background with some UI design. Secondly, the attention module for text prediction in SSTD misses the mark once the task is reduced to single-character detection. The other problem is for example the space key. In many cases, the key is empty without any icon or "space" word. This leads to a realization that just character recognition is not sufficient and a method for the actual key (not its text) detection must be included. The proposed solution is to use the current SOTA YOLOv7 for the character recognition according to section 4.2.1 and Canny edge detector described by section 4.2.2 for checking the positions of those keys that have predefined locations such as some special keys. This way there can be predicted potentially missing or incorrectly detected keys.

4.2.1 Neural network design for character detection

As mentioned in sections 2.2.4 and 4.1, YOLOv7 is the current SOTA in object detection so it was selected for this task as well. Text detectors like SSTD or other OCRs usually aim at detecting words or even sentences in a scene. In order to achieve this, attention mechanism and other text-specialised methods are used [36]. This would make them more suitable models if text detection was the objective. Nevertheless, single-character detection is meant to be solved. Consequently, the contextless feature of YOLOv7 is beneficial as it does not try to merge the characters into words.

The model was trained on 640x640 images from the generated dataset described in section 3.2. To fully disclose, there was no illusion of counting on great accuracy results, especially for special characters such as dots, commas, semicolons etc. as for instance a semicolon can be recognized as independent dot and comma. Many other examples like this one can be thought of. A lot of such detection inaccuracies are solved by the detection correction post-processing mechanism 4.3. Moreover, to remind the task specification from section 1.3, the main focus is on alphanumeric characters. The endeavor to recognize special characters as well was made but as a secondary objective with a separate evaluation.

Concerning the actual training of the model, it is the same as for the keyboards described in section 4.1.2 with just 2 subtle differences. The first one is an increased number of epochs to 50 as there are 99 classes and it takes longer to train. The second one is a modification of the original hyperparameters configuration file. Flipping of objects during training was turned off because for example flipping the left parenthesis results in the right one which is a different class. Moreover, changes in color, saturation and scale were reduced as it was already the subject of dataset generation. The detector is further evaluated in chapter 5.

When it comes to the usage of the model to detect characters on actual keyboards, an inconvenience comes up. Keyboards are usually very wide which is adversarial to the scaling. Let's assume a common scenario with a digital display where the keyboard covers about half of the screen. On the target full-HD image it would mean 1920x540 keyboard

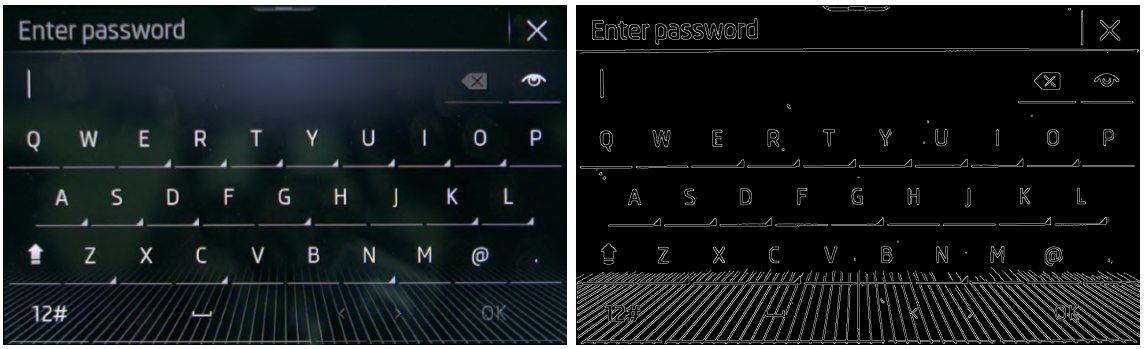
resolution which would scale down to 640x180. This results in significant information loss and 2/3 of the image under detection being padding. Instead of simply scaling the image down, it is split and several detections are run. Full-HD width 1920px is convenient as it is exactly 3 times the width of the detector's input. However, a certain overlap is necessary to avoid cutting a character in half during the split. The overlap used is 64px which is the biggest font size on which the model was trained. Consequently, the input image is split into several images which fit the 640x640 block. In the case of the full-HD image with the overlap included, it is 4 images, but the algorithm is flexible so it is always split into the right number and no information is lost. The splitting is done only on the x-axis, because 640px height is not expected to be exceeded or just slightly. Should it be significantly larger, the information loss during the scale-down is completely acceptable. Of course, multiple detections influence negatively the performance. Nevertheless, the images can be batched and run in a single computation just as another dimension of the matrix. Furthermore, as YOLOv7 is a real-time detector with detection time in milliseconds depending on the machine, the slow-down is tolerable for the use case.

4.2.2 Key regions proposal using classical computer vision techniques

The examples provided in section 2.1 show that both edge detection and thresholding can produce satisfying results. Due to their similar results and no obvious reason why one should be better than the other, the Canny edge detection technique was chosen as the classical algorithm candidate because it is a bit simpler to use. The thresholding requires slightly more operations. Owing to the simplicity and speed of this method, the initial idea was to use it as a region proposer and then a classifier could be used to recognize the characters. This strategy seemed to solve it all. It could handle empty keys like space and achieve outstanding accuracy as there already exist many character classifiers and datasets such as MNIST. Despite of initial successes, though, two major issues concerning both edge detection and thresholding arose. The first one is contour computation depicted by figure 4.8. As there can be seen, many smaller regions were often detected instead of a full key. This might happen when an edge is not complete and has spaces in it which compromise the contour. Another complication is the detection of both a key and a character. Which one to use? The inner box (character) would be more suited for the subsequent classification but then which to choose e.g. between t and 5? It could be decided based on the size but what if for instance j has its dot isolated or another contour error occurs? Too many inconvenient scenarios emerge which does not play well for this strategy.

Figure 4.8: Contour computation problem in edge detection region proposal

The second issue is the variety of UI designs that can create a huge amount of false positives. Something similar might happen with a background in a transparent keyboard. An example of such UI design is shown in figure 4.9. Image (a) shows a car infotainment keyboard with vertical lines going to the background at the bottom of the keyboard. This, however, renders the contour computation completely useless as image (b) demonstrates. There would be a huge amount of bounding boxes produced not only for the bottom vertical lines but also for the horizontal lines under the characters symbolizing the keys.



(a) Original image (b) Binary image after Canny edge detection

Figure 4.9: UI keyboard designs can generate too many edges.

What is more, it would be throwing out the progress in the object detection field and returning to two-stage detectors. For these reasons, a detection neural network approach was opted for in the end. Nevertheless, Canny edge detection is still run in parallel as a supplementary bounding box provider of potential keys for the post-processing phase. Firstly, it costs almost nothing as it is computed parallelly. Secondly, the neural network does not solve the empty space key problem. If there is no icon or word on the space key or it is not detected, the Canny edge detector might detect the key's rectangle. The same goes for other special keys such as shift if the icon or word is not detected. Which special keys are double-checked and how is the subject of section 4.3.6, the current section concentrates on how the bounding boxes of candidate key regions are obtained. The Python pseudocode 5 describes the process.

Algorithm 5 Python pseudocode for Canny edge detection using OpenCV

```
1  def run_canny_detection(img):
2      img = clahe(img)
3      img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
4      img = cv2.bilateralFilter(img, 8, 30, 70)
5      img = cv2.Canny(img, 25, 50)
6      contours, _ = cv2.findContours(img)
7      bboxes = convert_contours_to_bboxes(contours)
8
9      return bboxes
```

Before the actual edge detection is performed, several image modifications to improve the detection results are done. To highlight the edges a bit more, the clahe [37] contrast enhancing method is used. Then the image is transformed into a grayscale representation.

Finally, bilateral filtering [7] is used to reduce noise while keeping edges. The second parameter 8 specifies the pixel neighborhood distance which influences the other parameters. The third one 30 is a relative measure of how much are the colors mixed together. The last parameter 70 is a relative measure of how much the pixels influence each other taking the color into account as well. The subsequent Canny method accepts the lower and higher thresholds described in section 2.1.1 and outputs a binary image with edges. Contours are computed for the edges and converted to [x1, y1, x2, y2] bounding boxes.

4.3 Post-processing of keys detection results

This is the third and final stage to the whole process and it has several subtasks. In general, it tries to validate the detected keys, correct any inaccuracies and fill in the undetected keys. For this purpose, supported layouts, keywords and other patterns are predefined as rules and the detected characters are checked against these rules. To make an example, let's assume we have a detected "r" character and we want to check if it is the "r" in a qwerty row. The rule for this scenario says there should be "qwe" left of the character and "tyuiop" right of it. If there is a sufficient number of matching characters in the correct order, the check passes. Then the distances between characters are obtained and missing undetected characters can be computed or inaccurate bounding boxes corrected. Any sequence can be checked in such a manner and the following sections describe each part in more detail. The rule checking is the same, though. Before diving into the specifics, algorithm 6 demonstrates the whole process.

Algorithm 6 High-level key detection post-processing process

Input Recognized characters *chars* and Canny edge detection results *canny_bboxes*
Output Processed and accepted keys

1: $processed \leftarrow \{\}$
2: $rows \leftarrow find\_rows(chars)$
3: $check\_pinpad(rows, processed)$
4: $check\_keywords(rows, processed)$
5: $layout \leftarrow check\_layout(rows, processed)$
6: $check\_number\_line(rows, processed, layout)$
7: $check\_icons(rows, processed, layout)$
8: $check\_special\_characters(rows, processed, layout)$
9: $guess\_special\_keys\_from\_canny(processed, canny\_bboxes, layout)$
10: $correct\_capitalization(processed)$

In the beginning, none of the detected characters is accepted yet. Every detection is added to the result only if it matches a rule. To have an idea about the detection placements and their relationships, they have to be organized into a structure. Simple rows were selected. For two bounding boxes to be in the same row, they must overlap on the y-axis for at least 50 %. The rows are ordered bottom up and characters in a row are ordered left to right by the x-axis. This is sufficient because keyboards are basically grids and it is easy to check a character's row for a sequence rule. Next, a pin-pad layout existence is checked. This is necessary to do before a layout recognition because otherwise the common alphabetical characters below or next to the pin-pad numbers could be incorrectly viewed as an alphabetical layout. Once it is clear whether a pin-pad keyboard is being processed or not, qwerty vs. alphabetical layout recognition can begin. Nevertheless, to reduce the

number of characters in the layout recognition, keyword detection precedes it. Following is a 1-9 number line check, which is a very common sequence on keyboards. Notice that zero is omitted as it can be either at the start or end (more usual) of the sequence. Then, special key icons and special characters are processed. These are basically the remaining detection results and with them ends any positional or correctional processing of the character recognition. Thereafter, the Canny edge detection results are used to guess undetected special keys based on the already detected ones. Finally, correction of lower vs. upper case is performed. A lot of characters such as x vs. X or s vs. S etc. are hardly distinguishable from each other and the character detector can easily make a mistake. For that reason, discriminative characters "abdehmnqrty" whose lower and upper variants differ based on intuition and character detector results are selected. All characters are set to the prevailing case among the discriminative characters. It is worth mentioning that at the very beginning of the post-processing, all characters are transformed to lowercase for easier manipulation and the information about their original case is saved for this part.

