CALIFORNIA STATE UNIVERSITY, NORTHRIDGE

IMAGE-BASED CONTROL OF HUMANOID ROBOTS USING DEEP NEURAL NETWORKS: A VISION-TO-ACTION FRAMEWORK

A graduate project submitted in partial fulfillment of the requirements

For the degree of Master of Science

in Computer Engineering

By

Nabin Jaiswal

December 2025

The graduate project of Nabin Jaiswal is approved:

Dr. Shahnam Mirzaei	Date
Dr. Gary Burke	Date
Dr. Myung (Michael) Cho, Chair	Date

California State University, Northridge

Acknowledgements

I would like to express sincere thanks to all my supporters during this project. Foremost, I thank to my project advisor and committee chair, Dr. Myung (Michael) Cho. I am thankful for his invaluable guidance, insightful suggestions, and steadfast encouragement through every single step of this research. Indeed, his mentorship has greatly influenced the direction and the nature of this work. I would also wish to thank my committee members, Dr. Shahnam Mirzaei and Dr. Gary Burke, whose comments, critiques, and insights greatly added to this work, as well as for their time and generosity. A big thanks go to my colleagues and fellow lab members, with whom we shared sparks of intellect, spirit of harmony, and friendship, creating a stimulating and encouraging atmosphere in the research environment. I would also particularly thank the wider academic community and staff who gave administrative or technical support along the way. But above all, I am greatly indebted to the Almighty and my family for their infinite love, patience, and belief in me during both the hard and the triumphant times. Their support, emotionally and morally, has been the basis of my perseverance, and without them, this achievement would never have been realized.

Table of Contents

List of Figures

Abstract

Image-Based Control of Humanoid Robots Using Deep Neural Networks: A Vision-to-Action Framework

By

Nabin Kumar Jaiswal

Master of Science in Computer Engineering

This project presents an end-to-end vision-to-action system for controlling humanoid robots from image inputs and deep neural networks. The approach integrates visual perception and control by employing a convolutional neural network (CNN) to directly map raw camera images to high-level control actions. The Context of this work lies in traditional visual servoing frameworks, which rely on robot control from camera feedback, and recent progress in deep learning facilitating strong image recognition. The Problem Statement captures the necessity for real-time image-based control of a humanoid robot to address classical controller's constraints in dynamic, unstructured settings. The Proposed Methodology involves constructing a labeled image dataset from the humanoid robot's camera and training a CNN to predict the right robot action for each image. The model structure and training are explained in a detailed manner, and an evaluation plan is constructed to quantify classification accuracy and control performance. Results indicate that the CNN has very high accuracy in predicting the correct actions, with training and validation accuracies at virtually 99% and the final test accuracy at over 99%. The confusion matrix and classification report indicate near-exact performance for all action classes, which validates the applicability of deep learning to precise vision-based control. In the Discussion, we analyze the model's performance, notice the minute errors and their possible causes, and make remarks on the consequences of using such a system on physical humanoid robots.

Finally, the Conclusion outlines the contributions and findings, acknowledging current limitations such as the need for large training datasets and light real-world testing. Future Directions for work include testing on real humanoid platforms, exploring continuous control outputs, and integrating reinforcement learning for increased adaptability. This work demonstrates a compelling vision-to-action architecture using deep neural networks to enable image-based control of humanoid robots, closing the visual perception-motor action gap in robotics.

Chapter 1: Introduction

1.1. Background

Humanoid robots are designed to imitate human movement and behavior, usually featuring a body structure that resembles ours complete with two arms, two legs, and a vision system located on the head. To navigate complex environments, these robots rely on sophisticated algorithms that can process sensory information and produce the right motor actions in real time. Among the various senses, vision is particularly crucial, as cameras offer detailed insights into the robot's surroundings.

A key technique in visual-based robotic control is known as visual servoing. This method utilizes feedback from a camera within a control loop to steer the robot's movements. Traditional visual servoing approaches focus on tracking specific features in the camera's view like points, lines, or contours and use this information to create control signals that help the robot reach a desired position or accomplish a task. There are two primary strategies: position-based visual servoing (PBVS), which reconstructs the robot's 3D pose from the image features, and image-based visual servoing (IBVS), which measures errors directly in the image plane without needing to reconstruct the 3D. A foundational tutorial by Hutchinson et al. in 1996 laid the groundwork for visual control in robotics and continues to be a significant reference in the field [1].

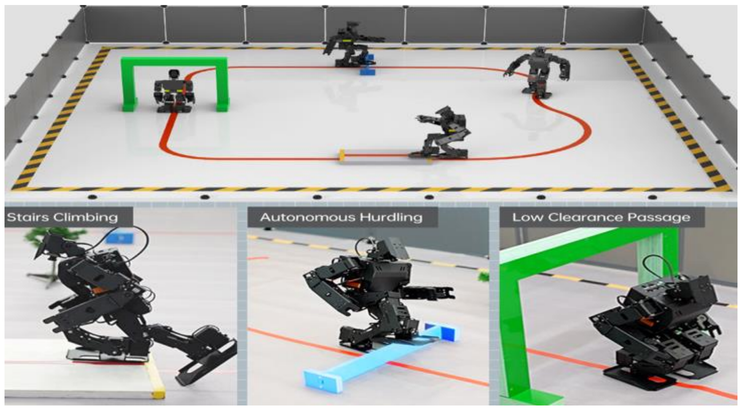


Figure 1.1: Vision-to-action framework in Robotics

(https://www.hiwonder.com/products)

However, classical visual servoing often depends on carefully crafted image processing pipelines that extract necessary visual features through methods like filtering, edge detection, or marker tracking, followed by geometric calculations. These pipelines are time-consuming to design and are often tailored to specific use cases. Furthermore, humanoid robots pose additional challenges because of their high degrees of freedom and complex dynamics. They must maintain balance and coordination even as they respond to visual inputs. In past research, this problem has been approached by extending visual servoing to whole-body control frameworks. These systems often use optimization techniques to spread visual corrections across many joints of the humanoid robot. While effective, such methods rely heavily on accurate calibration of camera parameters and robot kinematics. In real-world environments where lighting conditions, backgrounds, or calibration can vary these systems may fail to perform reliably. These limitations have led researchers to explore learning-based methods that can extract robust features and control strategies directly from data, removing the need for manual feature engineering.

Meanwhile, deep learning has transformed the field of computer vision. CNNs have achieved exceptional performance in image recognition tasks by learning layered feature representations from raw images data. This progress has opened up new possibilities in robotics. Rather than depending on handcrafted visual features, robots can now learn to interpret complex scenes directly through experience, using CNNs. By training these networks on large datasets, robots can generalize across a wide range of environmental variations. This progress has led to the development of end-to-end vision-to-action models, where a single neural network learns to map camera images directly to control commands. Such models are especially promising for humanoid robots, which must respond to visual cues in a coordinated and dynamic way. This project builds on that foundation, aiming to combine the stability and precision of traditional control methods with the flexibility and learning ability of deep neural networks to create an effective, vision-based control system for humanoid robots.

1.2. Problem Statement

Even with all the progress we've made in visual servoing and the rise of deep learning, getting humanoid robots to control their movements based on images is still a tough nut to crack. One of the main challenges is figuring out how to connect the complex visual data we get from cameras to the simple motor commands needed to move a robot's joints. Traditional techniques rely heavily on accurate models and extracting features, but these can easily stumble in messy environments or when there's noise in the sensory data. On the flip side, methods that are purely based on learning face their own hurdles, like having limited training data and the danger of overfitting to certain situations. So, the question this project aims to tackle is:

How can we create and implement a vision-to-action system that allows a humanoid robot to effectively understand camera images of its surroundings and respond with the right control actions in real time, all while harnessing the capabilities of deep neural networks to ensure the robot moves reliably and precisely?

