Handwritten digital card recognition and grasping program

By

Chenfu Zhang

An Honours report submitted for the Degree of:

Bachelor of Engineering in Robotics

April 2024

Synopsis

At present, in large factories, it is very common to use programmable manipulators to complete some specific tasks. However, in recent years the rise of the wave of home intelligence, for example, the function of many personal equipment has not reached the ideal level in people's minds. This is not to say that the existing technology is difficult to satisfy people, but that the cheap or affordable devices are unable to achieve some of the features that are now widely used on high-end devices. Since some functions can already be implemented on some high-end devices, it is necessary for engineers to try their best to improve the performance of the implemented functions, and on the other hand, they should also pay attention to how to apply those advanced functions to devices with relatively weak physical functions.

The purpose of this experiment is to implement some functions and collect the data generated in the process on a small, embedded device using common programming methods, using high-level programming languages, without considering the underlying optimizations. Specifically, this experiment will develop some simple scripts using small robotic arms and USB cameras and an AI development board to achieve simple image processing tasks, complex image processing tasks, and the task of grabbing objects in the camera, respectively. Complex image processing is the task of classifying images using convolutional neural networks. The simple image processing task is to preprocess the image to reduce its impact on the convolutional neural network prediction process.

This experiment uses the Jetson Nano development board to control a small robotic arm and uses this motherboard to receive and process signals from a USB camera. The Jetson Nano is a small embedded artificial intelligence computer from NVIDIA with a built-in Linux operating system.

Python is mainly used as the programming language in this experiment. OpenCV-Python can realize the function of taking images and videos and processing images and video data with Python language, image processing and computer vision tasks, and can complete the simple image processing function of this experiment. This toolkit is powerful and easy to use, which greatly reduces the learning cost of engineers and improves the efficiency of engineers to conduct experiments. In this experiment, Pytorch framework is used to implement convolutional neural network. Written in Python, it is a toolkit that integrates features related to convolutional neural networks and implements several popular convolutional neural network frameworks.

This experiment uses ROS-Python to control the signal from the camera. ROS-Python is an interface for a Robot Operating System based on the Python programming language. ROS is an open source, modular robotics software platform for building various types of robotics applications. This also reduces the learning cost of this experiment and improves the efficiency.

This experiment will provide necessary explanations of the algorithms used and offer justifications for their selection. Furthermore, the process of implementing these algorithms will be documented along with their performance under different parameters. Ultimately, the execution status of partial scripts implementing functionalities on the Jetson Nano development board will be recorded, with the same algorithms run on a personal computer, a hardware platform with more robust capabilities, for comparison and ensuring their generalizability. Conclusions and discussions regarding these results will be provided.

Acknowledgements

I would like to express my sincere gratitude to Dr. Keith Edgar Brown for his invaluable guidance, support, and mentorship throughout the course of this research. His expertise and insights have been instrumental in shaping the direction of this work.

Statement of Authorship

I, Chenfu Zhang

State that this work submitted for assessment is my own and expressed in my own words. Any uses made within it of works of other authors in any form (eg. Ideas, figures, text, tables) are properly acknowledged at their point of use. A list of the references employed is included.

Date ……9/4/2024……...….

Index

Introduction

1.1 Objective

This experiment will involve designing a program on the Jetson Nano development board to manipulate a small robotic arm for processing data captured by a USB camera and achieving object grasping and classification within the camera's field of view. Depending on the functionalities implemented, different devices will be selected to meet the desired objectives. Throughout the experiment, parameters chosen in the main program and their effects on functionality will be recorded, alongside the time required for the execution of complex functionalities. Simple props will be manually crafted for the experiment; however, the core focus of this experiment does not revolve around studying the performance of the robotic arm in object grasping. Therefore, limited description will be provided on this aspect in subsequent sections. Overall, this experiment aims to provide valuable data, showcasing the challenges encountered by an ordinary developer in implementing functionalities on small-scale devices and the operational efficiency of the functionalities achieved.

1.2 Experiment process and result evaluation

Firstly, the convolutional neural network model will be trained. A PyTorch framework will be used to construct the network architecture, and the NVIDIA GPU installed in the personal computer will be leveraged as the training device instead of the CPU. The Jetson Nano is a development board produced by NVIDIA, equipped with a discrete graphics card and NVIDIA graphics driver installed. Hence, the same functionalities can be invoked on both the personal computer and the development board. Throughout the experiment, basic consistent code will be selected and executed on different devices to ensure the code's universality, considering the potential impact of different PyTorch versions and graphics driver versions. The specific configurations of the

devices will be detailed in subsequent sections. To expedite the experiment's progress, the model training task will be conducted on the personal computer. This approach accelerates the experiment's pace and circumvents the computational power and memory space limitations typically encountered by most small-scale devices, which often utilize pre-trained and optimized models. The experiment will refrain from optimizing the model, given its complexity spanning multiple domains. However, discussions regarding the obtained results will be provided. The experiment will employ a convolutional neural network for image classification, with the program required to achieve an accuracy rate of over 97% when predicting images in the training dataset post-training. The training and testing phases of the convolutional neural network will be divided into multiple scripts, alongside scripts developed to showcase the dataset contents. In this experiment, a neural network model for recognizing individual handwritten digits will be constructed, employing the LeNet network architecture. Subsequent chapters will elucidate the rationale behind this choice and present the program's execution results.

Secondly, the functionality for preprocessing general images will be implemented. The images in the dataset utilized for this experiment exhibit a relatively uniform format, which does not necessarily correspond to real-world scenarios. Therefore, specialized preprocessing functionality-related code needs to be developed to optimize the image format. Apart from the OpenCV-Python tool, the Pillow-Python toolkit will also be introduced to convert images into formats more suitable for Python processing. The program will ensure the capability to locate the pixel centre coordinates of objects to be grasped within the images and extract the corresponding image information for convolutional neural network prediction.

Thirdly, scripts will be developed to capture signals from the camera. The Jetson Nano is an embedded development board that does not inherently support the simple USB

camera used in the project. However, the device manufacturer has completed the interface development, utilizing ROS-Python as the tool for invoking relevant functionalities. It is worth emphasizing that ROS currently has two versions, and this experiment adopts the first version (ROS1), with specific reasons to be elaborated in the next section. During the experiment, it will be necessary to adhere to the general naming conventions of ROS1 to create relevant folders and files and run programs in the specified environment. Ultimately, the signals received will be converted into image formats compatible with OpenCV-Python, and the camera's feed will be displayed in video format through a window, enabling the saving of one frame for processing and analysis. This functionality will be integrated as a module into subsequent development tasks.

Fourthly, the functionality to manipulate the robotic arm for grasping objects from the camera feed will be implemented. The robotic arm will be controlled using Python code provided by the device manufacturer, bypassing the need for ROS tools. However, the parameters for controlling the robotic arm's movement are not directly correlated with the central pixel coordinates returned from the image preprocessing. To achieve this functionality, it will be necessary to set up the environment in advance, manually adjust the parameters in the code by grasping different targets from the camera feed one by one, and derive a transformation formula for the final coordination. Due to inevitable errors, the coordinate transformation method obtained from experimentation may not be entirely reliable, necessitating further parameter adjustments. The results of the experiments, along with the experimentation process, will be provided in subsequent chapters.

Fifthly, the pre-processed images will be subjected to prediction using the trained neural network model. During the experiment, once the model's accuracy meets the expectations, it will be further tested using image data retrieved from the USB camera

feed, with detailed results elaborated in subsequent chapters. Ultimately, based on the experimental findings, corresponding objects for grasping will be produced. It is anticipated that recognition errors may occur during practical operations. In the subsequent chapters, explanations and interpretations will be provided based on the experimental results, comparison with the training dataset, and integration of relevant domain knowledge. The model is expected to correctly identify the category of the created objects under normal circumstances. Additionally, efforts will be made to elucidate why the fabricated objects can be accurately recognized, incorporating analysis derived from this experiment. Given that the primary focus of this experiment is not on investigating convolutional neural networks, the explanations will primarily originate from this experiment and will not include conjectures or conclusions from other researchers.