4.3.1 Numbers processing

There are 2 numbers-processing tasks performed. At the very beginning of the post-processing, a search for a pin-pad layout is run. The x coordinates of all numbers are checked against each other if they overlap to find potential columns. The column patterns are 147, 258, and 369 so if for instance 4 overlaps with 7, it is considered a column. A third character is allowed to be missing to be computed. Then row patterns 123, 456, and 789 are searched for. If both a column and a row rules match, the pin-pad layout is recognized and any missing number is computed based on x and y differences in found rows and columns. The method also counts with the possibility of both 123 and 789 rows being at the top and at the bottom of the pin-pad. Nothing else is expected among the pin-pad characters, not even the usual alphabet characters as those are not keys, so every other detection inside the pin-pad layout can be removed. Finally, a zero is expected below the pin-pad so if any detected zero is positioned there, it is accepted as well.

The second task is the detection of a number line "01234567890" where only one zero is expected but it can be on either end. This takes place after the layout detection which is convenient. If it is a qwerty layout, it is expected for the number line to be above the top layout row. In alphabetical layout or undetected layout (special characters mode or failed character recognition) it can be anywhere. Moreover, in the second case, the line doesn't need to be full but cut and continued in the next row. Therefore, the process is the following. Every detected and not yet accepted number is checked against the number line sequence rule. The check passes if there are at least 3 matches. In the case of a qwerty layout or full line acceptance, processing ends. Otherwise, the line could be cut and the action is repeated for the rest of the numbers.

4.3.2 Keywords processing

The reason for keyword recognition is that the special keys might not have their icons but their names on them. Furthermore, some special keys do not even have an icon. Because the keywords contain many characters which could negatively influence the layout recognition, the search for them happens before it and right after the pin-pad check. Moreover, the number of characters to be processed in layout recognition is reduced this way. Each special key can have several possible keywords and specific constraints. Generally, the requirement

for a minimum number of matched characters in a keyword for a keyword rule to pass is 2/3 of its length. The target special keys are the following:

- **Tab** — This key has a single keyword "tab". As it has a length of 3, two characters are sufficient for a match. However, due to "ab" being a very common sequence, the "t" needs to be one of the recognized.
- **Caps Lock** — Text on this key is sometimes shortened to just "Caps" or "CapsLk". Therefore, only the discriminative part "caps" is used for the keyword. Furthermore, detection of "lock" can be misleading as it might be contained in another key such as Num Lock.
- **Shift** — There is a single keyword "shift" for the key without any special treatment.
- **Space** — A lot of times, especially on multilingual keyboards, the currently set language is written on the space key. Hence, not only "space" but also "english" keywords are supported for this key. In addition, the "space" keyword is checked for conflict with "backspace" in favor of the latter.
- **Backspace** — Apart from the "backspace" keyword, this key has two additional shortened variants "backsp" and "bksp" which both are supported.
- **Enter** — Enter key supports "enter" and "return" keywords. The latter is especially common on android devices.
- **Mode** — This key is the most problematic one and supports many sequences which are commonly used for the key such as "abc", "123", "?123", "@#&", "#+=" and many others. The potential problems lie in "abc" vs. alphabetical layout, "123" vs. pin-pad/number row or special character mode sequences vs. special characters on a special character layout. Moreover, the "abc" usually means shift on a character layout (qwerty, alphabet) and mode (switch to characters) on a special character layout. Therefore, this keyword is saved for a retrospective classification based on the detected layout and other detected keywords or icons. If there are other shift and mode keys recognized, it is ignored. Otherwise, it is the undetected one or a shift in a character layout or a mode in a special character layout.
- **Page** — This is an edge case concerning special character modes. Sometimes, more than one page of special characters is available, so this key searches for "1/2" and "2/2" sequences with the slash character as a requirement.

Besides the sequence rules the distance between the characters must be checked. While in a layout the characters are expected to have unknown spaces between them, keywords should have the characters in close proximity. Therefore, if the characters are too far from each other where the tolerance is set to one-fifth of the mean character width, the potential keyword is ignored. Similarly, if there are some undetected but expected characters and there is not enough space for them, the candidate is not considered either.

When a candidate keyword passes all validations, it is not yet added to the results but saved aside. The reason behind it are potential uninteresting words that might confuse the processing and cause mistakes. When a keyword is detected, it is saved only if it is the first of its kind or better than the already recognized one. Which one is better is determined by the number of detected and skipped characters. As a consequence, only one special key of a kind is added to the accepted results. So if there are for example two shifts, which is common on physical keyboards, one on the left below Caps Lock and one on the right below enter, only the more confident detection is used. On one hand, a shift detection is lost, on the other, it is absolutely of no significance as the key is detected and can be used.

4.3.3 Layout processing

The goal of this part is to find an alphabetical or a qwert[y|z] layout among the detections. To reduce the number of processed characters and to avoid errors as much as possible, only layout discriminative characters "dgkmquvwx" which are not part of basic keywords (tab, caps, shift, space, enter) are used. Also, they are equally distributed by 3 in each qwerty layout row. Each of the discriminative characters is checked against predefined layout rules. The rule sequence for alphabetical layout is "abc...xyz" where it is expected that it can be cut anywhere. The qwerty layout has sequences for each of its rows ("qwertyuiop", "asdfghjkl", "zxcvbnm") and any matched sequence suffices to identify the layout. Undetected characters can be computed even in other rows. The minimum matched characters requirement is only 2, so with the currently processed character it makes 3 detected characters in a row for the row to be considered as a layout row. This allows for a lot of character detector mistakes which can be corrected but at the same time more rows can be matched as the same sequence or one can match both layouts. Thus, all matches are saved as candidates and the best one is selected based on the number of detected and skipped characters. The winner's layout is set as the recognized and is used as a template for missing characters computation. This is easy as both x and y distances between characters are known from the detected ones.

In the case of a qwerty layout, additional processing can be done. Firstly, unmatched rows can be computed if at least one character is detected in them. This is not possible in an alphabetical layout as it is unknown, where the row might be cut. Hence, the most confident detection of each not yet accepted character is used but nothing else is computed. In qwerty layout, on the other hand, the rows are exactly defined, so the rest of the row can be computed from a single detected character. Secondly, any other detections inside the qwerty layout are not allowed. Those can be either false positives or for example special characters foreshadowing their positions in another mode as figures 4.10 and 4.11 depict.

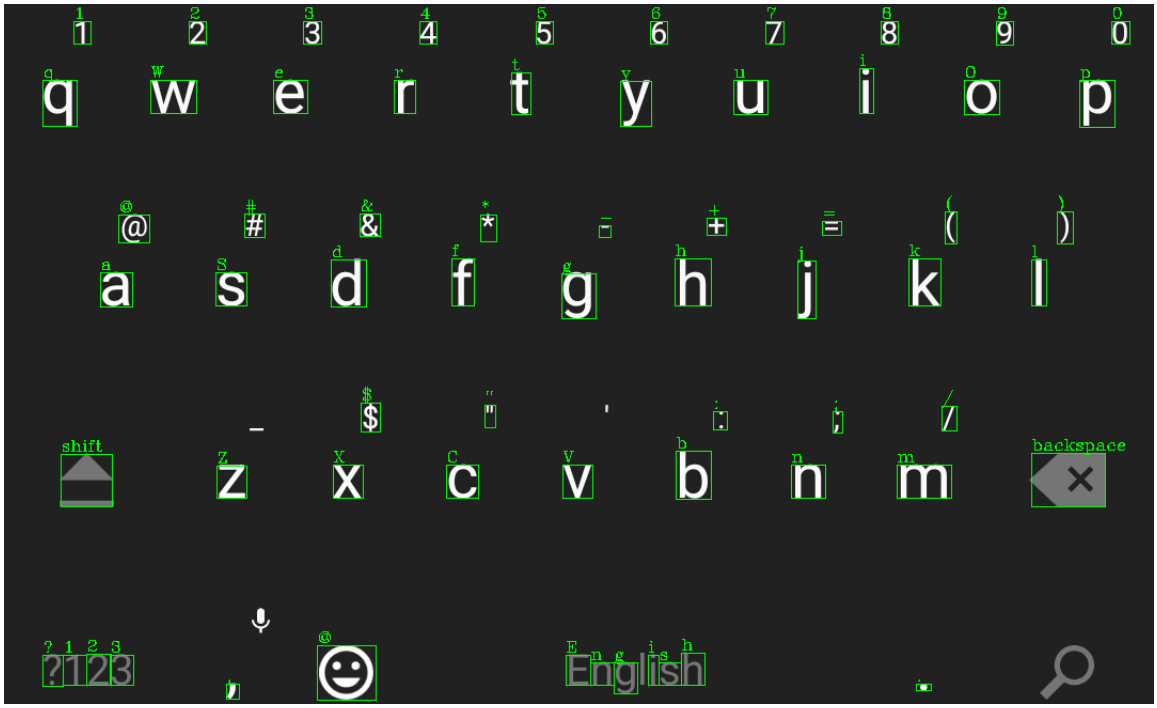


Figure 4.10: The detector can find characters in a qwerty layout that are not keys.