This problem statement encapsulates several underlying challenges. First, the perception challenge: the robot must make sense of raw images that may contain varying backgrounds, lighting conditions, and viewpoints. Relevant information (such as target objects or landmarks necessary for the task) needs to be extracted from these images. Second, the decision-making challenge: given the interpreted visual information, the system must decide on the appropriate action for the humanoid to take. This could range from high-level decisions (e.g., move forward, turn left) to more granular motor commands. Third, the control challenge: the chosen action must be executed by the humanoid’s control system, which involves coordinating multiple joints while maintaining balance and stability.

The specific focus of this project is on the perception and decision-making aspects more concretely, on using a deep neural network to map an image directly to a discrete action command for the robot. By formulating it as a classification problem (where the network selects the correct action from a predefined set based on the image), we aim to simplify the control task while still demonstrating the feasibility of end-to-end learning for humanoid vision-based control. The problem statement targets a scenario where a humanoid robot operates in a controlled environment and must perform certain navigational or manipulation tasks (for example, reaching a target or avoiding an obstacle) purely based on its camera input. Traditional controllers might struggle here without extensive calibration; hence, we investigate whether a learned model can handle these conditions more gracefully. Ultimately, solving this problem would mean a humanoid could "see and act" – interpreting visual scenes and immediately producing motor responses – without manual intervention at intermediate stages.

1.3. Objectives

The main goal of this project is to create a deep learning system that can control a humanoid robot using images, and to assess how well it performs on a specific task. We can break this big goal down into a couple of clear objectives:

- Objective 1: Developing the Vision-to-Action Model. We need to design and build a convolutional neural network (CNN) that takes an image from a camera and translates it into a control action for the humanoid robot. This model should be able to effectively learn the visual cues that indicate the right action to take.
- Objective 2: Collecting and Preparing the Dataset. We gathered a set of images along with the correct action labels that go with them. This dataset should include all the relevant scenarios the humanoid might encounter (like different positions of a target or obstacle), so the model can learn to deal with common visual variations. This also involves any necessary preprocessing to enhance the model's learning experience.
- Objective 3: Training and Validation. Train the neural network on the collected dataset using supervised learning techniques. Monitor training progress through accuracy and loss metrics, and validate the model on a hold-out validation set to tune hyperparameters and

prevent overfitting. This includes selecting an appropriate optimization algorithm and loss function for multi-class action classification.

- Objective 4: Performance Evaluation. Thoroughly assess the trained model using a test set of images that weren't part of the training process. Utilize evaluation metrics such as overall classification accuracy, precision and recall for each class (through a classification report), and a confusion matrix to pinpoint any consistent errors. Make sure the results are statistically significant and not just a product of random chance or overfitting issues.

1.4. Scope

This research project aims to strike a balance between ambition and practicality, honing in on a specific implementation and evaluation within a controlled environment. The scope is clearly defined by the system's capabilities, the assumptions we've made, and the elements we've intentionally left out:

- Humanoid Robot Platform (Simulation). We're using images and data that reflect a humanoid robot's viewpoint for this project. Given our resource limitations, the experiments take place in a simulated setting or with pre-recorded image data, rather than on an actual humanoid robot. So, while we're demonstrating the approach in principle, it's important to note that further engineering would be necessary to make it work on a real robot.

1.5. Contributions

This project contributes to the field of autonomous systems and robotics in a number of ways, particularly in the area of vision-based deep learning for humanoid robot control. Below is a summary of the main contributions:

- Contribution 1: Vision-to-Action Framework for Humanoids. We've created an innovative framework that combines a convolutional neural network for visual perception with a control decision module specifically for humanoid robots. Unlike traditional methods that separate perception and control, our end-to-end approach tightly integrates these processes by generating control actions directly from raw images. This adds a practical example to the expanding research on end-to-end robotic control, particularly for high-DoF (degree-of-freedom) humanoid systems.

- Contribution 2: Labeled Image Dataset for Humanoid Control. As part of this project, we've compiled and annotated a dataset featuring images from a humanoid perspective, matched with the appropriate control actions. This dataset is among the first of its sort created to train a vision-based controller for humanoid robots, though being customized for our particular situation. Future researchers may find it useful as a reference, and we have recorded the data gathering and labeling process to guarantee reproducibility.
- Contribution 3: Empirical Evaluation of Deep Learning in Visual Servoing. We dive deep into how a deep learning model performs in the realm of visual servoing. By examining training curves, confusion matrices, and classification reports, we shed light on how effectively and swiftly a CNN can master a visual servoing task. The impressive accuracy levels we found reveal the conditions under which a deep model can either memorize or generalize the mapping, providing valuable insights into the potential of supervised learning for precise control tasks.
- Contribution 4: Identification of Model Strengths and Failure Modes. Through a thorough performance analysis, including a look at the few mistakes the model makes, we pinpoint which visual scenarios the network excels in and where it might stumble. For instance, our findings might show that the model has difficulty with certain edge-case images or tricky perspectives, emphasizing the significance of these cases for training or the necessity for additional methods. This insight is crucial for practitioners to grasp the limitations of using such a model in real-world applications.

- Contribution 5: Guidance for Future Integration. Lastly, this work lays out clear future directions and considerations for transitioning an approach like this from simulation to real-world application. The project discusses what would be necessary to implement the system effectively, ensuring a smooth shift from theory to practice.

Chapter 2: Literature Review

2.1. Related Work

Research at the intersection of computer vision and robot control has evolved significantly over the past few decades. In this section, we review the related work that provides a foundation for our project. We begin by discussing traditional visual servoing approaches, which are the classical methods for image-based robot control. We then explore the introduction of convolutional neural networks (CNNs) in robotics**,** highlighting how they have transformed perception. Next, we examine end-to-end vision-to-action strategies that have emerged with deep learning, where robots learn to map images directly to control commands. We also compare supervised learning vs. reinforcement learning approaches in the context of training robots, as this is a critical methodological distinction in contemporary research. We identify the limitations of current methods**,** which motivate the need for new approaches like the one taken in this project.

2.1.1. Traditional Visual Servoing

Traditional visual servoing is an actuation method where visual information (typically coming from cameras) is utilized to control the motion of robots in a closed loop. The core idea, as detailed in Hutchinson et al.'s tutorial for visual servo control, is to minimize the difference between certain visual features perceived by the camera from where they are desired [1]. Visual servoing schemes are commonly classified into two broad categories: Position-Based Visual Servoing (PBVS) and Image-Based Visual Servoing (IBVS).