Finally, the code relevant to the ultimate functionality of the program will be integrated. The script for training the convolutional neural network will no longer run on the Jetson Nano. The functionalities of reading and converting the camera signal and controlling the robotic arm to grasp specified objects will be merged, with one frame of the image saved to a designated folder. Firstly, the images mentioned earlier will undergo preprocessing, followed by prediction by the pre-trained convolutional neural network model. The ultimate goal is for the robotic arm to accurately grasp objects from any position within the camera's field of view. The saved images should be in the PNG format. The script with recognition and classification capabilities should be able to correctly predict the images saved during the program's execution. Image processing operations performed by the program should not and must not degrade the original images. The program should record the time required for the execution of certain functionalities.

1.3 Discussion of applications

Most software designed to run on specific devices is optimized based on the hardware characteristics of the installed platform. This experiment observes hardware performance from the perspective of an ordinary developer, coding the program using general programming techniques and documenting its runtime effects. This approach aims to provide optimization directions for the device and to some extent reflect its real-world performance. Additionally, it may reveal any issues with the programming tools used in the manuscript, such as the subtle differences in PyTorch across different systems.

In the field of deep learning, PyTorch has emerged as a highly popular tool. The code style used in this paper is consistent and well-commented, making it easily understandable for researchers in this domain. For researchers seeking to develop a convolutional neural network (CNN) model using PyTorch, the code utilized in this project serves as an excellent example. Additionally, due to the inherent advantages of CNNs in handling image data tasks, researchers in this area should also possess some knowledge of image processing. This project employs OpenCV-Python as a versatile image processing tool, accompanied by comments, which can facilitate researchers in quickly getting acquainted with this convenient tool, rather than using traditional tools like MATLAB or others.

Background

Given the project's strong emphasis on practicality, selecting a robot operating system (ROS) capable of running on different heterogeneous computers stands out as the primary concern. The Robot Operating System (ROS) is an open-source robot operating system. It functions not as a conventional system for scheduling and controlling processes but rather provides a structured information layer to the main computer operating system [14]. However, the creation of many codes within specialized frameworks tailored to achieve specific objectives has led to some inconvenience. The emergence of ROS has enhanced the reusability of code with similar logic. Developed in 2009, ROS introduced its second version, ROS2, in 2022. ROS2 places emphasis on security, reliability, and adaptability to more complex tasks and algorithm requirements in non-traditional scenarios [13]. Despite the ongoing development of ROS2, its widespread adoption is not yet prevalent. Therefore, this experiment will continue to utilize ROS1 for controlling the robotic arm.

In the early stages of research on recognition problems, license plate recognition holds significant representativeness and shares considerable similarity with the current project. This similarity arises from the fact that license plates predominantly feature alphanumeric characters, resulting in recognition difficulties akin to those encountered in pure numerical recognition tasks. The concept proposed by Tran Duc Duan et al., where manual analysis and selection of target features within images are conducted, followed by the design of an algorithm for recognition [5], serves as a significant inspiration for this study. Additionally, research by Yuval Netzer et al. (2011) provides two variants of unsupervised learning methods at the time, used for digit recognition in natural images without employing convolutional neural networks [12]. Leveraging contemporary technology, it may be possible to accomplish recognition tasks through simple code implementations. Furthermore, the computational capabilities of current

computers are sufficient for conducting feature computations on such straightforward objects, significantly reducing inefficiencies resulting from simplistic or outdated logic. Christos Nikolaos E. Anagnostopoulos et al. utilized character recognition neural networks and significantly enhanced recognition accuracy through the application of new algorithms [1].

In recent years, Keiron O'Shea and Ryan Nash contend that artificial neural networks have greatly transformed the field of machine learning. Among them, convolutional neural networks (CNNs) excel in solving complex image recognition tasks. By leveraging specific input information rather than focusing on the entirety of a domain, it is possible to construct a simple network to achieve outstanding performance [4]. Additionally, Waseem Rawat and Zenghui Wang posit that the advancement in modern computer computational capabilities has significantly contributed to the widespread application of convolutional neural networks [6]. Yann LeCun et al., in their overview of deep learning, mention that deep convolutional neural networks have made breakthroughs in the field of image recognition [7].

Additionally, Yann LeCun et al., in their research on the application of gradient learning in document recognition, noted that convolutional neural networks outperform other gradient-based learning techniques in recognizing handwritten characters when trained with the backpropagation algorithm on multi-layer neural networks [9]. Saqib Ali et al.'s study presents a method to enhance the efficiency of handwritten digit recognition. They employed a convolutional neural network as a classifier, utilized the MNIST dataset, and employed DL4J as the digit recognition framework, resulting in a recognition accuracy of 99.21% in one of their experimental environments, while significantly reducing training and testing times [3].

Some researchers have conducted tests on several popular models. Neha Sharma et al.

tested popular convolutional neural networks, namely AlexNet, GoogLeNet, and ResNet50, for object recognition in real-time video streams. The conclusion mentions that convolutional neural networks can be easily integrated into various platforms and widely applied [2]. An important finding: the authors believe that training these networks requires certain hardware performance and is not suitable for training on general-purpose devices.

Convolutional neural networks (CNNs) can sometimes be complex, and for this project, employing simpler network architectures can enhance efficiency. Andrew G. Howard et al. suggest that the current trend in CNN research is to construct deeper and more complex networks to improve recognition accuracy. Similarly, Ian J. Goodfellow et al. used deep convolutional neural networks in a 2014 experiment to recognize digits in street view images, finding that increasing the depth of the CNNs led to improved accuracy, with the best performance corresponding to an 11-layer network [11]. However, in certain applications, particularly in robotics and autonomous driving technology, recognition-related tasks need to be performed under limited computational resources. Andrew G. Howard et al. introduced the MobileNets model, which achieves smaller and faster models by refining algorithms [10].

With the widespread adoption of convolutional neural networks (CNNs), there is no shortage of researchers writing articles geared towards practical applications. Matthew Y.W. Teow authored a paper introducing a small-scale convolutional neural network for recognizing handwritten digits, aimed at beginners and non-mathematical experts [15]. This provides clear guidelines for novice programmers interested in developing programs using CNNs to tackle recognition tasks. Sebastian Gerke et al. utilized convolutional neural networks to recognize soccer jersey numbers, employing two encoding schemes for the jersey number vectors and studying their performance [8]. This serves as another exemplary case.

Experiment environment

3.1 Hardware parameters

This section will introduce the important parameters of the important hardware equipment used in this experiment. The important hardware devices mentioned above include the Jetson Nano development board, the CPU and GPU used to train the convolutional neural network model, and so on. Unimportant hardware devices include displays and other devices that do not have a significant impact on the achievement of the objectives of this experiment.

3.1.1 Jetson Nano parameters

Table1 Jetson Nano parameters

AI Performance	472 GFLOPS
GPU	128-core NVIDIA Maxwell™ architecture GPU
GPU Max Frequency	921MHz
CPU	Quad-Core Arm® Cortex®-A57 MPCore processor
CPU Max Frequency	1.43GHz
Memory	4GB 64-bit LPDDR4 25.6GB/s”
Storage	16GB eMMC 5.1
Video Encode	1x 4K30 (H.265) 2x 1080p60 (H.265)
Video Decode	1x 4K60 (H.265) 4x 1080p60 (H.265)
CSI Camera	Up to 4 cameras 12 lanes MIPI CSI-2 D-PHY 1.1 (up to 18 Gbps)

USB	1x USB 3.0 (5 Gbps) 3x USB 2.0
Display	2 multi-mode DP 1.2/eDP 1.4/HDMI 2.0 1 x2 DSI (1.5Gbps/lane)
Other IO	3x UART, 2x SPI, 2x I2S, 4x I2C, GPIOs
Power	5W – 10W
Mechanical	69.6mm x 45mm 260-pin SO-DIMM connector

In the experiment, because there is no need to use the Jetson Nano development board for the training of convolutional neural network models, the power is generally only 5W. If the Jetson Nano is operated at 10W for too long, it will stop working, and the help of a cooling device such as a fan will be required (this is not done considering that the power experiment may cause irreversible damage to the device).