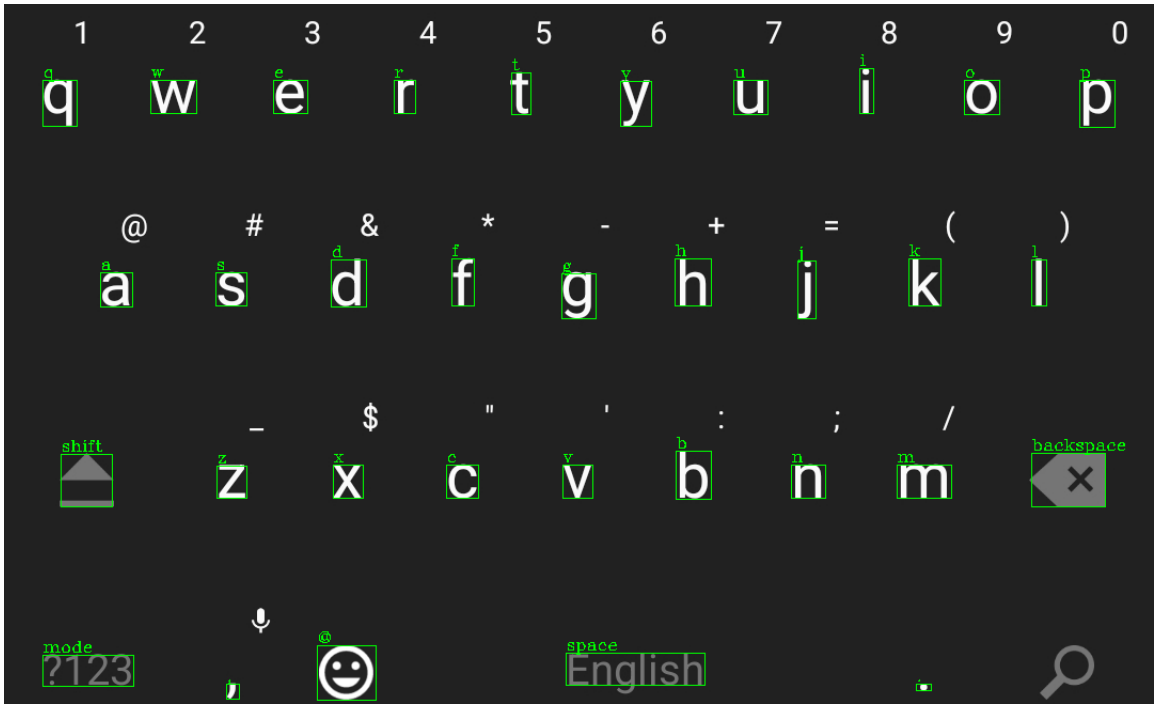


Figure 4.11: Characters inside the qwerty layout are removed during the post-processing. The numbers above the top row are removed as well due to a margin added to the layout box for this very reason. Characters can be even horizontally outside as the right parenthesis in the middle row demonstrates which is another reason for the margin.

4.3.4 Icons processing

Detected icons are processed after the layout and keywords are already known. This helps with icon validation and improves false positives detection. Generally, either a keyword or an icon is displayed on a special key, so a recognized keyword might be an indication of an invalid icon detection. Therefore, the keyword takes precedence and if it is detected, the icon is ignored. Also, the icon position is taken into consideration relative to the layout. Unfortunately, only the qwerty layout offers more or less standardized special key positions, so in the case of other layouts, simply the icons with the highest detection confidence scores are used. The following describes the additional qwerty layout checks or other nuances for each icon of interest:

- **Backspace** — The key should be to the right of a qwerty layout so it accepts the best only among such detections.
- **Enter** — The check behaves exactly the same as for backspace.
- **Shift** — The expected and preferred position is to the left of the “z” key. It might not necessarily be there, though. Next, it looks among those below “asd...” row. If not even there, the last check is anywhere to the left of the qwerty layout. It is not expected for a shift to be anywhere else.
- **Tab** — Tab is the most constraint special key. It is accepted only on a qwerty layout and only in its expected position which is left to the “q” key. Thus, the rule of keyword precedence works differently here. Retrospectively, the position of the tab keyword

is checked and it is removed if it is not correctly placed. This makes the icon the preferred choice, however, also only in the right position.

- **Space** — Similarly to the tab key, the icon can take precedence if it is better positioned. Should the keyword be in the bottom part of the keyboard, any space icon is ignored. On the contrary, the best icon is used with the preference of being below the bottom qwerty row.

4.3.5 Special characters processing

There is only one pattern searched for among the special characters and it is !@#$%^&*(). This sequence can be found above the qwerty layout or in special character layouts. Other special characters are not part of any rules and from those, only the most confident detection is used. Concerning the pattern, there exists a special case for qwerty layouts. Usually, the special characters above qwerty are on the same keys as numbers, where numbers are the main characters. For that reason, the existence of a number line above the top qwerty row must be checked. If it is present, the special character line is ignored. Moreover, any special characters above the number line, special character line, or top qwerty row are removed as those are considered the topmost keys in qwerty layout keyboards.

4.3.6 Canny detections processing

The objective is to find missing special keys. Unfortunately, too many unknowns limit this task. To begin with, non-qwerty layouts tend to have special keys anywhere so this action is constrained only to qwerty layouts. Moreover, tab or caps lock keys are not very common on smartphones and generally on touch screens, hence it is not even safe to assume their existence. Concerning enter and backspace, they should be present almost always and also the position ought to be to the right of the layout. However, their y position is the problem. The backspace key can be at the top on physical keyboards, at the bottom on smartphones, or even anywhere in between depending on the keyboard UI design. The same goes for enter key which has the usual middle position but that is not a rule as well. This leaves only shift and space keys. The space is one of the main reasons for the edge detection as it can be blank without any text or icon and there is no other way to recognize it. In addition, it can be recognized quite easily as it has its fixed position at the bottom in the middle and is significantly wider than all other keys. Then the shift key usually sticks to its position to the left of the bottom layout row. Furthermore, if it is not so, it is unlikely another key has its place. Thus, it can be relatively safely guessed that the found bounding box is indeed a shift. However, both space and shift key guessing occurs only if the corresponding key has not been found.

The input for this process are results of the detection from section 4.2.2. The accuracy might vary from very precise to a lot of noise bounding boxes which is the downside of a method with fixed parameters. Therefore, filtration is necessary. As the special keys are bigger than the character layout keys, the median size is computed and any smaller bounding box is removed. Furthermore, only bounding boxes from the bottom left quarter of the image are considered as neither of the keys is expected to be elsewhere. An example of filtered bounding boxes is shown in figure 4.12. Once the filtration is complete, the bounding boxes matching the expected coordinates are selected as the special keys.

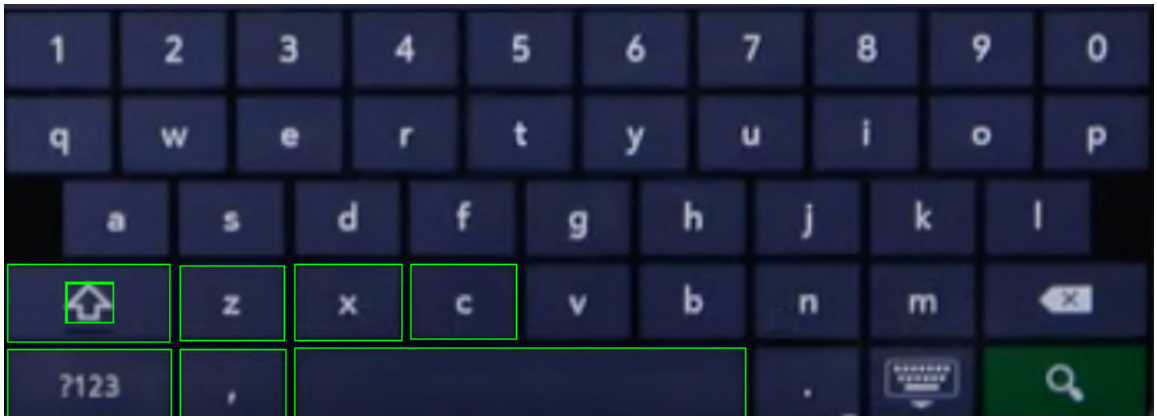


Figure 4.12: Canny detection targets shift and space keys which are constrained to the bottom left quarter of the image.

4.4 Final detection process

So far, individual parts of the detection have been described. The whole process is summarized in figure 4.13. The image processing starts with feeding the input image to a trained YOLOv7 keyboard detection neural network model as outlined in section 4.1. The output can be either used as a result of the required individual keyboard detection task or fed as input to the subsequent character detection. On the detected keyboard region is simultaneously run character detection using trained YOLOv7 model and Canny edge detection described in sections 4.2.1 and 4.2.2. The results of these two detections are then processed in the final post-processing phase presented in section 4.3. The final results are bounding boxes with their confidence scores for recognized characters and keyboard in JSON format. Optionally, images with rendered bounding boxes can be saved as well. The solution is very modular and it is easy to switch or retrain a model if needed as well as include any additional post-processing. A recognition running script is provided with the options of running full or partial (just keyboard or no post-processing) recognition.

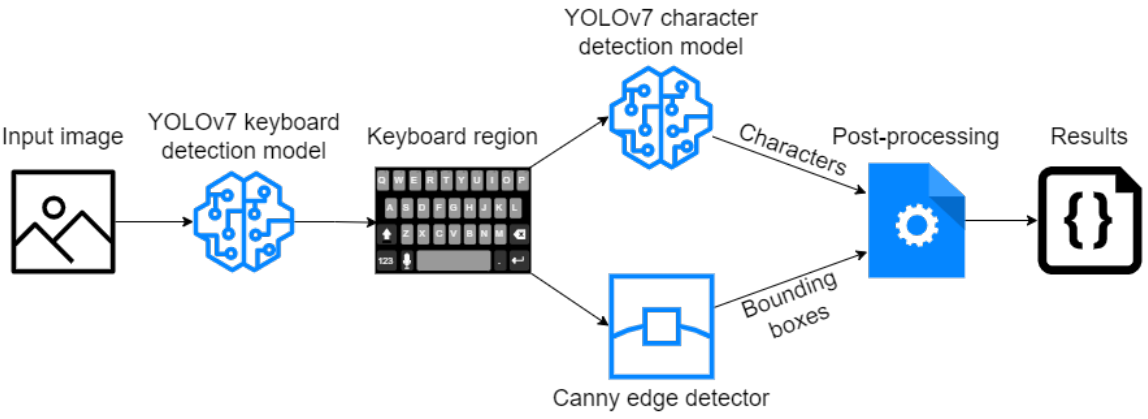


Figure 4.13: Flow of the image recognition process

The image processing is also visually demonstrated by the following figures. As an example is used one of the images from the validation dataset. It illustrates changes done

by each recognition stage and also several post-processing corrections. The first step of the process which is the keyboard region detection in the input image is shown in figure 4.14.