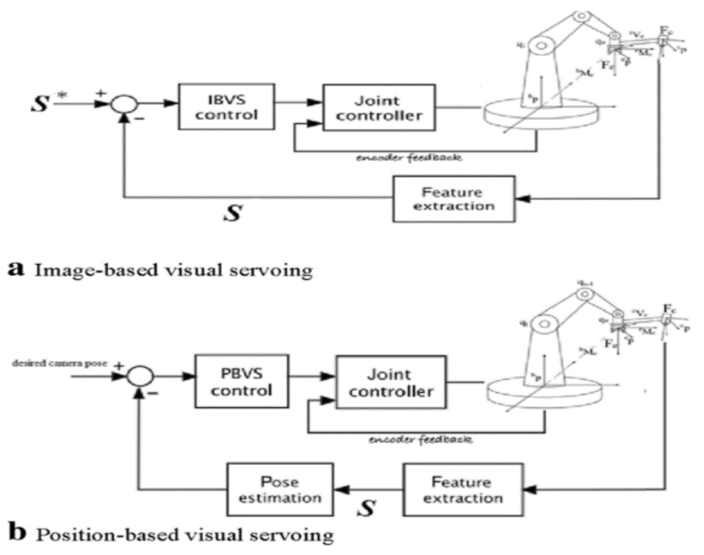


Figure 2.1: Position-Based Visual Servoing (PBVS) and Image-Based Visual Servoing (IBVS) [2]

(https://link.springer.com/article/10.1007/s41315-023-00270-6)

- Position-Based Visual Servoing (PBVS): PBVS estimates the target's 3D pose with respect to the robot or the camera, in terms of position and orientation, from image data. It often requires an already known model of the object and camera calibration. When the pose is estimated, traditional control algorithms (e.g., PID controllers) then drive the robot to make the current and desired poses as similar as possible. Basically, PBVS acts in the 3D Cartesian space; it maps image features into physical coordinates and subsequently derives control actions.
- Image-Based Visual Servoing (IBVS): IBVS, in contrast, operates in the image space. Rather than computing the 3D pose explicitly, IBVS formulates the error in terms of image feature parameters (e.g., difference in pixel positions of the current feature location and desired feature location). The controller attempts to minimize the image error through commanding robot motion.

2.1.2. CNNs in Robotics

Convolutional Neural Networks (CNNs) have significantly advanced the state-of-the-art in image analysis, which is particularly important for robotics since many robotic tasks depend on understanding visual information. The achievements of CNNs in image classification (such as AlexNet's landmark performance on ImageNet in 2012) and object detection have given robots enhanced "vision" – they can now identify and distinguish numerous objects and scenes when adequately trained. In robotics research, CNNs started being utilized for tasks like object identification, scene analysis, and depth estimation, all of which contribute to a robot's decision-making capabilities. Here are the types of CNNs.

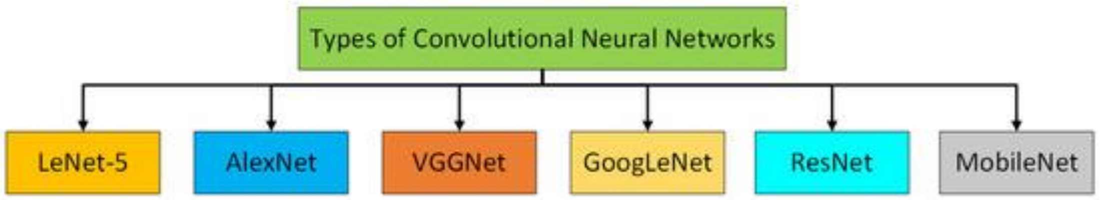


Figure 2.1.2: Types of Convolutional Neural Networks (CNNs) [3]

For example, a CNN can enable a robot to recognize objects to pick and place, or to detect obstacles for navigation. Early applications of CNNs in robotics often kept the perception and control modules separate: the CNN would process images and produce some intermediate result (like identifying where an object is), and then a traditional controller would use that information to act. This was a departure from classical vision algorithms which might have used feature descriptors or manual segmentation; CNNs provided a data-driven way to get more robust perception under varying conditions.

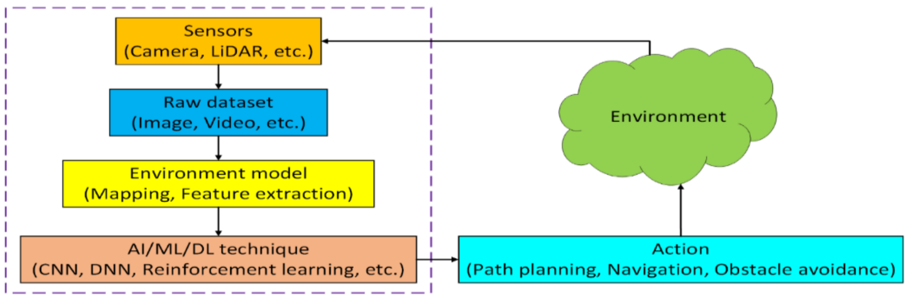

Sensors
(Camera, LiDAR, etc.)
Raw dataset
(Image, Video, etc.)
Environment model
(Mapping, Feature extraction)
AI/ML/DL technique
(CNN, DNN, Reinforcement learning, etc.)
Environment
Action
(Path planning, Navigation, Obstacle avoidance)

Figure 2.1.2.1: Illustration of a robotic perception framework [3]

2.1.3. Supervised vs Reinforcement Learning

When it comes to training robots to make decisions, two major paradigms are often discussed: supervised learning and reinforcement learning (RL). Both have been applied in vision-based robot control, but they differ in approach and applicability.[4]

The choice between supervised and reinforcement learning often comes down to the problem setting. For our project, supervised learning is appropriate because we have a clear idea of the correct action for each situation and can build a dataset accordingly. It allows us to directly teach the network the mapping and quickly assess performance on a test set [5]. If, instead, we were dealing with a scenario where defining the correct action per image was hard (maybe the best action depends on a long-term outcome), then reinforcement learning might be more suitable. In the long run, for a humanoid to truly autonomously learn in a complex environment, it will likely need some reinforcement learning capability. But that also would require either simulation or extremely careful training on the real robot, which as noted, is outside our scope here.

Chapter 3: Methodology

3.1. Dataset Description

A critical component of this project is the dataset used to train and evaluate the vision-to-action model. The dataset was custom-prepared to reflect the visual inputs a humanoid robot would encounter in the chosen task scenario, paired with the correct action that the robot should take. Below we describe the dataset in detail, including its contents, format, and how it was collected.

Images and Scenario: The dataset consists of images captured from a humanoid robot's first-person (onboard camera) perspective. Each image is a snapshot of the robot's environment at a given time. In our scenario, the humanoid is engaged in a navigation and object approach task: it must move toward a target

object placed in its environment. The camera is mounted roughly at the robot's eye level, providing a view that includes the ground in front of the robot and any objects at a reachable height.

Each image in the dataset is labeled with one of several discrete action classes that the robot should execute when seeing that image. The action set (which was determined based on the needs of the task) is as follows:

- Move Forward: Advance straight ahead toward the target.
- Move Backward: Move Backward
- Turn Left: Rotate left
- Turn Right: Rotate right analogously.
- Stop: Come to a halt

In total, the dataset contains 881 labeled images. This number includes both training and validation images (used during the learning process) as well as test images (held out for final evaluation). The breakdown is as follows:

- Training set: 700 images (approximately) – used to train the CNN model.

- Validation set: 100 images (approximately) – used to validate and tune the model during training (e.g., to perform early stopping or choose hyperparameters).
- Test set: 81 images used only after training is complete, to assess the model's performance on completely unseen data.

(Note: The above split is an approximation since the exact round numbers are adjusted to match 881 total images. In practice, we used an 80%/10%/10% split for train/val/test, which yields 705/88/88 if we round to nearest whole image. The slight difference is due to ensuring an even distribution of classes in each split.)

Data Collection: The images were collected in a controlled indoor environment using a simulated humanoid robot. The simulation allowed setting up various positions of the target object and varied orientations of the robot. For diversity, the lighting in the simulator was randomly varied between bright and dim, and the floor texture was switched between two types (e.g., tiled floor vs. carpet pattern). This introduced some visual variety to prevent the model from latching onto a trivial cue like a single background color or a specific lighting reflection. The humanoid was placed at different starting locations relative to the target for each image capture. Sometimes the target was directly ahead, sometimes to the left or right, and in a few cases very near (to simulate the condition for the "Stop" or "Move" action).