3.1.2 PC chip parameters of CPU

Table 2 CPU parameters

Product Collection	13th Gen Intel ® Core ™ i7 processor
Processor number	i7-13700H
Vertical markets	Mobile
Number of cores	14
Number of threads	20
cache	24 MB Intel® Smart Cache
Processor base power consumption	45W

3.1.3 PC chip parameters of GPU

Table 3 GPU parameters

	GeForce RTX 4050 Laptop GPU
NVIDIA CUDA Core	2560

Acceleration frequency	1605 - 2370 MHz
Video memory capacity	6 GB
The type of video memory	GDDR 6

3.2 Operating system environment

This section describes the software environment required to run the programs involved in the experiment. The code for this experiment will run on Windows and Linux. Some of the code can be run on two operating systems at the same time, while some of the code will only run on Linux due to different environments.

3.2.1 Windows environment

Table 4 Windows environment

Windows	11
Python	3.8
Pytorch	1.10.1
Cudatookit	11.3.1
CuDNN	8.2
Torchsummary	1.5.1
Numpy	1.23.2
Pandas	1.3.4
Matplotlib	3.5.0
Sklearn	0.0
Pillow	10.0.1
OpenCV	4.1.2.30

3.2.2 Linux environment

Table 5 Linux environment

Linux	Ubuntu 18.04

ROS	Melodic
Python	3.6.9
Pytorch	1.8.0
Cudatookit	10.2
CuDNN	8.2
torchvision	0.9.0
Pandas	1.1.5
Matplotlib	3.3.4
Pillow	8.3.2
OpenCV	4.5.3
Numpy	1.19.4

3.2.3 Illustrate

Windows 11 is not necessarily the choice of Windows system; Windows 10 can also run programs. The configuration of the software environment related to the NVIDIA graphics card driver, as well as the version of the software package related to "CUDA", will have a subtle impact on the convolutional neural network model, please pay attention to the version number and its compatibility when using. The Linux system has the only corresponding ROS system, please strictly follow the environment configuration in this document. The return values of some functions are different depending on the OpenCV version. Other Python packages and related toolkits can be used in newer versions, and there is a high probability that they will be able to be used normally. However, it should be noted that older versions of Python packages and related toolkits tend to be used and tested by more people, making it easier to get help from the Internet when you encounter difficulties.

Experimental process

In the appendix of this document, a link to the remote repository containing the code used in this experiment will be provided.

4.1 Build model

4.1.1 Data set

The MNIST dataset is a commonly used dataset in machine learning for handwritten digit recognition tasks. It consists of samples of handwritten digits from 0 to 9, with each sample being a grayscale image of size 28x28 pixels. The dataset comprises 60,000 training samples and 10,000 testing samples in total.

The images in the MNIST dataset are stored as grayscale images, and the images in the dataset are in the IDX-3 format, which is a type of image file format. Additionally, these images are compressed using gzip, so popular software cannot directly convert them into images for visualization. Therefore, in this experiment, a script named `plot.py` (located in the `Windows-code` directory) is written to display the images in the dataset.

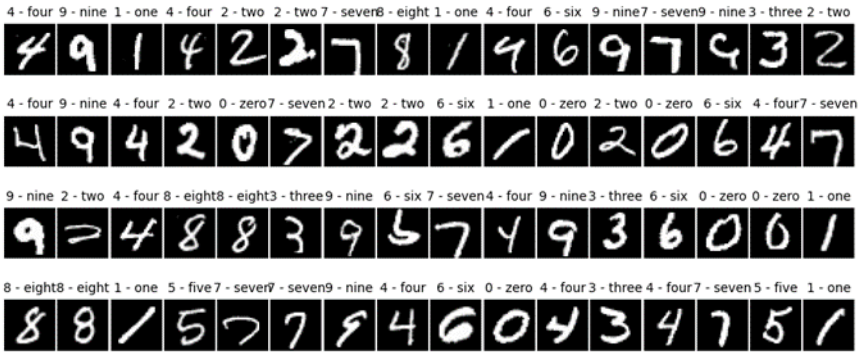

Figure 1 Pictures in MNSIT

In the code, there exists a folder named "MNIST" within another folder named "data" (this dataset can be found in a remote repository), which may seem redundant. However, MNIST is a widely utilized dataset, and due to its antiquity, there are

relatively few code interfaces available for processing it. At the outset of the program, a specialized interface for handling this type of data is employed, which batches 64 images together into one batch and shuffles their order before storage. Subsequently, the data from the first batch (saved as tensors representing images) is converted into a common image storage format for display. In PyTorch, tensors are multidimensional arrays akin to arrays in NumPy. They serve as the fundamental data structure in PyTorch for representing data and performing numerical computations. PyTorch supports tensor operations on both CPU and GPU, allowing tensors to be stored in either CPU or GPU memory.

In this section, Python was utilized to present an exposition of a non-modern traditional dataset, with tensors employed as the fundamental image format for training convolutional neural networks within the PyTorch framework. Tensors, akin to arrays but more intricate, do not necessitate a thorough comprehension of their underlying structure for effective utilization. A lack of clear understanding of the data types utilized during the experimental process is deemed methodologically unsound. However, due to time constraints, dedicated investigation into tensors as a data type will be deferred to future experiments. In subsequent experimental endeavours, greater consideration will be given to similar issues during the experimental planning phase.

4.1.2 Build a model

This experiment adopts the LeNet convolutional neural network model. The choice of this network model is twofold: first, it is entirely capable of handling simple tasks such as handwritten digit recognition, and second, it is sufficiently simple and fast in execution. In PyTorch, constructing a model involves inheriting from the `nn.Module` class and defining initialization and forward propagation methods. (This section utilizes the `model.py` file from the `Windows-code` folder in the remote repository.)

The experiment mainly adopts the network structure of LeNet. The activation function chosen is the Sigmoid function, and the pooling method chosen is average pooling. The first convolutional layer uses a 5x5 kernel size with padding set to 2 and 6 output channels. The second convolution uses a 5x5 kernel size without padding and has 16 output channels. After flattening, the model passes through three linear fully connected layers, ultimately outputting 10 results corresponding to the 10 digits. The magnitudes of these results represent the likelihood of the input image belonging to each class after computation. The output obtained using the Torchsummary package is shown in Figure 1.