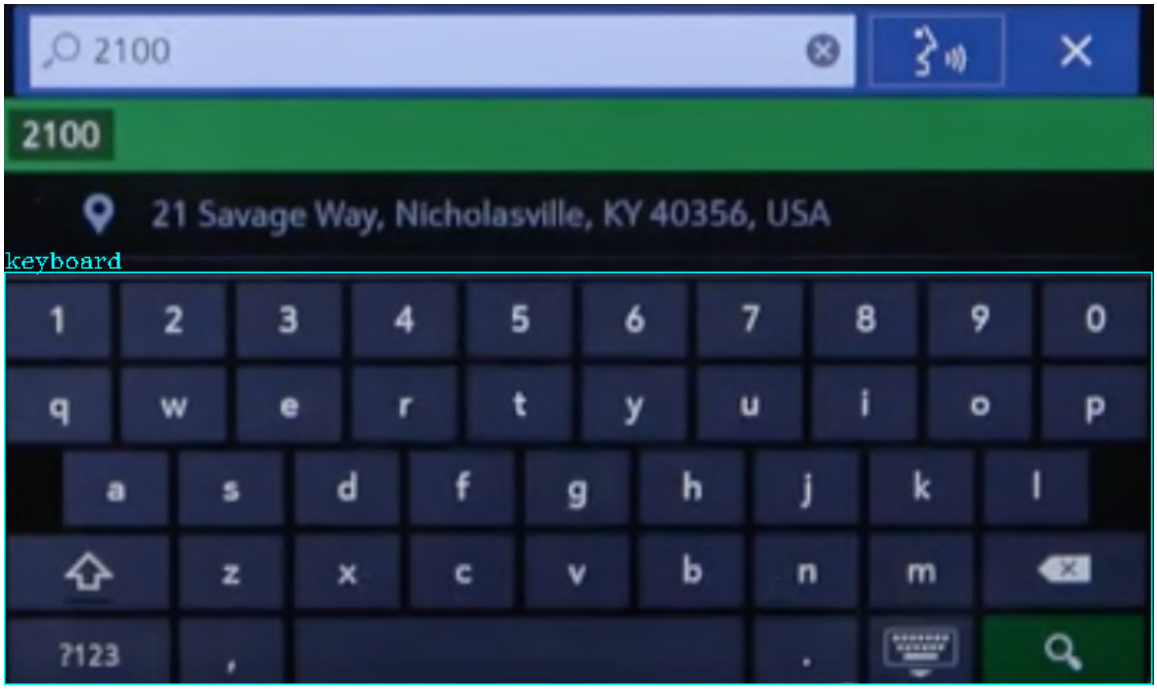


Figure 4.14: A keyboard region is detected in the input image using the YOLOv7 model.

In the next step, the detected keyboard region is used as input for the detection of keys. Figure 4.15 depicts single-character detection results. This detection is run in parallel with the Canny edge detection described in section 4.3.6. As can be seen, the detector made some mistakes. It did not recognize characters e, i, a, s, l and has false positive detections for tab and Q character at the bottom right corner. In addition, it incorrectly classified o as 0. Other than that, the detections are accurate. There can also be noticed sequence ?123 which stands for a keyboard mode changing key. It is up to the post-processing algorithm to find it.

Figure 4.15: Individual characters are detected in the keyboard using the YOLOv7 model.

Next, figures 4.16 and 4.17 show the post-processing phase. In figure 4.16 the algorithm computed the missing characters (cyan bounding boxes) and also managed to correct the o vs. 0 mistake. Moreover, no rule for the false positive tab and Q was matched so they were removed. Furthermore, the mode sequence was found and converted to a new key bounding box.

Figure 4.16: Character detections are corrected and missing characters are computed in the post-processing phase.

In figure 4.17 can be seen the final post-processing result including the Canny detections. It differs from the figure 4.16 in that it has the space key recognized. Moreover, the computed keys are not visually distinguished.

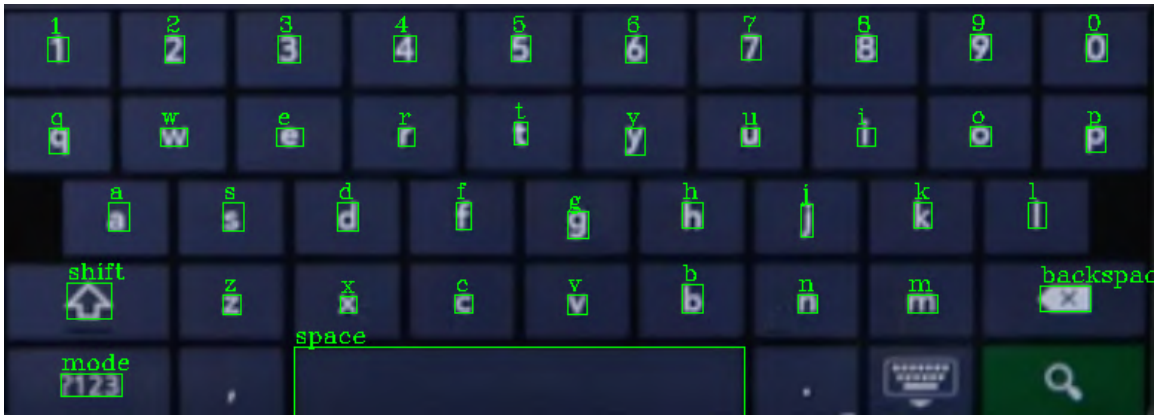


Figure 4.17: The post-processing is completed by incorporating the Canny detection results, namely the space key.

As already mentioned, the example image comes from the validation dataset. Therefore, to visually show the recognition accuracy, figure 4.18 illustrates the detected and expected keys. The green bounding boxes are the detected ones while the yellow show the expected coordinates. Almost all characters/keys were recognized and the overlap with the ground truth is very good. However, there can be noticed two undetected magenta bounding boxes for dot and comma characters. Finally, after transforming the detections back to the original image, figure 4.19 depicts the complete keyboard recognition result.

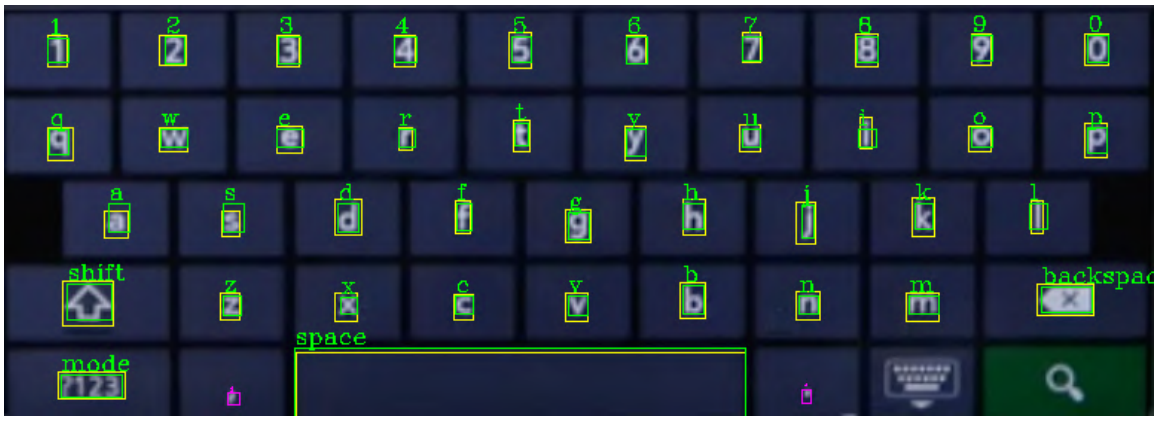


Figure 4.18: The detected keys (green) match very well the expected ones (yellow).

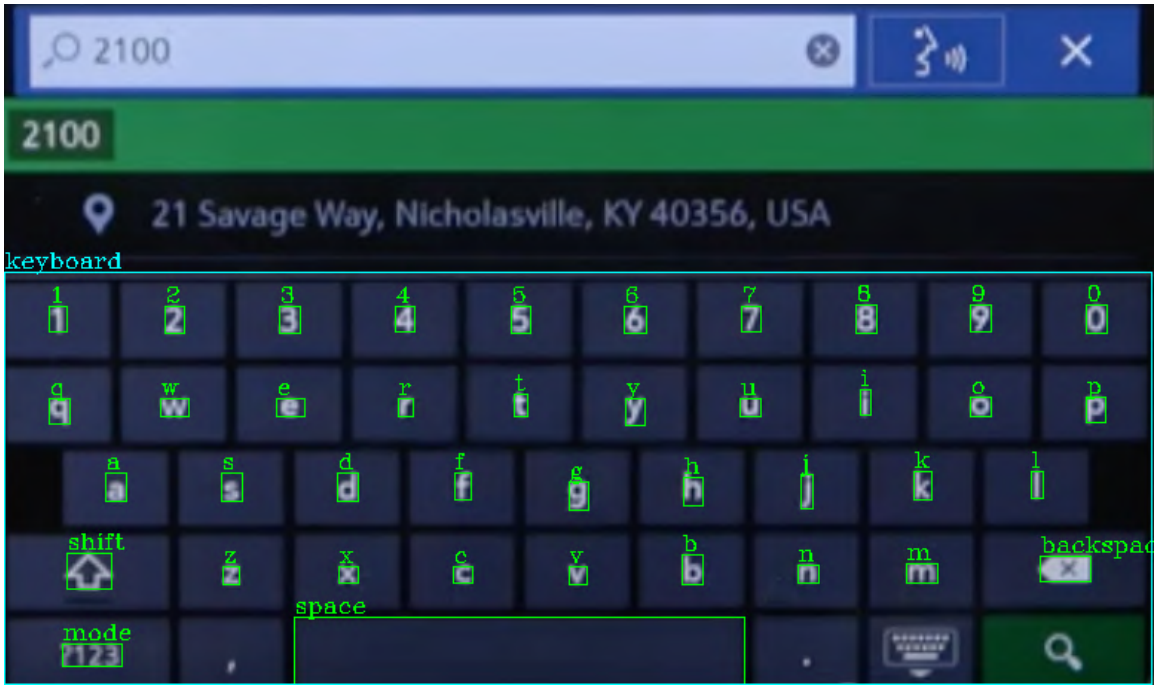


Figure 4.19: Fully processed image as a result of the image recognition

Chapter 5

Evaluation of results

In this chapter, individual recognition phases are evaluated. Some evaluation of YOLOv7 training strategies was already done in section 4.1.2. There were discussed differences between 16 vs. 32 batch size and finetuning vs. new training. Furthermore, three versions of 640x640 YOLOv7 models were introduced. The following sections 5.1 and 5.2 present the comparison of the YOLOv7 variations (yolov7-tiny, yolov7, yolov7-x) and discuss the choices made. The last section 5.3 shows results of the post-processing algorithm on its validation dataset. The metrics used for the evaluation are precision, recall and for the YOLOv7 models also mAP. The mAP was computed for 0.5 and 0.95 intersection-over-union values. A full account of the results is available on the attached medium A, here only selected metrics and results are used for demonstration.