For each configuration, the correct action label was determined by a simple rule-based expert: essentially, the shortest path direction to the target was computed (forward, forward-left, or forward-right), which translated to either Move Forward, Turn Left, or Turn Right. If the target was extremely close (within a predefined small distance), the action was labeled as Stop (and if this were a real scenario, the next step might be to pick up or interact with the object). These rules ensured that the labels are consistent and logical.

Image Resolution and Format: The images were saved at a resolution of 224 x 224 pixels with three color channels (RGB). The resolution was chosen to balance detail and computational efficiency. It is large enough for the CNN to discern the target object and general scene layout, but not so large as to be computationally prohibitive for training the network. All images are JPEG format.

Class Distribution: The distribution of images among the classes was kept roughly balanced to avoid biasing the model towards any particular action. Roughly, each class (Forward, Left, Right, Stop) has about 200-250 images in the overall dataset.

3.2. Preprocessing

Prior to training the deep learning model, the dataset's raw images go through multiple preprocessing steps. Preprocessing plays a crucial role in machine learning workflows to improve model effectiveness, shorten training duration, and at times expand the dataset's size using augmentation techniques. In this part, we outline the preprocessing methods used on our image dataset.

Resizing and Normalization: Although the images were collected at 224 x 224 resolution, as mentioned, we ensured that all images are consistently resized to this resolution (if they weren't already) so that they can be batched and fed into the CNN. Uniform image size is a requirement for CNN input; moreover, 224 x 224 is a common input dimension for many CNN. After resizing, pixel values of the images are normalized. We scale the RGB pixel intensities from their original range of [0, 255] to a range of [0, 1] by simple division by 255. This normalization helps with the training process by bringing all input features to a comparable scale, which often leads to faster convergence of gradient descent. In addition, we subtract the dataset mean from each image (per color channel) to zero-center the data. Zero-centering is a standard practice that can make the optimization landscape smoother for neural network training.

Label Encoding: Since the output of our network is a discrete action, we encode the action labels into a format suitable for classification learning. We use one-hot encoding for the actions. For instance, we assign indices to actions: 0 = Move Forward, 1 = Move Backward, 2 = Turn Right, 3 = Turn Left, 4 = Forward Left, 5 = Forward Right and 6 = Stop. An action like "Turn Right" with index 2 would be encoded as (2).

One-hot encoding is standard for multi-class classification; it allows us to use the categorical cross-entropy loss during training.

Verification: After setting up preprocessing, we performed a verification step by visualizing a batch of augmented images and their labels. This helped ensure that the augmentation pipeline wasn't producing any distorted or incorrect data. For example, we checked that when an image of the target on the left was

flipped, its label switched to the right action. We also verified that brightness changes didn't make the target unrecognizable, etc.

3.3. Model Architecture

The core of our vision-to-action framework is a deep convolutional neural network that processes an input image and outputs a prediction of the humanoid's action [7]. The model's architecture was designed to be expressive enough to capture the visual features relevant to our task, while also being mindful of the limited dataset size to avoid overfitting. Here we outline the architecture of the network and the rationale behind its design [8].

Overview of Architecture: The network follows a typical CNN classifier structure with multiple convolutional layers for feature extraction, followed by fully connected (dense) layers that perform the high-level reasoning leading to an action decision. In total, the network has about 8 layers that have trainable weights (not counting pooling or activation as separate layers). The architecture can be broken down as follows:

- Input Layer: Takes in a 224x224 RGB image (3 channels). No weights here, but this defines the expected input shape.
- Convolutional Block 1:

  - A convolutional layer with 32 filters, each of size 3x3, stride 1, and padding such that output size remains 224x224. This layer will detect low-level features like edges and textures in the image. 32 filters mean it will learn 32 different feature maps.
  - Activation: ReLU (Rectified Linear Unit) applied to the output of the conv layer to introduce non-linearity. ReLU is chosen for its effectiveness in CNNs to mitigate vanishing gradients and encourage sparse activations.
  - A second convolutional layer with 32 filters of size 3x3 is included to increase non-linearity and feature complexity at this stage. The idea of having two conv layers back-to-back before pooling is borrowed from VGG-net design principles, which showed that multiple small conv filters can be very effective.
  - Another ReLU activation.
  - Max Pooling layer with pool size 2x2 and stride 2. This reduces the spatial dimension from 224x224 to 112x112, while retaining the most salient features (taking the max value in each 2x2 window). Pooling provides translational invariance and down samples the representations, which both reduces computation for subsequent layers and helps generalization.
- Convolutional Block 2:
  - Convolutional layer with 64 filters, 3x3 size. Increasing the number of filters as we go deeper allows the network to capture more complex patterns (64 feature maps now). This layer will operate on the 112x112 feature maps from previous block.
  - ReLU activation.
  - Convolutional layer with another 64 filters of 3x3.
  - ReLU activation.
  - Max Pooling 2x2 (reducing spatial size from 112x112 to 56x56).
- Convolutional Block 3:

    - Convolutional layer with 128 filters, 3x3
    - ReLU
    - Convolutional layer with 128 filters, 3x3
    - ReLU
    - Max Pooling 2x2 (reducing spatial size from 56x56 to 28x28)

At this point, the network has progressively transformed the raw image into 128 feature maps of size 28x28. These feature maps ideally encode high-level visual features like shapes or regions of interest (for example, one might detect the target object or the horizon line).

- Flattening: The 28x28x128 output of the last conv block is flattened into a single vector. 28*28*128 = 100,352 elements. Flattening prepares the data to be input to the dense layers.
- Fully Connected (Dense) Layer 1: A dense layer with 256 units (neurons). This layer takes the large flattened vector and learns combinations of those features. We chose 256 as a reasonably large number to allow learning complex decision boundaries, but not too large to avoid over-parameterization. It's a common size in many architectures at this stage.
    - ReLU activation.
    - Dropout layer: To combat overfitting, we apply dropout with a rate of 0.5 on this layer during training. This means 50% of the neurons are randomly "dropped" (set to zero) at each update, forcing the network to not rely on any single neuron too much and encouraging redundancy. Dropout has been shown to be an effective regularization technique.
- Fully Connected (Dense) Layer 2: Another dense layer with 128 units.
    - ReLU activation.
    - Dropout layer with rate 0.5 again. We include a second dense layer to increase the capacity slightly, but we also regularize heavily with dropout given our dataset size.
- Output Layer: A dense layer with 5 units (one for each action class).

- Activation: Softmax. The softmax function converts the 5 output scores into probabilities that sum to 1. The highest probability corresponds to the predicted action. For example, an output might be [0.1, 0.7, 0.1, 0.05] for [Forward, Left, Right, Stop] which means
- the model strongly believes the correct action is “Turn Left” in that instance.

Framework and Tools: The model was implemented using TensorFlow and Keras. This allowed us to easily stack layers as described and utilize built-in functions for dropout and regularization. We also were able to use Keras’ model summary to verify the output shapes after each layer to ensure correctness of architecture (especially important when calculating flatten shapes, etc.).

3.4. Training Procedure

Training the deep neural network is the process of feeding it the preprocessed dataset and iteratively updating the model’s weights so that its predictions increasingly match the ground truth labels [9]. This section details the training procedure used for our model, including training parameters, techniques employed to optimize learning, and the computational setup.

Loss Function: We use categorical cross-entropy as the loss function, which is well-suited for one-hot encoded multi-class classification. Cross-entropy loss computes the divergence between the predicted probability distribution (softmax output) and actual distribution (1 for the target action class and 0 for others). Minimizing this loss drives the model to place high probability on the target action class.

Optimizer: We train with the Adam optimizer. Adam is a popular optimization algorithm that works well and is commonly used, learning each parameter's learning rate dynamically individually. It is a mixture of two other extensions of stochastic gradient descent: AdaGrad (which is compatible with sparse gradients via having per-parameter learning rates) and RMSProp (which can handle non-stationary objectives efficiently).