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1            [-1, 6, 28, 28]             156
           Sigmoid-2            [-1, 6, 28, 28]               0
         AvgPool2d-3            [-1, 6, 14, 14]               0
            Conv2d-4           [-1, 16, 10, 10]           2,416
           Sigmoid-5           [-1, 16, 10, 10]               0
         AvgPool2d-6             [-1, 16, 5, 5]               0
           Flatten-7                  [-1, 400]               0
            Linear-8                  [-1, 120]          48,120
            Linear-9                   [-1, 84]          10,164
           Linear-10                   [-1, 10]             850
================================================================
Total params: 61,706
Trainable params: 61,706
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.11
Params size (MB): 0.24
Estimated Total Size (MB): 0.35
----------------------------------------------------------------
```

Figure 2 TXT Result of model shown as screenshot

In this subsection, a convolutional neural network model using the LeNet structure is created programmatically. However, there are still some shortcomings and room for improvement. The LeNet model is an early convolutional neural network model, and it differs significantly from later convolutional neural networks.

Firstly, the depth of the LeNet model is insufficient, as mentioned in the background introduction where it's noted that increasing model depth can improve accuracy. Secondly, the ReLU activation function is more widely used and faster in execution, being chosen by many popular convolutional neural networks including GoogLeNet. Thirdly, average pooling consumes more computational resources compared to max pooling, and from the results, it's observed that among the convolutional neural networks developed after LeNet, max pooling is more popular in pooling operations.

4.1.3 Train the model

First, the original dataset is divided into training and testing sets, and the parameters required for training are defined. Initially, the program splits the original dataset into training and testing sets with an 80-20 ratio. In PyTorch, the gradient descent algorithm is implemented by defining an "optimizer," and for training the convolutional neural network in this session, the Adam algorithm is used. The loss function used during training is the cross-entropy function from the PyTorch toolkit. Subsequently, a series of arrays are defined to record the accuracy and loss values of each epoch during training, which will be used to display the training results. A variable is defined to store the best-performing model, and it is saved in ".pth" file format using the deep copy method. The file "model_train.py" from the `Windows-code` folder in the remote repository implements the content of this section.

Secondly, the model training process is conducted. The program utilizes two for loops to control the training process: one for-loop controls the training epochs, and within it, two parallel for-loops are nested to sequentially execute training and evaluation. This paragraph mainly discusses the training process. During training, the model is set to training mode. In each training epoch, the training set images are trained in batches of 64. After each image goes through forward propagation, it outputs an array containing

10 variables, corresponding to the probabilities of digits 0 to 9. The loss value is obtained using the cross-entropy function. Before applying the gradient descent algorithm, the gradients in the optimizer should be zeroed. Then, the gradient descent algorithm is applied.

Thirdly, the model is evaluated during the training process. During evaluation, the model is set to evaluation mode. Evaluation mode reduces unnecessary computations during runtime in PyTorch. In evaluation mode, only forward propagation is performed, and the loss value is obtained using the loss function. Throughout the program execution, both training and testing processes save accuracy and loss values, storing them in corresponding arrays. This facilitates visualization of the training results. Ultimately, the program saves the model with the highest accuracy achieved.

Finally, the training process is visualized for analysis. Excessive training epochs may lead to overfitting, where the model performs well on the training set but poorly on unseen data. In simpler terms, the model may be able to recognize the digit "2" in the training set but fails to do so in the test set, despite the similarity between the two. Visualizing the entire training process helps in selecting an appropriate number of training epochs.

The experiment opted for 20 training epochs and displayed the loss and accuracy for each epoch. As depicted in Figure 3, the data shows a slight overfitting tendency after surpassing 10 epochs. The model's accuracy and loss seem to converge around the seventh epoch.

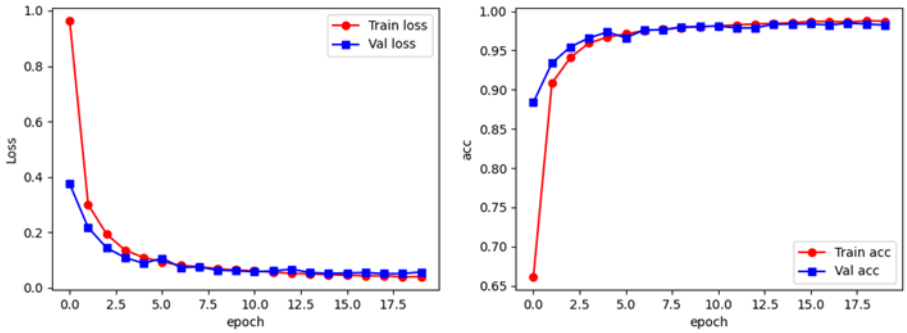

Figure 3 Training process

It's worth noting that PyTorch allows saving models in two different formats. During training, a deep copy method was used to save the model's state dictionary using `model.state_dict()` format. This format only includes the model parameters. Alternatively, you can save the entire model directly. In this experiment, since the model is relatively small, the file sizes of both formats are similar. The "best_model.pth" file in the remote repository contains the model parameters, while "whole_model.pth" contains the entire model. When using these two types of model files, the code will differ.

```
# load whole model
# model = torch.load('whole_model.pth')