5.1 Keyboard detector

As described in section 4.1.2, yolov7-tiny is smaller, faster and uses fewer resources than the original YOLOv7 variation from the paper [29] at the cost of accuracy. On the contrary, yolov7-x is supposed to be more accurate at the cost of reduced processing speed, higher resource consumption and bigger size. Figure 5.1 illustrates the comparison in resource consumption on averaged memory allocation in percents on a 2-core GPU provided by Kaggle platform. The relative difference between the models is very similar in GPU utilization and other physical metrics as well. Another factor is the actual size of the models. The tiny model has only about 12 KB, while the original one takes about 73 KB and the yolov7-x even 140 KB. So far, the tiny version is the winner. The comparison of mAP@.95 metric development during the training, which gives an account of the accuracy properties of the models, is shown in figure 5.2. Recall, precision and loss functions are available on the attached medium A, but as mAP is dependent upon recall and precision, it is a sufficient representation of the detector's overall capabilities. As can be seen, the training progressed very similarly and all 3 models came to the same results. This proves that for the keyboard detection task, the bigger models do not bring any additional value. Hence, the yolov7-tiny version is used as it is as accurate as the other versions while being significantly faster and requiring fewer computational resources. The full account of the training results on the testing dataset is in table 5.1. Apart from mAP@.95 the results are identical. There can be seen a slight improvement in mAP@.95 among the bigger models, but that can be viewed as insignificant. The near 2 % difference between yolov7-tiny and yolov7-x is negligible especially if taking into consideration the cost of size, speed and resources.

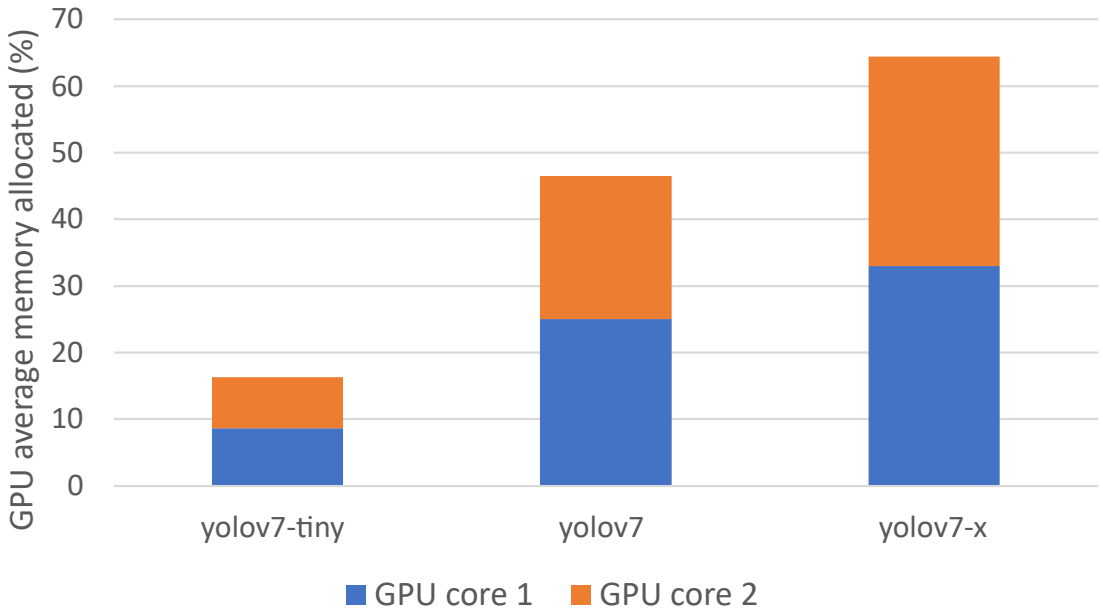


Figure 5.1: The tiny version of the YOLOv7 proves to be substantially more efficient in terms of resource consumption. Comparisons of other physical metrics look fairly similar.

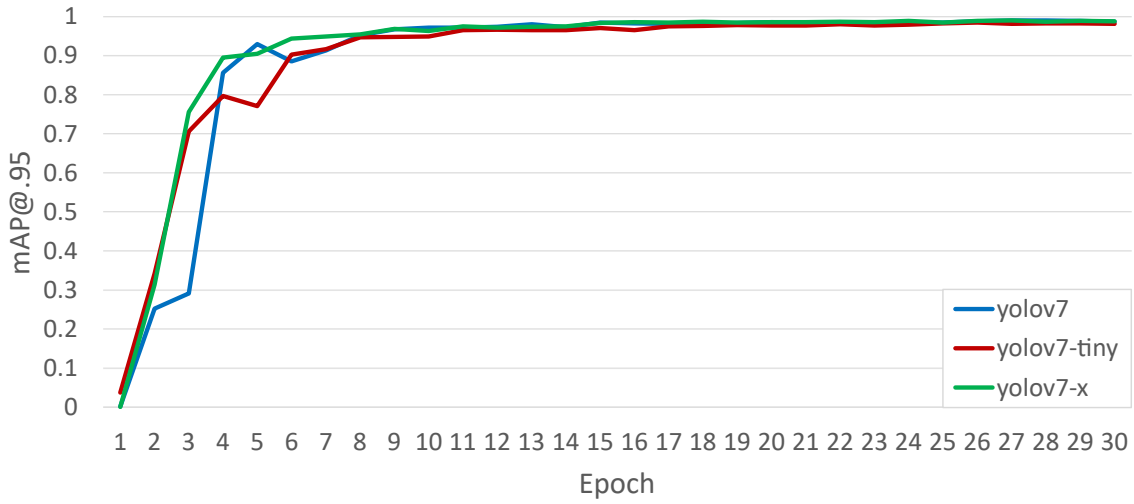


Figure 5.2: The development of mAP@.95 on validation data during training indicates a minimal difference in the expressive power of the models.

Model	Precision	Recall	mAP@.5	mAP@.95
yolov7-tiny	1	1	0.996	0.971
yolov7	1	1	0.997	0.98
yolov7-x	1	1	0.997	0.989

Table 5.1: The results of individual models on the testing keyboard dataset show that all of them are almost equally accurate.

5.2 Character detector

Due to using the same model as for the keyboard detection, the character detector was evaluated in the same manner. Concerning resource consumption, the difference between the models got reduced as can be seen in figure 5.3. The tiny model is still the clear winner, but not as notably as in keyboard detection shown in figure 5.1. This is caused by the increased number of classes and thus a greater number of detections made.

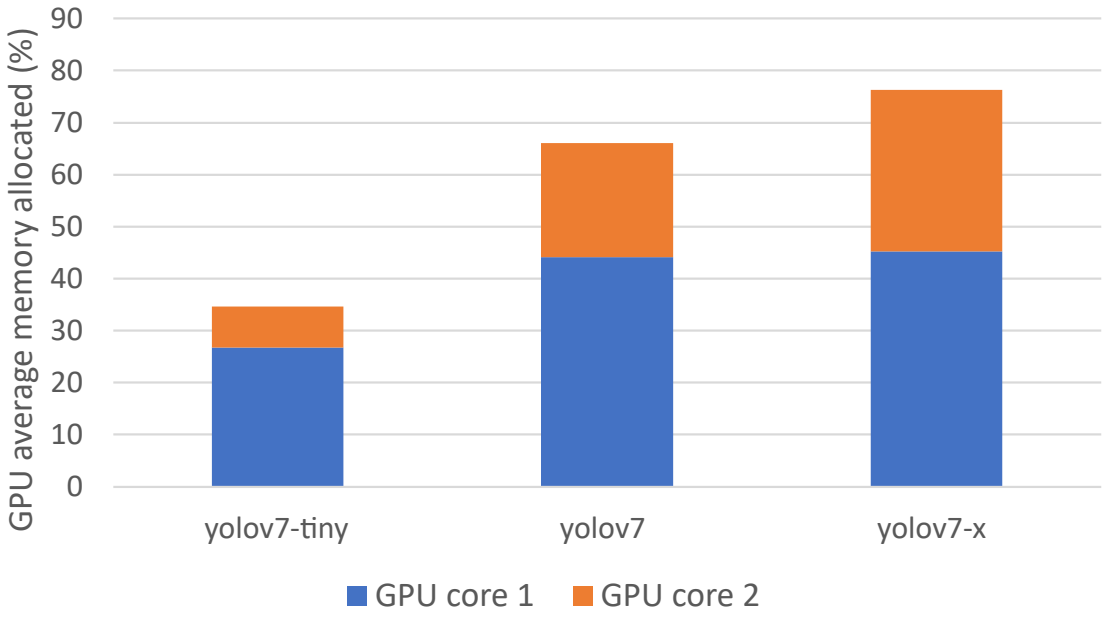


Figure 5.3: The resource consumption is a bit more leveled than in keyboard detection, but the fact that the bigger the model the higher the consumption still holds. Again, other metrics such as GPU utilization show similar differences.

When it comes to the expressive power of the models, the tiny model falls behind as demonstrated by figure 5.4. The bigger models are again almost identical, with the difference that yolov7-x was trained slightly smoother. The higher number of classes and presumably some similar-looking characters are problematic for the tiny model though. Nevertheless, it is still quite sure with itself as figure 5.5 shows. While it still cannot reach the same precision as the bigger models and it climbs slower, it eventually comes close. The problem lies in recall where the chart looks similar to figure 5.4. It simply cannot detect as many characters and the ones detected have lower confidence scores. The complete results are laid out in table 5.2. The yolov7-x version again does not bring any improvement so due to its size it is considered no more. Thus, the original YOLOv7 model is selected for the character recognition task. However, the next section 5.3 explains why the tiny model might still be of use. The presented data are overall results. Character-specific metrics are available on the attached medium A. To demonstrate the accuracy variability of individual characters, figure 5.6 shows their F1 curves. The lowest curves belong to special characters such as dot, comma or underscore while the top ones are for special key icons and capital alphabet characters. In conclusion, it is visible that the accuracy varies which was expected not only because of special characters but also non-discriminative upper and lower case letters.