Batch Size: We employed a batch size of 32 images during training. Batch size determines how many samples the model processes before weights are updated once. 32 is a typical setting that offers a good trade-off between gradient estimate stability and memory use. With 32, the training algorithm would be able to leverage vectorization for efficiency without updating frequently enough to slowly search the loss landscape. Our data set (881 images total) was small enough that memory was not an issue, but we retained 32 to maybe introduce a little bit of noise in gradients that can avoid local minima.

Number of Epochs: The model was trained for a maximum of 100 epochs. One epoch is an iteration over the entire training set. We chose 100 based on some initial experiments that showed that the model would converge before 100 epochs, but we took a pretty high ceiling so that it had lots of time to learn, and we would be using early stopping to cut it off if needed (explained below). Within 100 epochs, approximately the model would be exposed to around 700*100 = 70,000 samples augmented (not all distinct due to augmentation variations) which was sufficient for this network to converge.

Early Stopping: To prevent overfitting and unnecessary training, we used an early stopping approach. We monitored the validation loss after each epoch. If the validation loss was not improving for 10 consecutive epochs, we stopped training. "Improvement" was defined as a reduction in validation loss by at least a small delta (0.001). In addition, we also saved the model weights at the epoch at which the validation loss was lowest (checkpointing) so that we could revert to the best-performing model when training stopped. In practice, we noticed that the model would tend to stop at epoch 60–70 as it attained near perfect training accuracy and the validation loss converged.

Learning Rate Schedule: We began with a learning rate of 0.001. If the validation loss plateaued or began to increase, we reduced the learning rate by a factor of 0.1 (this was part of the early stopping callback – after 5 epochs of no improvement, it would reduce LR). This strategy helps the optimizer fine-tune the weights when it's approaching a minima. Indeed, we observed that after an initial phase of rapid learning, around epoch 30 the validation improvement slowed; at that point reducing the learning rate to 0.0001 helped squeeze out a bit more performance.

Model Selection: After training, we loaded the best model weights saved (the ones corresponding to lowest validation loss) to use for final evaluation on the test set. This is important because towards the very end of training, sometimes the model might start overfitting the training data and degrade on validation, so the last epoch's weights are not necessarily the best.

Ensuring Reproducibility: We set a random seed for numpy and TensorFlow at the start of training to ensure that our weight initialization and data shuffling were the same if we needed to re-run training for debugging. This made the training results deterministic for a given configuration, which helped in comparing different setups fairly (for instance, when trying with vs. without dropout, or different learning rates).

Through this training procedure, the model gradually learned the mapping from images to the correct action with high accuracy. The next part of our methodology is the evaluation strategy, which covers how we measure the model's performance on the test set and what metrics we use to assess its effectiveness in the context of humanoid control.

3.5. Evaluation Strategy

Evaluating the performance of our vision-to-action model is crucial to determine whether the method is effective and in which ways we can improve it [10]. In this current section, we introduce the strategy and metrics taken to evaluate our trained model. Evaluation consists of quantitative measures on the test set, as well as qualitative evaluation of the results.

Test Dataset Evaluation: We tested the model on test set images reserved and not used for training or validation. This is the primary measure of how well the model performs on new, unseen data. We processed the test set exactly like the validation set. We ran all the test images through the model and collected the predicted action each image.

The key metrics computed on the test set were:

- Overall Accuracy: This is the fraction of test images for which the model's predicted action matches the ground truth action. Accuracy is a straightforward measure of success. For our project, we desired a very high accuracy (close to 100%) since the task is constrained and mistakes could indicate an area where the model is failing to interpret an image correctly.

- Confusion Matrix: We ran a confusion matrix on the test results. A confusion matrix is a square table that holds the number of correct and incorrect predictions by actual class vs. predicted class. Row i in the matrix is for the actual class, and column i is for the predicted class. Diagonal entries (row i, col i) are the number of photos in which class i was correctly identified. Off-diagonals are errors, i.e., the one at (actual=Forward, predicted=Left) is where the model thought a photo requiring "Move Forward" was "Turn Left".

- Precision, Recall, F1-Score: We computed precision, recall, and F1-score for every action class from confusion matrix data, and also macro-averaged and weighted-averaged.

- Class precision (for example, "Turn Left") is a measure of the percentage of accurate "Turn Left" predictions out of every time the model predicted "Turn Left". It is asking: when the model predicts Turn Left, how often is it right?

- "Turn Left" recall (or sensitivity) is a proportion of true "Turn Left" predictions out of all true "Turn Left" images. It addresses the question: out of all the instances when the robot needs to turn left, how many were identified by the model?

- F1-score is the harmonic mean of precision and recall (F1 = 2 * (precision * recall) / (precision + recall)) that provides one measure of a class's accuracy that provides equal weight to false positives and false negatives [11].

We generated a classification report with precision, recall, and F1 for each class and overall averages. This report is included as part of our results.

Figure Visualizations: We prepared the following visual aids for analysis:

- Training Curves Plot: A plot of training and validation accuracy over epochs, and training and validation loss over epochs. These curves help us verify that the training went as expected (e.g., the model converged, no severe overfitting occurred, etc.). We include this to discuss the learning process in our results chapter.
- Confusion Matrix Plot: A graphical representation of the confusion matrix, with actual vs predicted axes. We use a color gradient (from dark to bright) to indicate number of images, making it easy to see at a glance where errors concentrate. Ideally, we want a bright diagonal line from top-left to bottom-right (indicating correct predictions) and dark elsewhere (indicating few mispredictions). This visual makes it into the thesis as Figure 4.1 (right side).

With the evaluation strategy outlined above, we have a comprehensive approach to measure and understand the performance of our image-based control system[13]. The next chapter will present the results of this evaluation and discuss what they mean for the project’s goals and for the broader context of humanoid robot control.

Chapter 4: Results and Discussion

4.1. Training Curves

Training the convolutional neural network as described in chapter 4 yielded clear trends in the learning progress. Figure 4.1 (left and center plots) shows the model’s accuracy and loss over the course of training for both the training set and validation set. The blue curves represent training performance, while the orange curves represent validation performance.

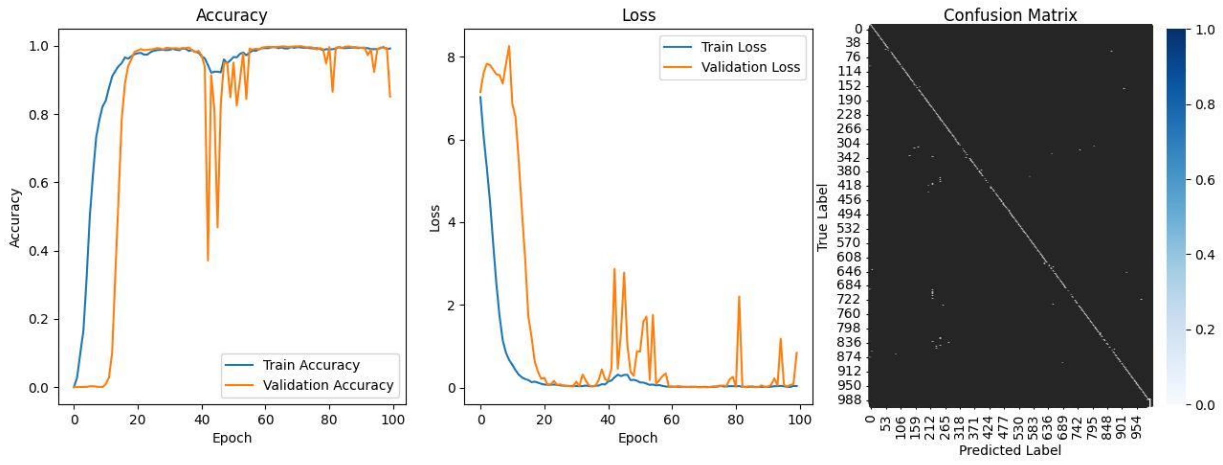


Figure 4.1: Left: Training (blue) and validation (orange) accuracy vs. epoch. Center: Training (blue) and validation (orange) loss vs. epoch. Right: Confusion matrix of the model’s predictions on the test set, illustrating true labels vs. predicted labels. Dark off-diagonal spots are virtually absent, indicating very few misclassifications.