# load model parameter
model = LeNet()
model.load_state_dict(torch.load('best_model.pth'))
```

The above code demonstrates two ways of using the model files. On Windows operating systems, both methods yield the same result. However, on Linux operating systems, loading the entire model (`whole_model.pth`) may result in errors related to Linux system permissions. According to PyTorch's official documentation, the second method, which involves creating a model object and loading the model parameters into

it, is considered the preferred approach. In this experiment, the second method is used to utilize the trained model on Linux operating systems.

4.1.4 Test the model

The process of testing the model is similar to evaluating the model during training. During testing, there is no batching operation on the dataset. Instead, each image is directly fed into the model for forward propagation, and the overall accuracy is computed. If readers need to inspect the classification results for each image, they can refer to the PyTorch documentation to implement their own code for outputting the results. The file `model_test.py` from the `Windows-code` folder in the remote repository implements this section's content. When performing the classification task, the accuracy rate is more than 98%.

4.1.5 Result and Discussion of this part

This section delineated the process of training convolutional neural networks, yielding a model achieving a classification accuracy exceeding 98% in the classification task, utilizing the LeNet network architecture. While the network models employed in this experiment may be somewhat antiquated, their usage mitigated stringent hardware requirements. Should there be improvements in device capabilities, the exploration of more contemporary and complex convolutional neural network models will be undertaken. Additionally, this section elucidated the intricacies encountered and resolved pertaining to the discrepancies in programming tools across different operating systems. Subsequent experiments will incorporate heightened consideration of such issues during the preparatory phase.

4.2 Pre-processing of normal image

This section mainly explains how to use the OpenCV-Python toolkit to preprocess images and obtain coordinates for subsequent robotic arm gripping operations. The

card with numbers displayed in this section serves as the object to be grabbed. The `CVmethod.py` file in the remote repository implements the content of this section. The `picture` folder in the `Windows-code` directory of the remote repository stores the images used in this section.

4.2.1 Processing

To expedite the experiment's efficiency, two different devices were used to capture images of the same card. The images saved in PNG format originate from the camera controlled by the Jetson Nano development board, while those saved in JPG format were captured using a smartphone.

Figure 4 Picture from camera

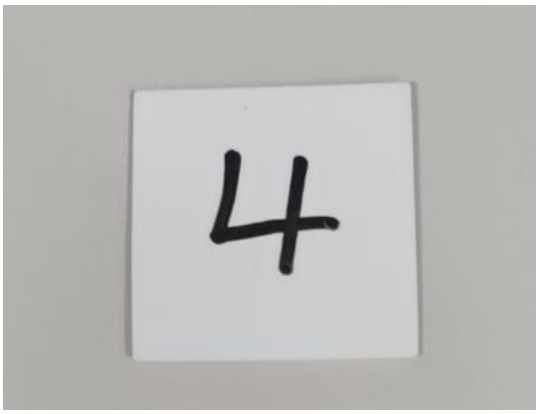

Figure 5 Picture from mobile phone

The experiment can be summarized into several steps, with each step's results showcased in Figure 6.

1. Image Reading and Conversion: The first step involves reading the image and converting it to grayscale.
2. Image Inversion and Binarization: In the second step, inversion and binarization operations are applied because the images used to train the network model have a black background.
3. Erosion and Dilation: The third step involves erosion and dilation operations.

4. Contour Detection: The next step utilizes functions from the OpenCV-Python toolkit to find all image contours.

5. Coordinate Generation: Useful contour information is used to generate coordinates, and the found contours are displayed.

6. Bounding Rectangle Drawing: Using the coordinates and contour information, bounding rectangles are drawn and visualized. Additionally, cropping of the original image to obtain the effective portion is performed (although this operation is relatively simple, the resulting images are too small for display in the figure).

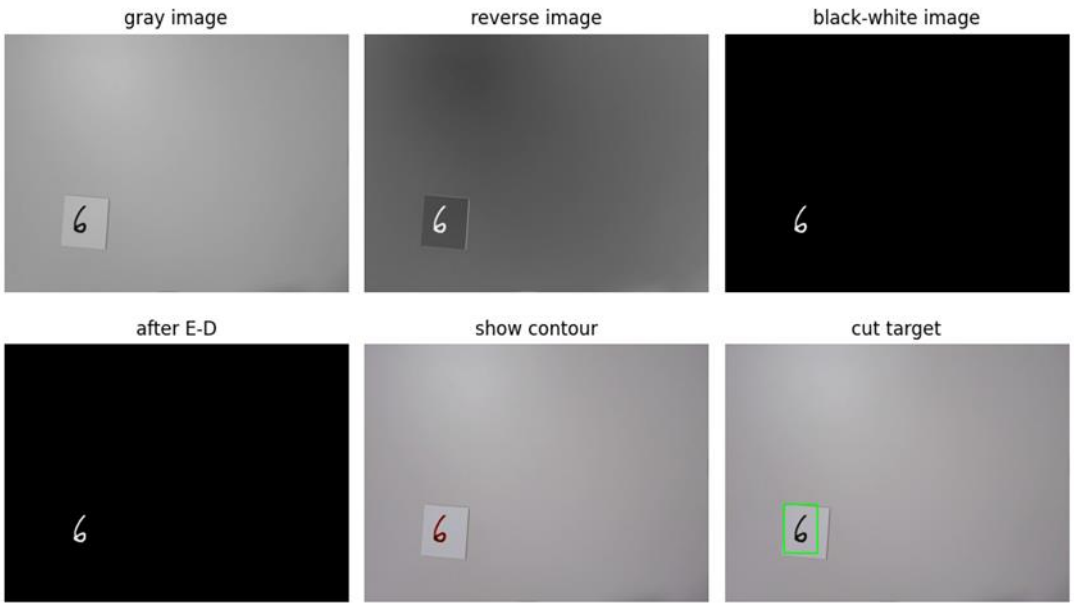

Figure 6 Processing

Each of these steps contributes to preprocessing the image data and extracting useful features for further analysis.

4.2.2 Processing Details

In image processing, a channel refers to an independent data plane storing image information. An image typically comprises one or multiple channels, each representing different aspects of the image. In three-channel colour images, the commonly used

RGB colour model represents the image. The RGB model decomposes colours into three colour channels: red (R), green (G), and blue (B). In such images, each pixel consists of three values representing the intensity of the red, green, and blue channels, respectively. By combining the intensities of these three channels, a colour image is generated, where the colour of each pixel is determined by the combination of red, green, and blue colours.

A grayscale image, on the other hand, is a single-channel image where each pixel contains only one value, representing the pixel's grayscale level or brightness. In grayscale images, colour information is disregarded, retaining only the brightness information of the image. Grayscale images are commonly used to represent black-and-white or monochrome images, where the grayscale value of each pixel represents the brightness level in the image, typically ranging from 0 (black) to 255 (white).

Regardless of whether the image is in JPG or PNG format, it is initially represented as a three-channel colour image. However, for training neural network models, it's necessary to convert the images into single-channel format, as required by the model. This conversion can be achieved using the "cvtColor()" function from the OpenCV library. After the conversion, an inversion operation is also required. OpenCV represents images using NumPy arrays, which are two-dimensional arrays consisting of integer values ranging from 0 to 255.

Next, the images will undergo a binarization process. The OTSU algorithm is a method used in image processing to automatically determine a threshold for converting grayscale images into binary images. One advantage of the OTSU algorithm is that it does not require a manually specified threshold; instead, it automatically determines the optimal segmentation threshold based on the grayscale distribution of the image. Therefore, it is suitable for various types of images. The experiment initially uses the

OTSU algorithm for this purpose.

In this experiment, the background colour may vary in intensity, but the colour of the digits is pure black. This is both the reason for choosing the OTSU algorithm and the cause of the errors encountered. However, OTSU does not require manually specified thresholds, which could make the results unpredictable (Figure 7 illustrates this process). Hence, ultimately, this experiment adopts a strategy of manually specifying the threshold.

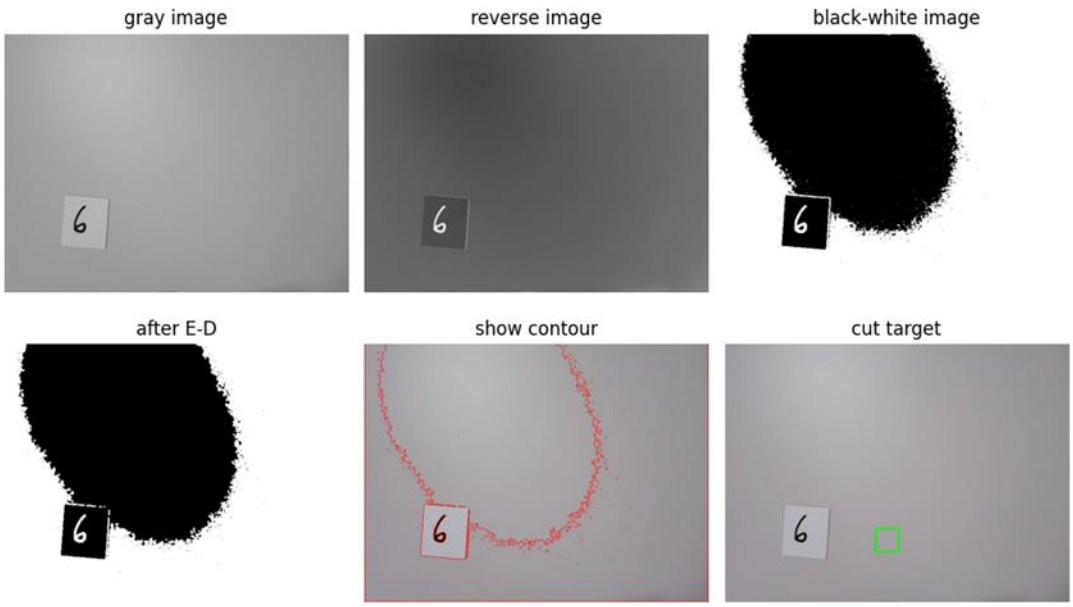

Figure 7 OTSU method

In this experiment, erosion and dilation operations were applied to the images, collectively known as "Opening" in image processing. This operation involves performing erosion followed by dilation on the image. Opening operation is commonly used to eliminate small-scale noise, smooth edges, separate touching objects, and extract bright features from images. It helps in removing small-scale details while preserving the main shape features. In this experiment, it helps in eliminating small-scale shadows caused by lighting variations (Figure 8 illustrates this process).