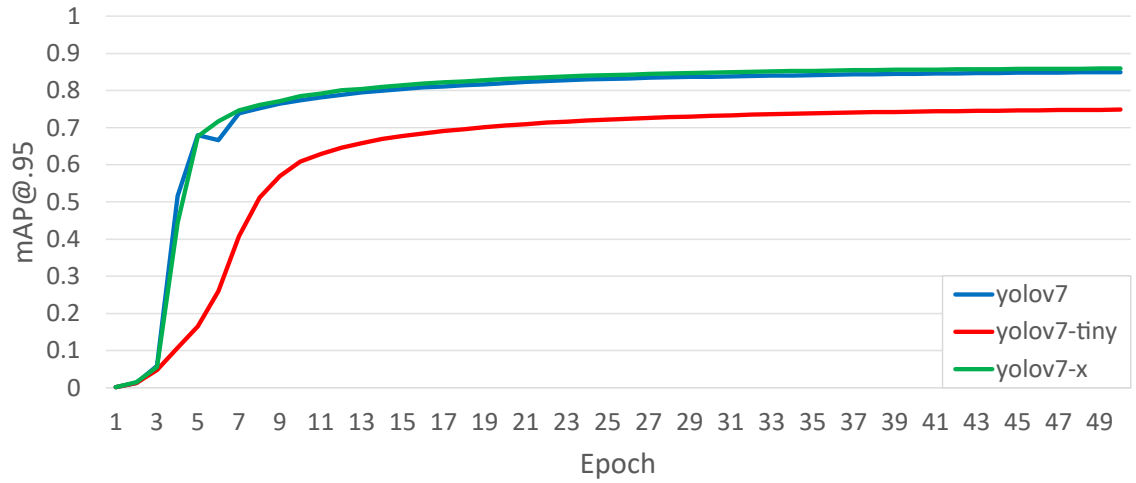


Figure 5.4: The development of mAP@.95 on validation data during training indicates that bigger yolov7-x is not stronger that the normal version. However, in comparison with the keyboard detector, the tiny version is not as accurate as the bigger models.

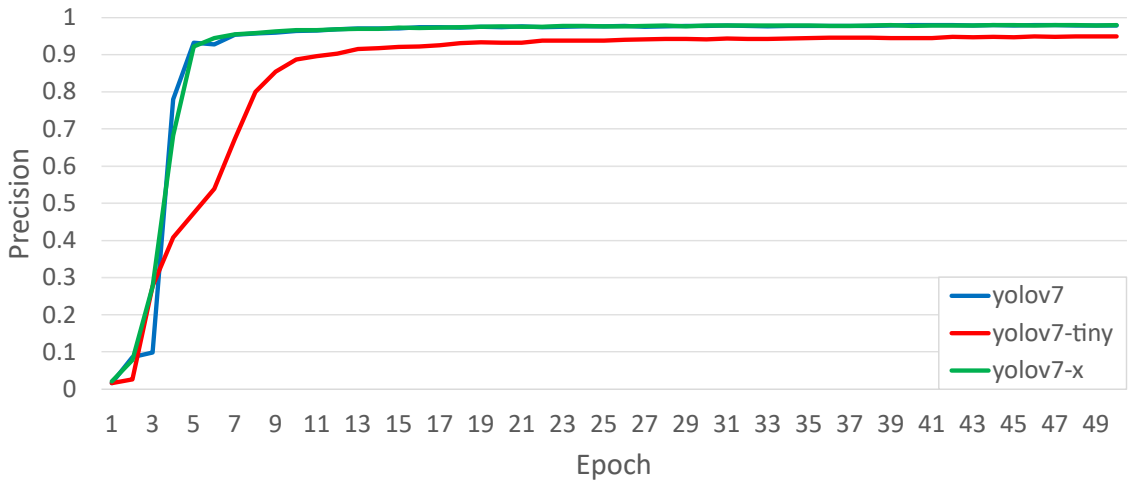


Figure 5.5: The development of precision on validation data during training shows that the bigger models have the same precision. The tiny version falls behind a bit but it is still competitive.

Model	Precision	Recall	mAP@.5	mAP@.95
yolov7-tiny	0.948	0.852	0.912	0.748
yolov7	0.979	0.951	0.977	0.848
yolov7-x	0.98	0.95	0.978	0.858

Table 5.2: The results of individual models on the testing character dataset show that while the tiny model is less accurate, especially in terms of recall and mAP@.95, the difference between the normal and bigger versions is minuscule.

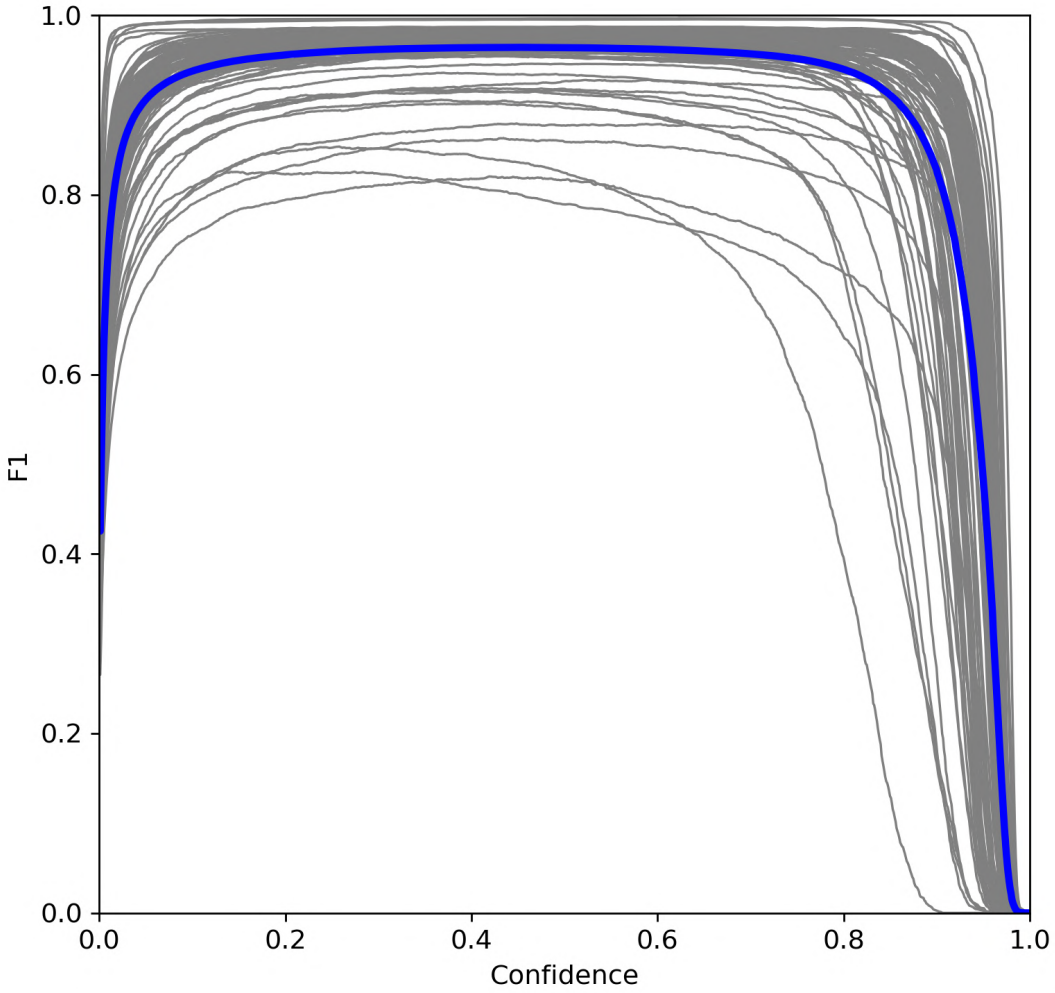


Figure 5.6: The majority of F1 curves for individual characters (gray) is better than the overall F1 curve (blue) from the original YOLOv7 model on character testing dataset. This is due to some less accurate special characters.

5.3 Post-processing algorithm

The character detector's results show that there is room for improvement. The post-processing algorithm not only improves overall recognition results but also removes the accuracy gap between the tiny and normal models. On the post-processing validation dataset, the character recognition with applied post-processing achieves similar though slightly worse results shown in table 5.3 than the single-character detection on the character dataset. Notwithstanding, the results are still impressive considering this is a real-world and not a generated dataset. Moreover, these are averaged numbers impaired with special character detection on which not much post-processing is done.

As the table shows, the results for special characters are nowhere near as good as for the other keys. Partially, it is due to dots vs. noise/commas/colons/semicolons or underscores vs. dashes etc. It may to a degree be also due to some low-quality validation images. As not many full-HD samples could be obtained, the dataset consists of keyboards

of diverse quality. It can be presumed that on full-HD images from AIVA cameras, the results might be slightly better. It is worth reminding, that special character recognition was not the main objective so for a bonus the results can still be considered satisfactory.

The special keys results are significantly better. Though they are not yet perfect, taking into account that these keys do not have to have just icons but also words on them, the results are favorable. The letters in words are usually much smaller than on regular keys, hence harder to recognize. Furthermore, because of the close proximity to other characters in the word, the detector can be easily baffled, especially in lower-quality images. Consequently, joining letters to words and recognizing the words in spite of missing (undetected or incorrectly detected) letters with such precision and recall is considered a success.

Despite special characters not doing so well, the results for target alphanumeric characters exceed expectations and show how powerful the post-processing really is. The reason for this is quite simple. Let's consider a qwerty layout. All the detector needs to recognize is one character in each row and at least a minimum-requirement number of characters in a row for the layout to be recognized as qwerty. This gives the algorithm a starting point and both x and y distances between characters. The rest can be computed or corrected with almost 100 % accuracy. As a consequence, the detector's responsibility is to provide just a bare minimum. This is also the reason why the tiny model actually performs slightly better. It does not produce as many false positives to confuse the algorithm as the normal version does and the undetected characters can be easily computed.

	yolov7		yolov7-tiny	
	Precision	Recall	Precision	Recall
All keys	0.942	0.949	0.964	0.942
Alphabet	1	1	1	1
Numbers	0.998	0.987	1	0.993
Alphanumeric	**0.999**	**0.997**	**1**	**0.998**
Special keys	0.965	0.910	0.987	0.914
Special characters	0.684	0.767	0.762	0.705

Table 5.3: Among the results of the post-processing algorithm on the validation dataset stand out alphanumeric characters.

Chapter 6

Conclusion

In this work, a research was done into popular object detection methods and a summary was made. Special attention was paid to the evolution of YOLO algorithm where each version until the most recent one at the time of writing the thesis was described. Particular detail was given to the current state-of-the-art YOLOv7. Moreover, datasets for keyboard detection, single-character detection and post-processing were devised and created. These datasets were used for training and validation of the recognition methods. The datasets are unique as no other is to the best of my knowledge publicly available. They can serve not only the keyboard typing automation task but also any other which might use keyboard detection or single-character detection. Then, a three-phased image processing strategy was designed and implemented. Firstly, YOLOv7 neural network model is used for keyboard detection. Secondly, characters are recognized in the detected keyboard region again using YOLOv7. The found characters symbolize potential keys and are further processed in the last stage. The post-processing algorithm corrects mistakes based on predefined supported layouts. Furthermore, it can be easily extended to handle new scenarios. The results of each phase were evaluated and found satisfactory. The accuracy of the methods exceeds expectations, particularly keyboard detection and recognition of alphanumeric characters. Consequently, a novel modular solution for keyboard and keys recognition was created which can be effortlessly extended or its parts replaced. This solution is to be used in the production environment of Y Soft AIVA system where it will help with automating robotic keyboard typing. On top of that, the thesis received Jiří Kunovský award from Excel@FIT 2023 conference based on votes from the professional public.