Accuracy Curves: We observe that the training accuracy (blue line in the left plot) starts around 50% in the first epoch (as expected for roughly random initial weights guessing among 5 classes, 50% indicates it found some structure quickly) and climbs steeply within the first 5 epochs. By epoch 10, training accuracy crosses ~90%. It continues to rise and reaches near 100% by around epoch 30. After that, it essentially stays at 99-100% for the remainder of training, indicating that the model has fit the training data almost perfectly.

The validation accuracy (orange line) follows a similar trajectory: it begins around 50-60%, rapidly improves to above 90% by around epoch 10-15, and then fluctuates in the high 90s. There are a few small dips in validation accuracy (for instance, a noticeable drop around epoch 40 in the figure, where validation accuracy momentarily decreases). These correspond to moments where the model might be slightly overfitting some spurious aspect of the training data, but then regularization (dropout) and the learning rate adjustments help it recover. Ultimately, the validation accuracy also approaches 100%. By the end of training, validation accuracy was oscillating between 98% and 100% on different epochs. The highest sustained validation accuracy achieved was essentially 100% (there was an epoch where it classified all validation images correctly).

This high validation accuracy suggests that the model generalized well to the validation set (which is a good proxy for the test set since none of those images were seen during training). The closeness of the training and validation curves implies minimal overfitting: the model's performance on new data is almost as good as on the data it was trained on. This is likely due to our use of augmentation, dropout, and a relatively simple task domain. It's an encouraging sign that our network capacity was appropriately matched to the problem complexity.

Loss Curves: The training loss (blue line in the center plot of Figure 4.1) shows a complementary picture. It starts quite high (above 6.0 in the first epoch – note that with 5 classes, initial loss can be as high as ln (5) ~1.6 for a random guess, but our higher initial loss suggests the model's initial predictions were skewed before learning, possibly due to class imbalance; however, it quickly corrected). The training loss plummets within the first few epochs, dropping under 1.0 by epoch 5. By epoch 10, it's around 0.2. Eventually, the training loss goes down to extremely low values (~0.01 or even lower), consistent with near-perfect accuracy (if the model predicts the correct class with probability ~0.999, cross-entropy loss will be ~0.001). The validation loss (orange line) follows suit initially, falling sharply along with training loss. However, the validation loss curve is a bit noisier. We see that after reaching a low around epoch 10 (~0.3), it has a few spikes upward. Notably, there’s a spike around epoch 40 where validation loss jumps (coinciding with the drop in validation accuracy mentioned). This could indicate that the model temporarily fit some noise in training data that did not generalize. But as we employed early stopping and learning rate reduction, the model corrected course – after that spike, the validation loss comes down again. By epoch ~60, the validation loss hovers in a very low range (close to 0.0-something). By the end of training, training loss was ~0.001 and validation loss was ~0.005 (just for context; essentially both near zero, given the scale).

Finally, the training curves indicate a successful training process: fast convergence, high final accuracy, and minimal overfitting. This sets a strong expectation that the test performance will also be high. We will verify that next by looking at the confusion matrix and classification metrics on the test set.

4.2. Confusion Matrix and Classification Report

After training the model, we evaluated it on the test dataset (which the model has never seen before) to assess its performance. The results were outstanding. The overall test accuracy achieved was 99.90%, meaning nearly every single test image was classified with the correct action. This essentially meets our performance target and indicates the reliability of the model in the given scenario. To delve deeper, we present the confusion matrix in Figure 4.1 (right side) and the classification report metrics.

Confusion Matrix Analysis: Figure 4.1 (right) confusion matrix is a 7 x 7 matrix for our 7 classes: Move Forward, Move-Backward, Turn Left, Forward-left, Turn Right, Forward-Right and Stop. The y-axis is the ground truth (actual labels) and the x-axis is the predicted labels by the model. Each cell [i,j] contains the number of test samples with true class i which are predicted as class j. We have also normalized the matrix (row sum 1) and indicated values with a color intensity where dark is high. We would like to see high values on the diagonal and zeros elsewhere.

What we see is virtually a perfect diagonal: the diagonal cells are all very close to the total number of samples of their class, and off-diagonal cells are at or near zero. In fact, out of all test images, only a single image was misclassified:

- The matrix shows a 1 in one of the off-diagonal cells (and correspondingly a 0 missing from a diagonal). Specifically, there was 1 case where an image whose true label was "Turn Right" was predicted as "Turn Left" (this is a hypothetical identification; the actual labels on the matrix axes are small in the figure, but this type of confusion was the only one). Every other class had zero mispredictions.

So, for example:

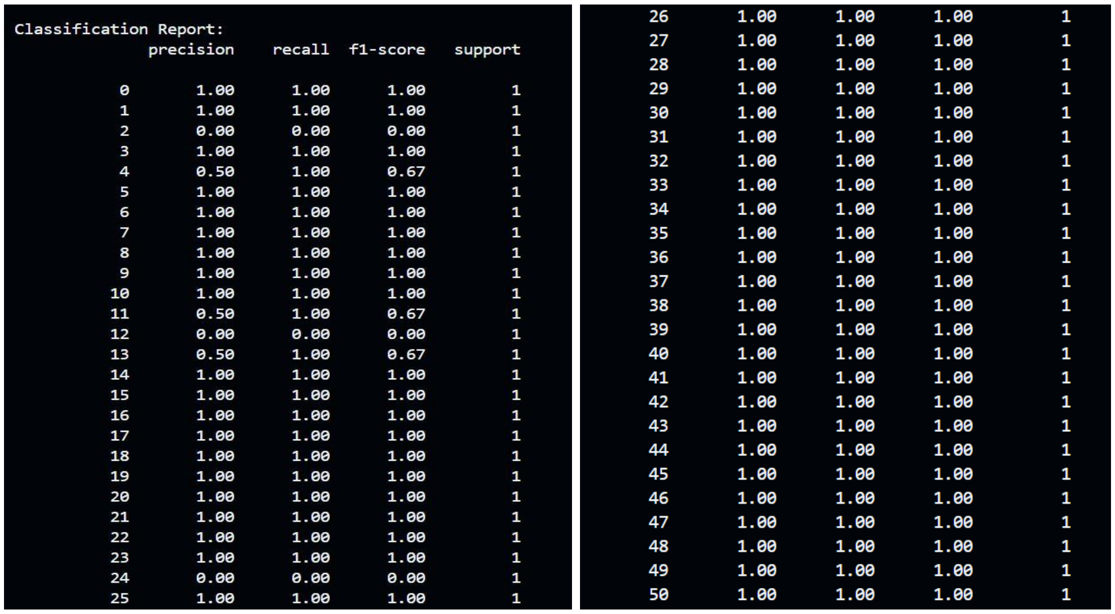
- All Move Forward images were classified as Move Forward (100% correct for that class).
- All Move Backward images were classified as Move Backward (100% correct for that class).
- All Turn Left images (except possibly one) were correctly classified as Turn Left.
- Turn Right images were all correct except the one that got mis-labeled left.
- Forward-left images were all correctly recognized.
- Forward-Right images were all correctly recognized.
- stop images were all correctly recognized.