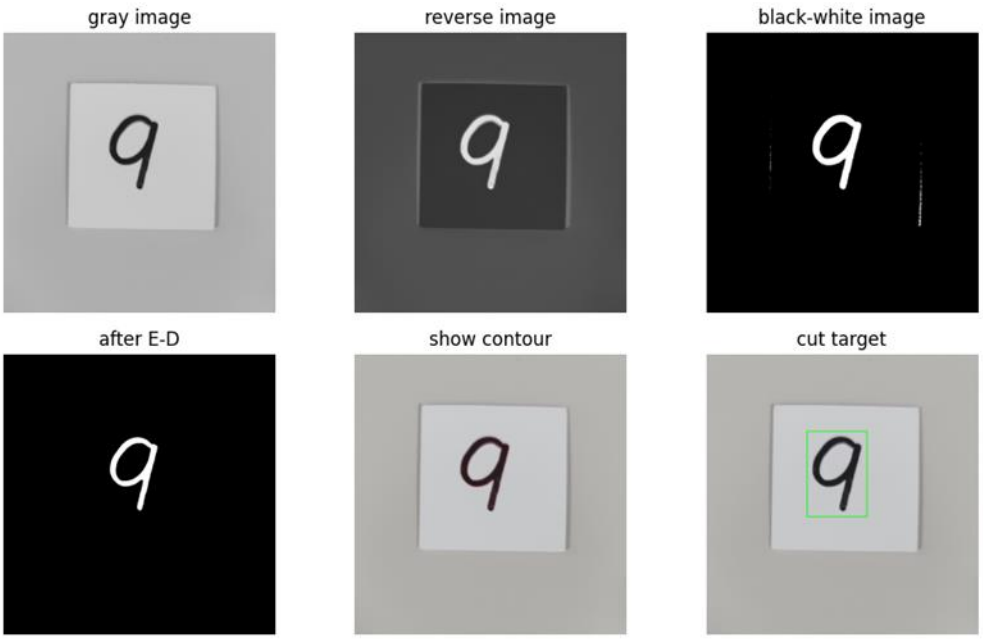

Figure 8 Erosion and Dilate

In the thresholding process, this experiment chose 180 as the threshold value. This choice aims to filter out as much invalid information as possible while retaining relevant details. Despite setting the threshold to 180, white invalid pixel blocks caused by shadows can still be observed in the lower-right region of the thresholded image. Due to the small size of the digits in the PNG format images, there is a risk of compromising the integrity of the digits during the erosion operation. Therefore, during the erosion operation, the size of the convolution kernel was set to 3x3 (Figure 9).

The choice of threshold value and the size of the convolution kernel are interconnected. A too large threshold value results in overly thin digits, compromising the overall integrity of the digits. The erosion operation helps eliminate invalid white pixels, and the size of the convolution kernel determines the extent of erosion. An excessively large kernel(like 5*5) size can also overly erode the digits in the image, leading to distortion. Based on extensive experimentation, this experiment adopted a threshold

value of 180 and used a 3x3 kernel size for the erosion operation.

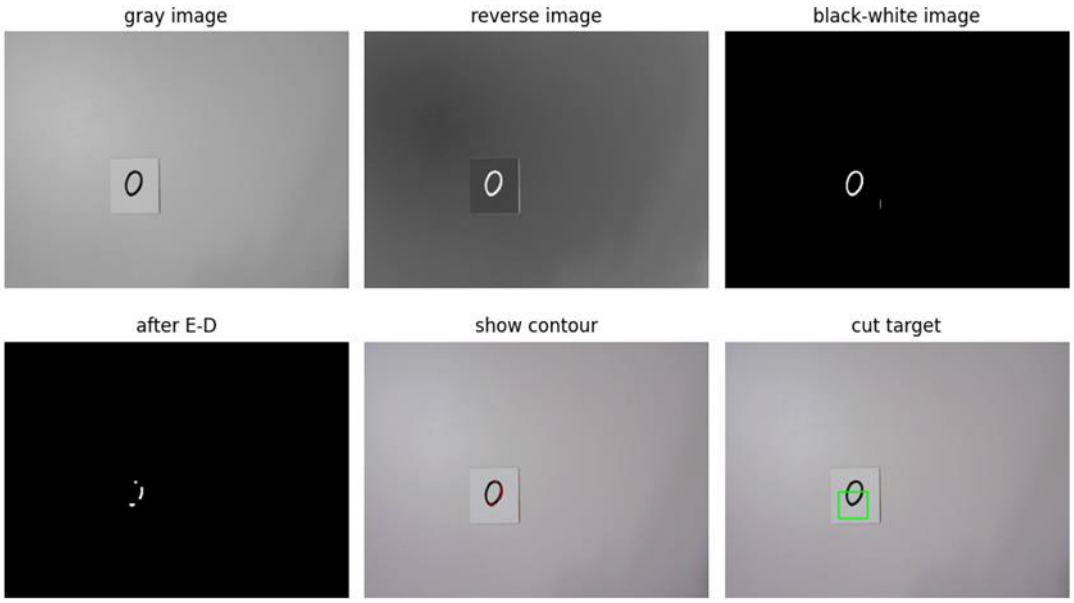

Figure 9 Bad Erosion

In contour detection, the algorithm utilized in this experiment detects all contours present in the image. The corresponding algorithm in the OpenCV toolkit organizes contours hierarchically. For instance, the digit '0' may comprise two contours: one representing the outer boundary of the digit, and the other representing the inner contour. The contour detection algorithm collects all contours and stores them in an array. After thresholding, erosion, and dilation operations on the image, only the digit portion remains. Through experimentation, it has been determined that accessing the first element in the array invariably retrieves the outer contour of the digit.

This experiment indeed adopts the mentioned approach, which not only identifies the correct contours but also simplifies the code logic. Upon determining the storage location of the correct contours, accessing this information enables retrieval of the contour length, coordinates, height, and width. With this information in hand, the image can be cropped to obtain the desired results. Displaying contours and bounding

rectangles would alter the original image. Additionally, contours can only be displayed on images with three channels and will break the original image. Therefore, during program execution, at each step of image processing, the result from the previous step is duplicated, and subsequent operations are performed on this duplicate. This approach also facilitates centralized display of images at the end of the program.

4.2.3 Result and Discussion of this part

This subsection accomplished image preprocessing operations, extracting pertinent information such as the pixel coordinates of digits during the process. The utilization of simplistic algorithms in this experiment streamlined code logic and mitigated the learning curve. However, as elucidated in the section regarding parameter settings, the simplicity of approach engendered complexity in the experimental process. Precision in parameter specification, down to the individual digit, proved imperative for attaining effective results or risked diminishing the algorithm's generality. While the contour detection function employed herein facilitated detection of multiple targets, limitations arose due to discrepancies in return values stemming from variances in OpenCV-Python versions, as well as the inherent complexity of return values, necessitating detection of singular digits. In future experiments, given ample time, allocation of additional resources towards acquiring proficiency in prevalent and intricate algorithms and knowledge domains will be prioritized.

4.3 Get and handle video signal

4.3.1 ROS package

On a Linux system, create a folder named "catkin_ws." The name of this folder must strictly adhere to the ROS naming convention, as "catkin_ws." This naming convention is mandated by ROS and cannot be altered for this experiment. Configure the ROS runtime environment using the "catkin_create_pkg" command. Referring to the tutorial provided by the hardware manufacturer, the command used in this

experiment is "catkin_create_pkg beginner_hiwonder std_msgs rospy roscpp." Here, "beginner_hiwonder" is the name of a folder, while "rospy" indicates that ROS-Python may be utilized in this environment. It is advised not to delete any part of the command, as lack of understanding of its significance will not impact the experiment. Finally, use the "catkin_make" command to build the packages in the workspace.

Upon completion of the aforementioned process, the "catkin_ws" folder will contain configuration files, including "build," "devel," and "src" directories. The "beginner_hiwonder" folder created earlier will be located within the "src" directory. Finally, execute the ". ~/catkin_ws/devel/setup.bash" command to activate the configuration files and add the "catkin_ws" workspace to the ROS environment.

4.3.2 Show the footage

In this experiment, the program subscribes to information from the "usb_cam" node. However, the information returned by this node cannot be directly manipulated by the OpenCV toolkit. As mentioned earlier, OpenCV stores image information using numpy arrays, so the information needs to be converted into the numpy array format to be provided to the interfaces in the OpenCV toolkit. Once the conversion is complete, the image content can be displayed using OpenCV windows. It is important to note that most images are stored in RGB format, while OpenCV only accepts images in the BGR format. While the conversion process is theoretically straightforward, OpenCV does not automatically determine the format of unknown image information. Therefore, manual conversion of the format is necessary, and this operation will also be used in subsequent parts of the program.

4.4 Catch object

The code used in this chapter is the "arm.py" file in the "Linux-code" folder of the remote repository. The approach of this experiment is as follows: First, reset the robotic

arm and camera to obtain an image from a fixed perspective (as shown in Figure 10). Then, proceed to pick up four different building blocks from the camera's field of view. The specific patterns on the blocks are not crucial; rather, the important aspect is their positions. The method "jetmax.set_position()" in the program is used to control the robotic arm. After recording the initial position of the robotic arm, the parameters are modified individually, and through iterative debugging, the robotic arm is moved to the correct positions while recording the parameters.

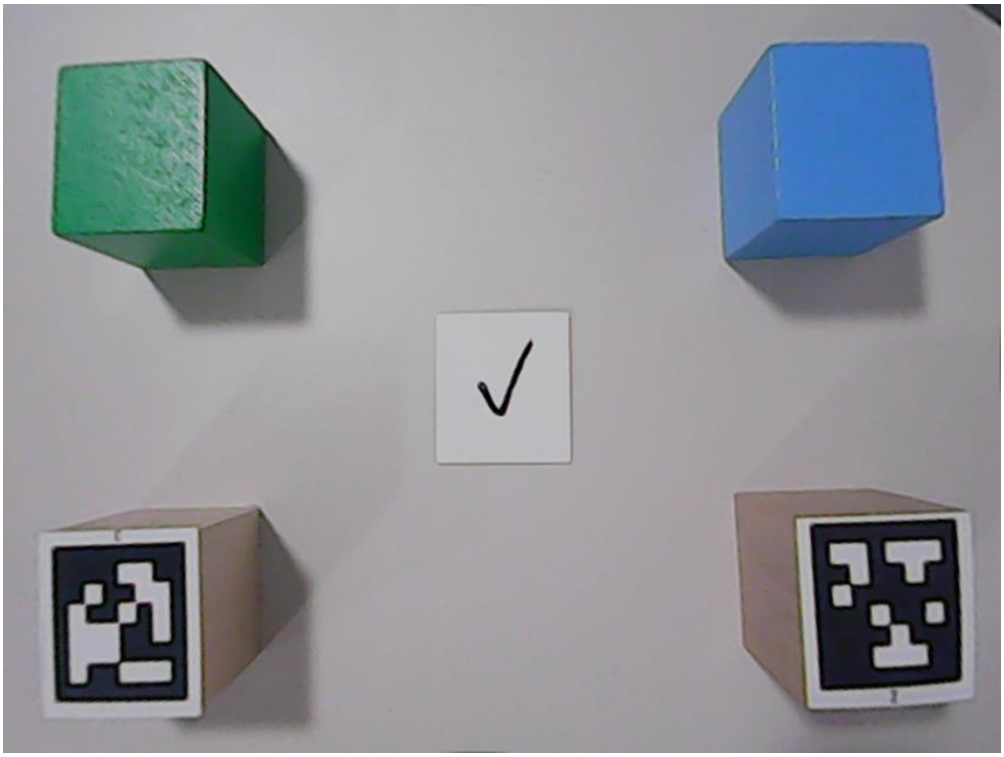

Figure 10 Location picture

After obtaining the correct parameters, proportional calculations are performed. The camera's field of view is 640x480 pixels. By recording the parameters when the robotic arm moves to the far left and far right positions, transformation can be performed. The same method can be applied to obtain the transformation for vertical coordinates. The pattern in the middle is used to verify whether the program's transformation results are correct.