In future work, the actual integration into the AIVA system needs to be done. Other than that, there is room for improvement with respect to special characters' accuracy. Moreover, additional character sets can be supported such as other alphabets, diacritics or more special characters. The same holds for support of new keyboard layouts. Also, as new state-of-the-art detectors will be emerging, the models can get retrained to improve the overall results.

Bibliography

[1] Liu, Z. J., Gupta, R. K., Chen, K., Ferry, B. and Lacasse, S. A Scalable Computer Vision Framework For Mobile Device Auto-Typing. In: *2020 IEEE International Conference on Consumer Electronics (ICCE)*. 2020, p. 1–6. DOI: 10.1109/ICCE46568.2020.9043107.

[2] Liu, Z., Ferry, B. and Lacasse, S. A Deep Neural Network to Detect Keyboard Regions and Recognize Isolated Characters. In: *2019 International Conference on Document Analysis and Recognition Workshops (ICDARW)*. 2019, vol. 5, p. 140–145. DOI: 10.1109/ICDARW.2019.40095.

[3] Kyash. *Rx-keyboard-detector* [online]. GitHub, 2021. Available at: https://github.com/Kyash/rx-keyboard-detector.

[4] semmel. *On-screen-keyboard-detector* [online]. GitHub, 2021. Available at: https://github.com/semmel/on-screen-keyboard-detector.

[5] Baek, Y., Lee, B., Han, D., Yun, S. and Lee, H. Character Region Awareness for Text Detection. In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019, p. 9357–9366. DOI: 10.1109/CVPR.2019.00959.

[6] Canny, J. A Computational Approach To Edge Detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*. december 1986, PAMI-8, p. 679 – 698. DOI: 10.1109/TPAMI.1986.4767851.

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

[8] Yousefi, J. Image Binarization using Otsu Thresholding Algorithm. may 2015. DOI: 10.13140/RG.2.1.4758.9284.

[9] Uijlings, J., Sande, K., Gevers, T. and Smeulders, A. Selective Search for Object Recognition. *International Journal of Computer Vision*. september 2013, vol. 104, p. 154–171. DOI: 10.1007/s11263-013-0620-5.

[10] Gandhi, R. *R-CNN, Fast R-CNN, Faster R-CNN, YOLO — Object Detection Algorithms* [online]. July 2018 [cit. 2022-11-15]. Available at: https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e.

[11] Krizhevsky, A., Sutskever, I. and Hinton, G. ImageNet Classification with Deep Convolutional Neural Networks. *Neural Information Processing Systems*. january 2012, vol. 25. DOI: 10.1145/3065386.

[12] Girshick, R., Donahue, J., Darrell, T. and Malik, J. Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. november 2013. DOI: 10.1109/CVPR.2014.81.

[13] Evgeniou, T. and Pontil, M. Support Vector Machines: Theory and Applications. In: *Machine Learning and Its Applications: Advanced Lectures*. Springer Berlin Heidelberg, 2001, p. 249–257. DOI: 10.1007/3-540-44673-7_12. ISBN 978-3-540-44673-6.

[14] Li, F.-F., Wu, J. and Gao, R. *Lecture 9: Object Detection and Image Segmentation* [online], 26. april 2022 [cit. 2022-11-15]. Available at: http://cs231n.stanford.edu/slides/2022/lecture_9_jiajun.pdf.

[15] Girshick, R. Fast R-CNN. In: *2015 IEEE International Conference on Computer Vision (ICCV)*. 2015, p. 1440–1448. DOI: 10.1109/ICCV.2015.169.

[16] He, K., Zhang, X., Ren, S. and Sun, J. Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. june 2014, vol. 37. DOI: 10.1109/TPAMI.2015.2389824.

[17] Ren, S., He, K., Girshick, R. and Sun, J. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. june 2015, vol. 39. DOI: 10.1109/TPAMI.2016.2577031.

[18] Zafar, I., Tzanidou, G., Burton, R., Patel, N. and Araujo, L. *Hands-On Convolutional Neural Networks with TensorFlow*. Packt Publishing, 2018. ISBN 9781789130331.

[19] Redmon, J., Divvala, S., Girshick, R. and Farhadi, A. You Only Look Once: Unified, Real-Time Object Detection. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, p. 779–788. DOI: 10.1109/CVPR.2016.91.

[20] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S. et al. Going deeper with convolutions. In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015, p. 1–9. DOI: 10.1109/CVPR.2015.7298594.

[21] Ivanov, M. *The evolution of the YOLO neural networks family from v1 to v7.* [online]. October 2022 [cit. 2022-11-17]. Available at: https://medium.com/deelvin-machine-learning/the-evolution-of-the-yolo-neural-networks-family-from-v1-to-v7-48dd98702a3d.

[22] Redmon, J. and Farhadi, A. YOLO9000: Better, Faster, Stronger. In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, p. 6517–6525. DOI: 10.1109/CVPR.2017.690.

[23] Redmon, J. and Farhadi, A. YOLOv3: An Incremental Improvement. *CoRR*. april 2018. DOI: 10.48550/arXiv.1804.02767.

[24] Bochkovskiy, A., Wang, C.-Y. and Liao, H.-y. YOLOv4: Optimal Speed and Accuracy of Object Detection. *CoRR*. april 2020. DOI: 10.48550/arXiv.2004.10934.

[25] Thuan, D. *Evolution of Yolo algorithm and Yolov5: The State-of-the-Art object detention algorithm*. Bachelor's Thesis. Available at: https://www.theseus.fi/handle/10024/452552.

[26] Nelson, J. and Solawetz, J. *Responding to the Controversy about YOLOv5* [online]. June 2020 [cit. 2022-11-19]. Available at: https://blog.roboflow.com/yolov4-versus-yolov5/.

[27] Li, C., Li, L., Jiang, H., Weng, K., Geng, Y. et al. YOLOv6: A Single-Stage Object Detection Framework for Industrial Applications. september 2022. DOI: 10.48550/arXiv.2209.02976.

[28] Solawetz, J. and Nelson, J. *What's New in YOLOv6?* [online]. July 2022 [cit. 2022-11-19]. Available at: https://blog.roboflow.com/yolov6/.

[29] Wang, C.-Y., Bochkovskiy, A. and Liao, H.-y. YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors. *ArXiv preprint arXiv:2207.02696*. july 2022. DOI: 10.48550/arXiv.2207.02696.

[30] Boesch, G. *YOLOv7: The Most Powerful Object Detection Algorithm (2022 Guide)* [online]. 2022 [cit. 2022-11-19]. Available at: https://viso.ai/deep-learning/yolov7-guide/.

[31] Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S. et al. SSD: Single Shot MultiBox Detector. In: *Computer Vision – ECCV 2016*. Springer International Publishing, October 2016, p. 21–37. DOI: 10.1007/978-3-319-46448-0_2. ISBN 978-3-319-46447-3.

[32] Jong, T. A. de. *Moiré lattice generator*. Zenodo, august 2021. DOI: 10.5281/zenodo.5710707. Available at: https://doi.org/10.5281/zenodo.5710707.

[33] Long, X., Deng, K., Wang, G., Zhang, Y., Dang, Q. et al. *PP-YOLO: An Effective and Efficient Implementation of Object Detector*. July 2020. DOI: 10.48550/arXiv.2007.12099.

[34] Soudy, M., Afify, Y. and Badr, N. RepConv: A novel architecture for image scene classification on Intel scenes dataset. *International Journal of Intelligent Computing and Information Sciences*. april 2022, p. 1–11. DOI: 10.21608/ijicis.2022.118834.1163.

[35] Lee, C.-Y., Xie, S., Gallagher, P., Zhang, Z. and Tu, Z. Deeply-Supervised Nets. In: Lebanon, G. and Vishwanathan, S. V. N., ed. *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics*. San Diego, California, USA: PMLR, 09–12 May 2015, vol. 38, p. 562–570. Proceedings of Machine Learning Research.

[36] He, P., Huang, W., He, T., Zhu, Q., Qiao, Y. et al. Single Shot Text Detector with Regional Attention. In: *2017 IEEE International Conference on Computer Vision (ICCV)*. 2017, p. 3066–3074. DOI: 10.1109/ICCV.2017.331.

[37] Zuiderveld, K. VIII.5. Contrast Limited Adaptive Histogram Equalization. In: *Graphics Gems*. 1994, p. 474–485. DOI: 10.1016/b978-0-12-336156-1.50061-6.

Appendix A

Contents of the attached medium

- `/doc` — Solution documentation
  - `/latex` — LaTex source code for the thesis
  - `thesis.pdf` — This text of the thesis
- `/results` — Image processing results
  - `/models` — Trained models for keyboard and character detection
  - `/statistics` — Numeric and visual training and evaluation results
  - `training_metrics.xlsx` — Metrics for each training
- `/src` — Source code of the solution

Appendix B

Excel@FIT 2023 Poster

Keyboard and Keys Image Recognition

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Author: Jan Lorenc
Supervisor: Jan Pluskal

Excel@FIT 2023

About

The goal of this thesis is to create a working solution for keyboard keys recognition to automate robotic writing on keyboards. The work is split into separate keyboard detection, single-character detection and post-processing of the results. Each of these parts required individual datasets.