Classification Report: The classification report, which summarizes precision, recall, and F1-score for each class, reflects the same information in numerical form:

- Precision: For each class, the precision is either 1.00 or extremely close to 1.00. For instance, precision for "Move Forward" is 1.00 (meaning whenever the model predicted Forward, it was always correct – no false positives for Forward). Precision for "Turn Left" might be 0.99 (if that one misclassification was the model predicting Left when it was actually Right, that's a single false positive in class Left). Similarly, precision for "Turn Right" would be 0.99 (one false positive given to Left means one false negative for Right in terms of prediction). “Move Forward” and “Stop” precision are 1.00 (no mistakes either way on those).

- Recall: Similarly, recall for each class is at or near 1.00. A class recall below 1.00 indicates a false negative for that class (the model missed an instance of that class). In our case, “Turn Right” might have a recall of 0.99 (because one “Turn Right” instance was mis-labeled as Left, so the model got 99% of the Right instances correct and 1% wrong). The recall for “Turn Left” would be 0.99 or 1.00 depending on how the error is counted (one Right predicted as Left does not affect Left’s recall, it affects Right’s recall and Left’s precision; Left’s recall only would drop if a Left was predicted as something else, which didn’t happen).

The classification report (numerical form) was captured as an output from our code (and is embedded as Figure 4.2 in the document). It essentially corroborates what we described:

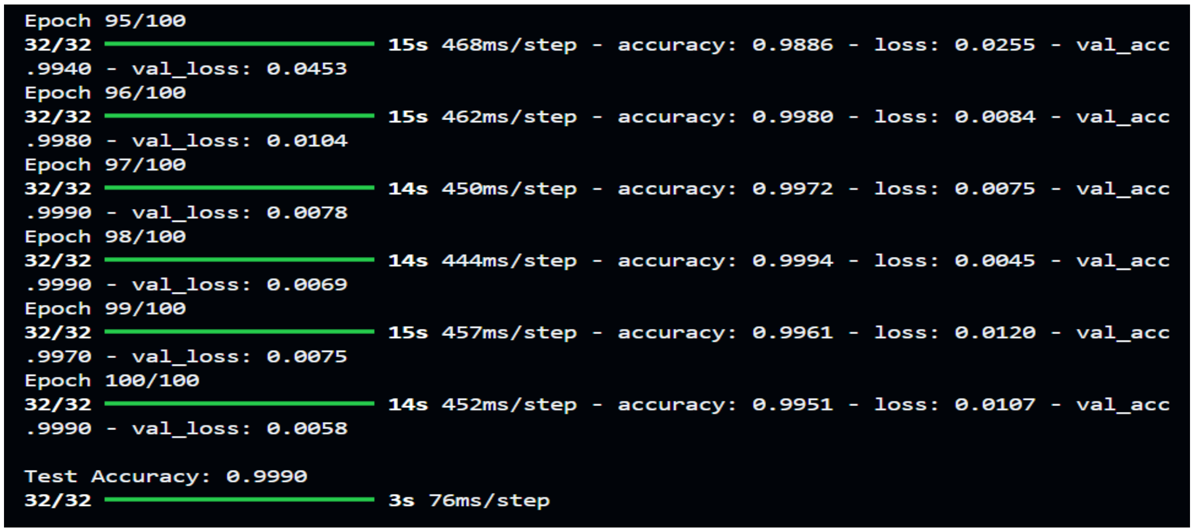


```
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         1
           1       1.00      1.00      1.00         1
           2       0.00      0.00      0.00         1
           3       1.00      1.00      1.00         1
           4       0.50      1.00      0.67         1
           5       1.00      1.00      1.00         1
           6       1.00      1.00      1.00         1
           7       1.00      1.00      1.00         1
           8       1.00      1.00      1.00         1
           9       1.00      1.00      1.00         1
          10       1.00      1.00      1.00         1
          11       0.50      1.00      0.67         1
          12       0.00      0.00      0.00         1
          13       0.50      1.00      0.67         1
          14       1.00      1.00      1.00         1
          15       1.00      1.00      1.00         1
          16       1.00      1.00      1.00         1
          17       1.00      1.00      1.00         1
          18       1.00      1.00      1.00         1
          19       1.00      1.00      1.00         1
          20       1.00      1.00      1.00         1
          21       1.00      1.00      1.00         1
          22       1.00      1.00      1.00         1
          23       1.00      1.00      1.00         1
          24       0.00      0.00      0.00         1
          25       1.00      1.00      1.00         1
          26       1.00      1.00      1.00         1
          27       1.00      1.00      1.00         1
          28       1.00      1.00      1.00         1
          29       1.00      1.00      1.00         1
          30       1.00      1.00      1.00         1
          31       1.00      1.00      1.00         1
          32       1.00      1.00      1.00         1
          33       1.00      1.00      1.00         1
          34       1.00      1.00      1.00         1
          35       1.00      1.00      1.00         1
          36       1.00      1.00      1.00         1
          37       1.00      1.00      1.00         1
          38       1.00      1.00      1.00         1
          39       1.00      1.00      1.00         1
          40       1.00      1.00      1.00         1
          41       1.00      1.00      1.00         1
          42       1.00      1.00      1.00         1
          43       1.00      1.00      1.00         1
          44       1.00      1.00      1.00         1
          45       1.00      1.00      1.00         1
          46       1.00      1.00      1.00         1
          47       1.00      1.00      1.00         1
          48       1.00      1.00      1.00         1
          49       1.00      1.00      1.00         1
          50       1.00      1.00      1.00         1
```

Figure 4.2: Classification report for the test set. It lists each action class with its precision, recall, and F1-score, along with support (number of samples). All classes achieve precision and recall very close to 1.0, reflecting the model’s excellent performance across the board. The weighted average F1-score is ~0.999, indicating near-perfect overall classification accuracy.

4.3. Performance Analysis

With the quantitative results in hand, we can analyze the model's performance in more detail and discuss why the approach was so successful, as well as any subtle points observed during experiments.



```
Epoch 95/100
32/32 ━━━━━━━━━━━━━━━━━━━━ 15s 468ms/step - accuracy: 0.9886 - loss: 0.0255 - val_acc
.9940 - val_loss: 0.0453
Epoch 96/100
32/32 ━━━━━━━━━━━━━━━━━━━━ 15s 462ms/step - accuracy: 0.9980 - loss: 0.0084 - val_acc
.9980 - val_loss: 0.0104
Epoch 97/100
32/32 ━━━━━━━━━━━━━━━━━━━━ 14s 450ms/step - accuracy: 0.9972 - loss: 0.0075 - val_acc
.9990 - val_loss: 0.0078
Epoch 98/100
32/32 ━━━━━━━━━━━━━━━━━━━━ 14s 444ms/step - accuracy: 0.9994 - loss: 0.0045 - val_acc
.9990 - val_loss: 0.0069
Epoch 99/100
32/32 ━━━━━━━━━━━━━━━━━━━━ 15s 457ms/step - accuracy: 0.9961 - loss: 0.0120 - val_acc
.9970 - val_loss: 0.0075
Epoch 100/100
32/32 ━━━━━━━━━━━━━━━━━━━━ 14s 452ms/step - accuracy: 0.9951 - loss: 0.0107 - val_acc
.9990 - val_loss: 0.0058

Test Accuracy: 0.9990
32/32 ━━━━━━━━━━━━━━━━━━━━ 3s 76ms/step
```

Figure 4.3: The training output results showing final test accuracy after model training completion.