Initially, for X axis, the formula obtained used 320 minus the coordinate value obtained from the image processing function, then divided by 2.5 for horizontal transformation.

However, after experimentation, due to the presence of errors, the value of 2.5 did not yield the correct coordinates. After extensive testing, a value of 2.2 was eventually used as the divisor, which significantly improved the accuracy of positioning the robotic arm to the correct location.

4.5 Consolidate the code and run

4.5.1 Image Processing and Testing

The programs used in this section are the `CVtest.py` file in the `Windows-code` directory and the `test.py` file in the `Linux-code` directory of the remote repository. These programs integrate the image processing functionality described earlier with the convolutional neural network (CNN) capabilities. By encapsulating the image processing into two functions to enhance the readability of the main method, the result obtained is the cropped image from the image processing process, as illustrated by the image labelled "cut target" in Figure 6 above, with a green border surrounding the region of interest. Subsequently, after processing through interfaces from the Pillow and torch packages, such an image can be utilized by the convolutional neural network for classification purposes.

It is important to note that after the forward pass computation, the convolutional neural network returns an array of length 10, where each element represents the probability corresponding to each class. In this experiment, the elements in this array correspond to the classes in ascending order of their indices, which are integers less than 10.

The `Cvtest.py` file running on Windows and the `test.py` file running on Linux are almost identical. The only difference is that the `test.py` file outputs the phrase "here" to the console. This is because invoking the GPU for computation, regardless of the operating system, incurs a significant overhead in terms of time. When running the program on my PC, the system takes about 3 seconds to invoke the GPU for assistance,

after which the time required for predicting a single image can be neglected. However, on the Linux system, it takes nearly 2 minutes to invoke the GPU, and thereafter, the time taken to predict a single image remains stable at 1 minute and 57 seconds.

4.5.2 Catch from video signal

The program used in this section is the `CVcatch.py` file located in the `Linux-code` folder of the remote repository. This program captures the final frame of the video feed after the display is stopped, and then analyses and detects objects in the captured image. After obtaining the coordinates of the target, it controls the robotic arm to grasp the target object.

It should be noted that due to the different working environment of ROS compared to Python, the program can only save images in a specific location within the `catkin_ws` directory. Saving images in other locations may result in errors related to Linux permissions. This is also the reason why the two important functionalities of the program are separated and executed independently.

4.5.3 Result and Discussion of 4.3、4.4 and 4.5

The manuscript consists of three sections detailing the process of programming with ROS tools on a Linux platform. It encompasses tasks such as reading and displaying video stream data, object grasping based on video information, and ultimately integrating the code on the Linux operating system. Throughout the experiment, reliance predominantly rested on vendor-provided code interfaces rather than direct signal reading and programming based on the pinouts of the Jetson Nano development board. While engineers are inherently tasked with exploring the hardware functionalities of devices, this experiment's scrutiny of hardware devices remained superficial, confined solely to their utilization.

Conclusion

In summary, this experiment utilized programming on the Jetson Nano development board and employed a convolutional neural network model to manipulate a robotic arm for object grasping while ensuring accurate classification thereof. Programming methodologies largely adhered to the official guidance documentation of the utilized toolkit, thereby ensuring code readability.

This experiment has yielded the following achievements. Firstly, MNIST dataset was employed to train a convolutional neural network capable of classifying handwritten digits, utilizing LeNet as the network architecture. PyTorch was utilized as the programming tool, with Python as the programming language. The obtained convolutional neural network model achieved a classification accuracy exceeding 98% on the test set. Secondly, OpenCV-Python was utilized as the tool for processing video and image data. Through programming, successful completion of two functionalities was achieved: real-time processing of images captured by the camera and preprocessing of images used for prediction. The program can identify the pixel coordinates of the target object to be detected in each image and crop out unnecessary parts of the image. Thirdly, ROS-Python was employed to manipulate the robotic arm for accurate grasping of the target object. Through experimentation, a formula for converting pixel coordinates into control parameters for the robotic arm was derived.

In future endeavours, with the support of high-performance equipment, it may be plausible to employ more popular and complex convolutional neural network models along with larger databases. The present experiment entailed programming efforts across both Windows and Linux operating systems, uncovering certain issues. In forthcoming experiments, subtle distinctions in programming tools across different operating systems will be thoroughly evaluated prior to commencement.

Reference

[1] C. N. E. Anagnostopoulos, I. E. Anagnostopoulos, V. Loumos, and E. Kayafas, “A License Plate-Recognition Algorithm for Intelligent Transportation System Applications,” IEEE Transactions on Intelligent Transportation Systems, vol. 7, no. 3, pp. 377–392, Sep. 2006, doi: https://doi.org/10.1109/tits.2006.880641.

[2] N. Sharma, V. Jain, and A. Mishra, “An Analysis Of Convolutional Neural Networks For Image Classification,” Procedia Computer Science, vol. 132, pp. 377–384, 2018, doi: https://doi.org/10.1016/j.procs.2018.05.198.

[3] S. Ali, Z. Shaukat, M. Azeem, Z. Sakhawat, T. Mahmood, and K. ur Rehman, “An efficient and improved scheme for handwritten digit recognition based on convolutional neural network,” SN Applied Sciences, vol. 1, no. 9, Aug. 2019, doi: https://doi.org/10.1007/s42452-019-1161-5.

[4] K. O’Shea and R. Nash, “An Introduction to Convolutional Neural Networks,” arXiv:1511.08458 [cs], Dec. 2015, Available: https://arxiv.org/abs/1511.08458

[5] T. D. Duan, T. L. H. Du, T. V. Phuoc and N. V. Hoang, "Building an automatic vehicle license-plate recognition system", Proc. Int. Conf. Comput. Sci. RIVF, pp. 59-63, 2005.

[6] W. Rawat and Z. Wang, “Deep Convolutional Neural Networks for Image Classification: A Comprehensive Review,” Neural Computation, vol. 29, no. 9, pp. 2352–2449, Sep. 2017, doi: https://doi.org/10.1162/neco_a_00990.