Conditions

Due to extending an existing system AIVA, some factors can be taken into consideration:

- Full-HD cameras
- Camera calibration system
- Lab/office environment

Technologies

- PyTorch + OpenCV
- YOLOv7
- Canny edge detection

Datasets

Keyboards

- 615 keyboards of different types from various devices
- Data augmentation ➞ 20 000 images
- Scene background (COCO17) or single color
- 1280x720 px

Figure 1: A keyboard is generated on a background of the same color as is its averaged background color

Characters

- 99 classes
- Data augmentation ➞ 50 000 images
- Grayscale (high contrast between keyboard background and characters)
- 640x640 px (prepared for YOLOv7)

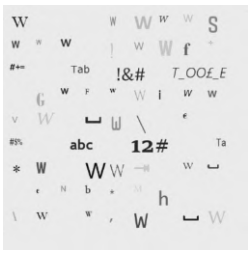


Figure 2: Main character W and other random characters generated for character detection training

Post-processing

- Validation dataset
- 120 manually annotated images
  - 60 for various layouts
  - 60 for missing character computation
- Any input image size (real-world examples)

Figure 3: An annotated smartphone keyboard with some missing characters expected to be computed

Image processing

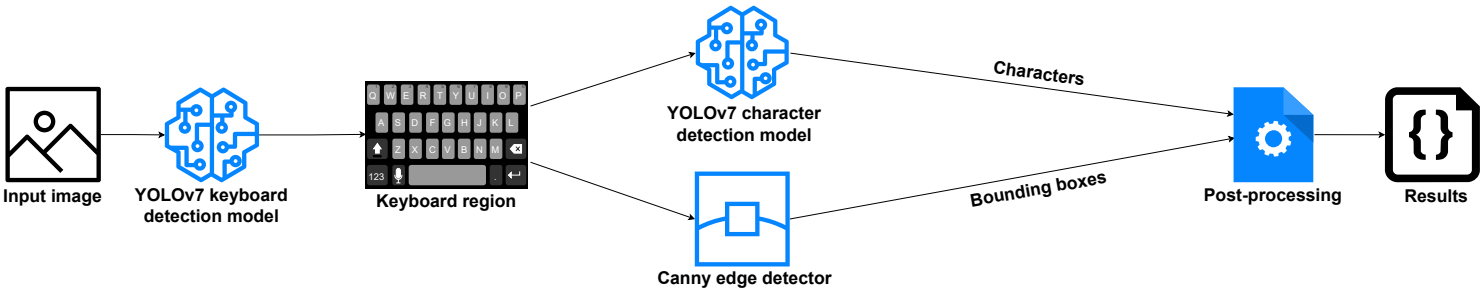


Figure 4: Flowchart of the recognition process

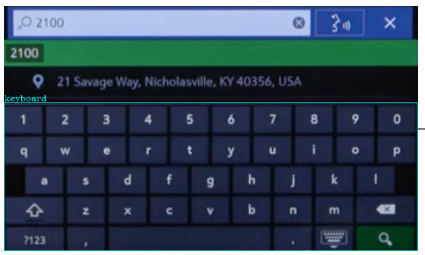


Figure 5: The first phase detects a keyboard region in an input image

Figure 6: The character recognition is run on the detected keyboard region. Here it can be seen that some characters were undetected and there are also two false positives.

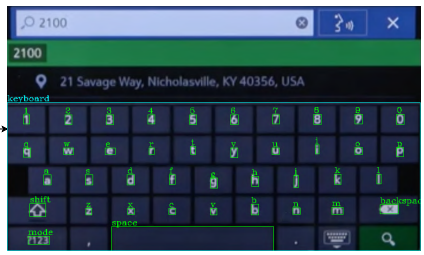


Figure 7: Final detection result after application of the post-processing algorithm. Missing characters were computed. False positives were removed. Special key "mode" was recognized and space was found thanks to the Canny edge detection.

Recognition results

Keyboard detection

Dataset	Precision	Recall	mAP@.5	mAP@.95
Validation	1	1	0.996	0.985
Test	1	1	0.996	0.971

Table 1: Results of selected tiny variation of the YOLOv7 model on the generated keyboard detection dataset

Character detection

Dataset	Precision	Recall	mAP@.5	mAP@.95
Validation	0.979	0.953	0.979	0.85
Test	0.979	0.951	0.977	0.848

Table 2: Results of the YOLOv7 model on the generated character detection dataset

Dataset	Precision	Recall	mAP@.5	mAP@.95
Validation	0.95	0.851	0.913	0.749
Test	0.948	0.852	0.912	0.748

Table 3: Results of tiny variation of the YOLOv7 model on the generated character detection dataset

Post-processing

	Normal model		Tiny model	
Charset	Precision	Recall	Precision	Recall
All	0.942	0.949	0.964	0.942
Alphabet	1	1	1	1
Numbers	0.998	0.987	1	0.993
Alphanumeric	**0.999**	**0.997**	**1**	**0.998**
Special keys	0.965	0.910	0.987	0.914
Special characters	0.684	0.767	0.762	0.705

Table 4: Results of post-processing on the validation dataset. Target alphanumeric characters outperform.

Appendix C

Excel@FIT 2023 Article

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2023
http://excel.fit.vutbr.cz

Keyboard and Keys Image Recognition

Jan Lorenc*

Abstract

The goal of this thesis is to create a solution for keyboard keys recognition to automate robotic writing on keyboards. Datasets for keyboard detection in an image, character detection in an image and post-processing correction of the character detection based on various keyboard layouts were created as prerequisits for this work. This research presents several approaches towards keyboard keys detection problem and selects the most suitable one. The chosen strategy is to split the problem into 3 phases which correspond to the prepared datasets. Firstly, a separate keyboard detection is run. Secondly, characters are recognized in the detected keyboard region. These tasks are accomplished using neural networks and Canny edge detection technique. The last phase is the post-processing of the detection results (character correction, autocompletion of undetected characters, special keys distinction etc.). The results of each phase are evaluated. The contribution of the thesis lies in the creation of the datasets for keyboard and keys detection, and novel modular and extensible solution for the recognition process that yields very promising results.

*xloren15@vut.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

The idea for this thesis comes from the company Y Soft which develops a robotic automation solution AIVA. The ultimate goal is to teach the robot to autonomously write on keyboards. To accomplish such a task, tools for keyboard and keys detection are required. As no other open-source project seems to exist, a custom solution has to be developed.

There are two tasks to be solved. The first one is keyboard recognition in an image. This enables the robot to decide if it can write and where. The second task is single-character detection where characters represent the keys. The objective is to recognize alphanumerical keys and some special keys. In addition, special character detection is attempted. For both tasks, training datasets had to be created.

On this topic, only one other solution could be found. Researchers from Amazon also tried to solve automated keyboard typing for their framework [1]. Their article [2] has been an inspiration, especially in the solution design. Unfortunately, neither the code nor the dataset seems to be publicly available.

The proposed solution is to split the image recognition process into 3 phases. Firstly, a keyboard region is detected using YOLOv7 [3] neural network, the current SOTA in object detection. Secondly, characters are recognized in the detected keyboard region using YOLOv7 and Canny edge detection technique. Lastly, the character detection results are processed to make corrections.

The main contribution of the work is the improvement of the production solution Y Soft AIVA. It greatly simplifies the current keyboard typing process and reduces the time for automation script definitions. Furthermore, it provides datasets for keyboard and character object detection to the public. Moreover, it is to the best of my knowledge the only open-source solution of its kind.

2. Datasets creation

No available keyboard object detection dataset could be found. Therefore, custom datasets needed to be prepared. In total, 3 datasets were created. These are for training keyboard and single-character detectors and validating post-processing correction algorithms.

Concerning the keyboards dataset, 615 keyboards of different types from various devices were collected. These were data augmented and generated to random backgrounds. The augmentation methods used were

scaling, blurring, brightness changing and the addition of various noises, transparency and moiré effects.

When it comes to the characters, the dataset is in grayscale. Randomly gray-colored characters and backgrounds to put them on were generated with a constraint of at least a 24-pixel intensity difference between them. This can be done because keys on keyboards are very contrastive and moving to grayscale removes any color design effects. The augmentation methods used were the same as for keyboards.

Concerning the post-processing validation dataset, keys on 120 selected keyboards with different layouts were annotated. The goal is to check if correct characters were recognized, redundant characters removed or missing characters computed.

3. Solution design

For the keyboard detection model was used *tiny* version of YOLOv7 neural network architecture. It achieved the same accuracy as the *standard* version while using fewer resources and it is significantly faster. It was trained for 30 epochs with batch size 16 and it takes 640x640 input images. It can accept any size but is scaled or padded to this resolution.

The input for the character detection is the output of the keyboard detection model. On the detected keyboard region is simultaneously run YOLOv7 model trained for characters and Canny edge detection. The reason for this is that the character detector cannot recognize e.g. space key when it is blank. The Canny detector provides supplementary results for the subsequent post-processing. The neural network was in comparison to the keyboard detector trained for 50 epochs due to having 99 classes instead of 1, so it takes longer. Both *tiny* and *standard* models were trained and this time the difference was significant. On the other hand, the post-processing results demonstrate why *tiny* version might be sufficient.

The goal of the post-processing algorithm is to recognize the layout and fix any incorrections. Undetected characters can be computed, letter case for characters such as xX, oO etc. corrected or special key keywords found. This further improves the recognition results.

4. Recognition results

The keyboard detection was a huge success. Even the *tiny* model achieves 100 % recall and precision and over 97 % mAP@.95 for both validation and testing data. Concerning the character detection, the numbers on the single-character test dataset are lower as table 1 depicts. Nevertheless, the results of the *standard* model are still nice considering the inclusion of special characters such as dots and commas. The *tiny* version fares much worse and while it is quite sure with its results with relatively high precision, the lower recall says it cannot find a lot of characters.

	Precision	Recall	mAP@.95
Keyboards (tiny)	1	1	0.97
Characters (tiny)	0.948	0.852	0.748
Characters (yolov7)	0.979	0.951	0.848

Table 1. Trained model results on test datasets

The post-processing algorithm, however, improves the character detection results and also removes the gap between *tiny* and *standard* models. On the post-processing validation dataset, the character recognition with applied post-processing achieves similar though slightly worse results shown in table 2 than the single-character detection on the character dataset. Notwithstanding, the results are still great considering this is a real-world and not a generated dataset. Moreover, these are averaged numbers impaired with special character detection on which not much post-processing is done. If focused only on the target alphanumeric characters, both precision and recall are incredibly good. What is more, the worse recall of the *tiny* model actually helps the post-processing as it does not handle many false positives and it leads to even better results. This demonstrates the power of the post-processing algorithm to correct detections and compute missing characters based on predefined layouts such as qwerty.

	Precision	Recall
All keys (yolov7)	0.942	0.949
All keys (tiny)	0.964	0.942
Alphanumeric (yolov7)	0.999	0.997
Alphanumeric (tiny)	1	0.998

Table 2. Post-processing algorithm results on post-processing validation dataset

5. Conclusions

The thesis offers a working modular solution for keyboard and keys image recognition problems. Any model can be easily retrained and switched to a different one. Similarly, additional post-processing techniques to cover other special cases can be included. The achieved results exceed expectations and in the future, support for more character sets can be added.

Acknowledgements

I would like to thank my supervisor Jan Pluskal for his help and Y Soft for making the thesis possible.

References

[1] Zongyi Liu, R.K. Gupta, Kun Chen, Bruce Ferry, and Simon Lacasse. A scalable computer vision framework for mobile device auto-typing. 01 2020.

[2] Zongyi Liu, Bruce Ferry, and Simon Lacasse. A deep neural network to detect keyboard regions and recognize isolated characters. 09 2019.

[3] Chien-Yao Wang, Alexey Bochkovskiy, and Hong-yuan Liao. Yolov7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors. 07 2022.