[7] Y. LeCun, Y. Bengio, and G. Hinton, “Deep Learning,” Nature, vol. 521, no. 7553, pp. 436–444, May 2015, doi: https://doi.org/10.1038/nature14539.

[8] S. Gerke, K. Muller, and R. Schafer, “Soccer Jersey Number Recognition Using Convolutional Neural Networks,” 2015 IEEE International Conference on Computer Vision Workshop (ICCVW), Dec. 2015, doi: https://doi.org/10.1109/iccvw.2015.100.

[9] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” Proceedings of the IEEE, vol. 86, no. 11, pp. 2278–2324, 1998, doi: https://doi.org/10.1109/5.726791.

[10] A. G. Howard et al., “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications,” arXiv.org, 2017. https://arxiv.org/abs/1704.04861

[11] I. J. Goodfellow, Y. Bulatov, J. Ibarz, S. Arnoud, and V. Shet, “Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks,” arXiv:1312.6082 [cs], Apr. 2014, Available: https://arxiv.org/abs/1312.6082

[12] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Ng, “Reading Digits in Natural Images with Unsupervised Feature Learning,” 2011.

[13] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, “Robot Operating System 2: Design, architecture, and uses in the wild,” Science Robotics, vol. 7, no. 66, May 2022, doi: https://doi.org/10.1126/scirobotics.abm6074.

[14] M. Quigley et al., “ROS: an open-source Robot Operating System,” 2009.

[15] M. Y. W. Teow, “Understanding convolutional neural networks using a minimal model for handwritten digit recognition,” IEEE Xplore, Oct. 01, 2017. https://ieeexplore.ieee.org/abstract/document/8239052

Appendix

URL of the remote repository

https://github.com/ZhangChenFu/HW_Final_Report.git

Minutes of meetings

Meeting Date	19 September 2023	Meeting Week	2 (Semester 1)
Meeting Duration	9:00 – 9:30 AM	Meeting Format	Offline meetings
Present: Dr. Keith Edgar Brown, Chenfu Zhang, Fuquan Zhang, Chenxi Meng, Wentian Wang, Zixin Wang			
Agenda: Project progress and project plan. Answering question.			

Meeting Date	26 September 2023	Meeting Week	3 (Semester 1)
Meeting Duration	9:00 – 9:30 AM	Meeting Format	Offline meetings
Present: Dr. Keith Edgar Brown, Chenfu Zhang, Fuquan Zhang, Chenxi Meng, Wentian Wang, Zixin Wang			
Agenda: Project progress and project plan. Answering question.			

Meeting Date	3 October 2023	Meeting Week	4 (Semester 1)
Meeting Duration	9:00 – 9:30 AM	Meeting Format	Offline meetings
Present: Dr. Keith Edgar Brown, Chenfu Zhang, Fuquan Zhang, Chenxi Meng, Wentian Wang, Zixin Wang			
Agenda: Project progress and project plan. Answering question.			

Meeting Date	10 October 2023	Meeting Week	5 (Semester 1)
Meeting Duration	9:00 – 9:30 AM	Meeting Format	Offline meetings
Present: Dr. Keith Edgar Brown, Chenfu Zhang, Fuquan Zhang, Chenxi Meng,			

Wentian Wang, Zixin Wang
Agenda: Project progress and project plan. Answering question.

Meeting Date	24 October 2023	Meeting Week	7 (Semester 1)
Meeting Duration	9:00 – 9:30 AM	Meeting Format	Offline meetings
Present: Dr. Keith Edgar Brown, Chenfu Zhang, Fuquan Zhang, Chenxi Meng, Wentian Wang, Zixin Wang			
Agenda: Project progress and project plan. Answering question.			

Meeting Date	31 October 2023	Meeting Week	8 (Semester 1)
Meeting Duration	9:00 – 9:30 AM	Meeting Format	Offline meetings
Present: Dr. Keith Edgar Brown, Chenfu Zhang, Fuquan Zhang, Chenxi Meng, Wentian Wang, Zixin Wang			
Agenda: Project progress and project plan. Answering question.			

Meeting Date	7 November 2023	Meeting Week	9 (Semester 1)
Meeting Duration	9:00 – 9:30 AM	Meeting Format	Offline meetings
Present: Dr. Keith Edgar Brown, Chenfu Zhang, Fuquan Zhang, Chenxi Meng, Wentian Wang, Zixin Wang			
Agenda: Project progress and project plan. Answering question.			

Meeting Date	14 November 2023	Meeting Week	10 (Semester 1)
Meeting Duration	9:00 – 9:30 AM	Meeting Format	Offline meetings
Present: Dr. Keith Edgar Brown, Chenfu Zhang, Fuquan Zhang, Chenxi Meng, Wentian Wang, Zixin Wang			
Agenda: Project progress and project plan. Answering question.			

Meeting Date	21 November 2023	Meeting Week	11 (Semester 1)

Meeting Duration	9:00 – 9:30 AM	Meeting Format	Offline meetings
Present: Dr. Keith Edgar Brown, Chenfu Zhang, Fuquan Zhang, Chenxi Meng, Wentian Wang, Zixin Wang			
Agenda: Project progress and project plan. Answering question.			

Meeting Date	16 January 2024	Meeting Week	1 (Semester2)
Meeting Duration	10:00 – 10:30 AM	Meeting Format	Offline meetings
Present: Dr. Keith Edgar Brown, Chenfu Zhang, Fuquan Zhang, Chenxi Meng, Wentian Wang, Zixin Wang			
Agenda: Project progress and project plan. Answering question.			

Meeting Date	23 January 2024	Meeting Week	2 (Semester2)
Meeting Duration	10:00 – 10:30 AM	Meeting Format	Offline meetings
Present: Dr. Keith Edgar Brown, Chenfu Zhang, Fuquan Zhang, Chenxi Meng, Wentian Wang, Zixin Wang			
Agenda: Project progress and project plan. Answering question.			

Meeting Date	30 January 2024	Meeting Week	3 (Semester2)
Meeting Duration	10:00 – 10:30 AM	Meeting Format	Offline meetings
Present: Dr. Keith Edgar Brown, Chenfu Zhang, Fuquan Zhang, Chenxi Meng, Wentian Wang, Zixin Wang			
Agenda: Project progress and project plan. Answering question.			

Meeting Date	6 February 2024	Meeting Week	4 (Semester2)
Meeting Duration	10:00 – 10:30 AM	Meeting Format	Offline meetings
Present: Dr. Keith Edgar Brown, Chenfu Zhang, Fuquan Zhang, Chenxi Meng, Wentian Wang, Zixin Wang			

Agenda: Project progress and project plan. Answering question.

Meeting Date	13 February 2024	Meeting Week	5 (Semester2)
Meeting Duration	10:00 – 10:30 AM	Meeting Format	Offline meetings
Present: Dr. Keith Edgar Brown, Chenfu Zhang, Fuquan Zhang, Chenxi Meng, Wentian Wang, Zixin Wang			
Agenda: Project progress and project plan. Answering question.			

Meeting Date	27 February 2024	Meeting Week	7 (Semester2)
Meeting Duration	10:00 – 10:30 AM	Meeting Format	Offline meetings
Present: Dr. Keith Edgar Brown, Chenfu Zhang, Fuquan Zhang, Chenxi Meng, Wentian Wang, Zixin Wang			
Agenda: Project progress and project plan. Answering question.			

Meeting Date	5 March 2024	Meeting Week	8 (Semester2)
Meeting Duration	10:00 – 10:30 AM	Meeting Format	Offline meetings
Present: Dr. Keith Edgar Brown, Chenfu Zhang, Fuquan Zhang, Chenxi Meng, Wentian Wang, Zixin Wang			
Agenda: Project progress and project plan. Answering question.			

Meeting Date	12 March 2024	Meeting Week	9 (Semester2)
Meeting Duration	10:00 – 10:30 AM	Meeting Format	Offline meetings
Present: Dr. Keith Edgar Brown, Chenfu Zhang, Fuquan Zhang, Chenxi Meng, Wentian Wang, Zixin Wang			
Agenda: Project progress and project plan. Answering question.			

Meeting Date	19 March 2024	Meeting Week	10 (Semester2)
Meeting Duration	10:00 – 10:30 AM	Meeting Format	Offline meetings

Present: Dr. Keith Edgar Brown, Chenfu Zhang, Fuquan Zhang, Chenxi Meng, Wentian Wang, Zixin Wang
Agenda: Project progress and project plan. Answering question.

Meeting Date	26 March 2024	Meeting Week	11 (Semester2)
Meeting Duration	10:00 – 10:30 AM	Meeting Format	Offline meetings
Present: Dr. Keith Edgar Brown, Chenfu Zhang, Fuquan Zhang, Chenxi Meng, Wentian Wang, Zixin Wang			
Agenda: Project progress and project plan. Answering question.			

Meeting Date	2 April 2024	Meeting Week	12 (Semester2)
Meeting Duration	10:00 – 10:30 AM	Meeting Format	Offline meetings
Present: Dr. Keith Edgar Brown, Chenfu Zhang, Fuquan Zhang, Chenxi Meng, Wentian Wang, Zixin Wang			
Agenda: Project progress and project plan. Answering question.			