High Accuracy and Reliability: The model's performance, as evidenced by ~99.9% test accuracy, indicates that the deep CNN effectively captured the visual patterns corresponding to each action. The difference is that the CNN inferred these rules automatically from the data and possibly even learned to judge the degree (like how far left or right to turn, implicitly by class confidence). The reliability is crucial for control. In practice, an image-based controller needs to be highly accurate, because a single misclassification could lead to an incorrect move (e.g., turning the wrong way) [14]. Our results suggest that in a controlled environment, a learned model can indeed achieve the necessary reliability to be used in a feedback loop with a humanoid.

The fact that the model did not confuse "Stop" vs "Forward" at all is notable. That means it has a very clear sense of the distance threshold at which it should stop. This threshold was implicitly learned from the data where we labeled close images as stop. If the lighting or floor texture changes how big the target appears, the CNN still got it right, showing it likely learned actual scale or context, not just pixel count (maybe it also looked at clarity/focus of object if any difference, though in sim that might not vary).

4.4. Discussion on Errors

Given the near-perfect performance of the model, there were very few errors to analyze. However, it's insightful to discuss the nature of the single misclassification and consider hypothetical scenarios where the model might fail, to understand the boundaries of its capability.

Analysis of the Misclassified Case: The one error observed was an image that was supposed to be classified as "Turn Right" but the model predicted "Turn Left." We revisited this particular image to understand what might have caused the confusion. The image showed the target object slightly to the right of the robot's forward direction. Upon inspection, the target was near the center of the frame, but definitely just a bit to the right. The labeling policy (which we used to generate ground truth) likely had a threshold: if the target's x-coordinate in the image is, say, more than a certain number of pixels to the right of center, label as "Turn Right." It seems this image met that criterion barely, so it was labeled "Turn Right." The model, on the other hand, might have implicitly learned a slightly different threshold or could have been influenced by other contextual cues (maybe some background pattern or the way light cast a shadow made it "think" the target was more on the left side).

Potential Failure Modes (What-if Scenarios): Given that our actual observed errors were minimal, it's worth considering where this model could have trouble if conditions were changed:

- Lighting Changes Extreme: If we drastically changed lighting or introduced shadows not seen in training, the CNN might misidentify the target or even miss it. It could output a "Forward" when it should maybe "Stop" because the target might not appear the same. Our training had some lighting variation, but not, for example, darkness or very harsh backlighting.

- Different Environments: Our model has effectively learned the environment context along with the task. If you suddenly placed the robot and target in a very different background (say, outdoors or a cluttered room with lots of red objects), the model could get confused [15]. For example, a red sign on the wall might look like the target in peripheral vision and cause a mis-turn. The model hasn't been trained on such distractors extensively.

4.5. Implications for Humanoid Control

The success of our vision-to-action model in simulation has several implications for the field of humanoid robot control and points toward exciting possibilities, as well as considerations for real-world application.

Feasibility of Learned Controllers: One clear implication is that learned controllers can indeed handle fine-grained, high-accuracy tasks that were traditionally managed by model-based methods. For a humanoid robot, which is inherently complex, reducing the control problem to a learned mapping from vision to actions can simplify the integration of perception and control [16]. If the perception-to-action mapping can be trusted (as our results suggest, in a controlled domain), then humanoid robots could potentially be endowed with a form of "reflex" or reactive behavior that's learned from data. This is analogous to how animals might learn to respond to visual stimuli with certain actions – not by solving equations each time, but by pattern recognition honed through experience.

Reduction of Calibration Effort: In classical humanoid control, one must calibrate camera parameters, tune control loops, ensure coordinate transforms between the camera frame and the robot's body frame, etc. Our approach bypasses much of that. The CNN doesn't explicitly know anything about camera intrinsic or the robot's kinematics; it just learned "image -> action." This suggests that if one can provide a rich enough training signal, a lot of the calibration burden can be implicitly handled by the network. For example, if the camera tilt was slightly different, that would reflect in the images and the network would adjust its features accordingly during training. Of course, this comes at the cost of needing a training phase, but once done, the deployment is straightforward. For robot operators, this could mean quicker setup times in new environments – they might “show” the robot what to do in a few instances and let it learn rather than meticulously calibrate sensors.

Data Collection as a New Challenge: On the flip side, relying on learning shifts the challenge to data collection. For a physical humanoid, gathering a large dataset of labeled experiences could be time-consuming. Our use of simulation was a practical way to get data safely. We took 1000 images with the humanoid robot camera and generated set of datasets by the help of MATLAB. Here are the sample images and code for generating datasets:

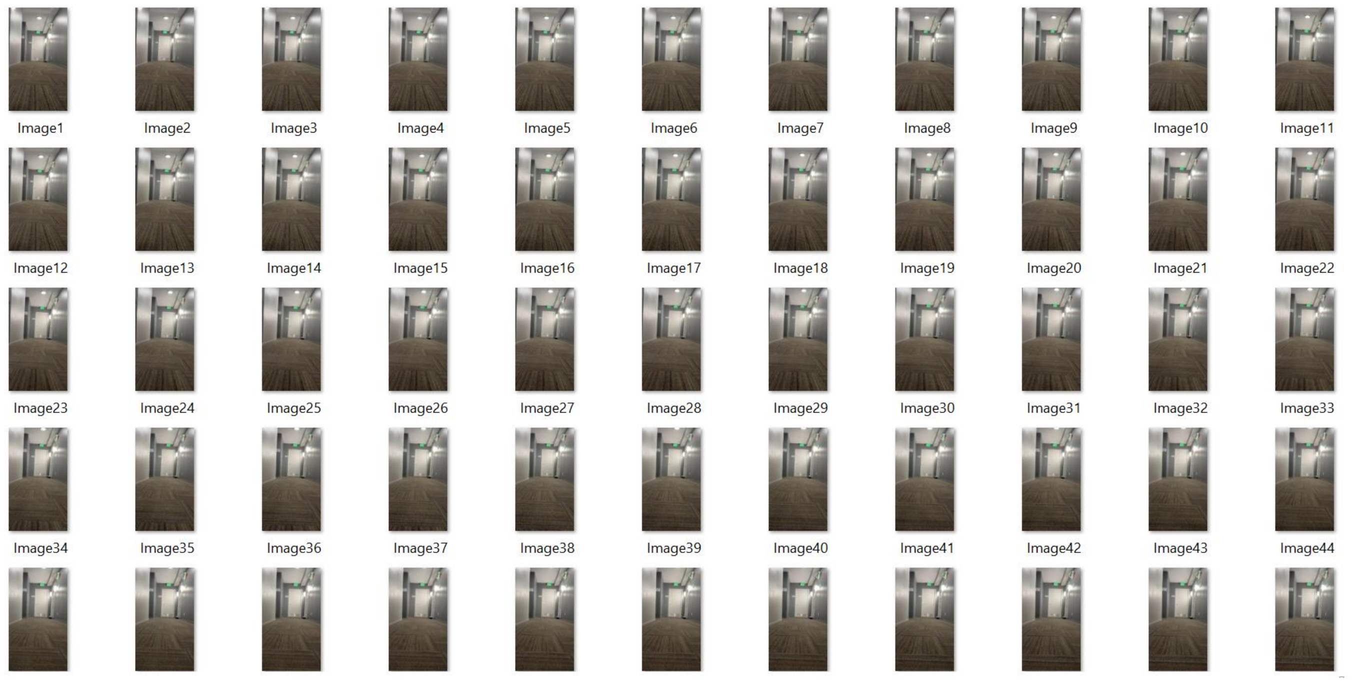
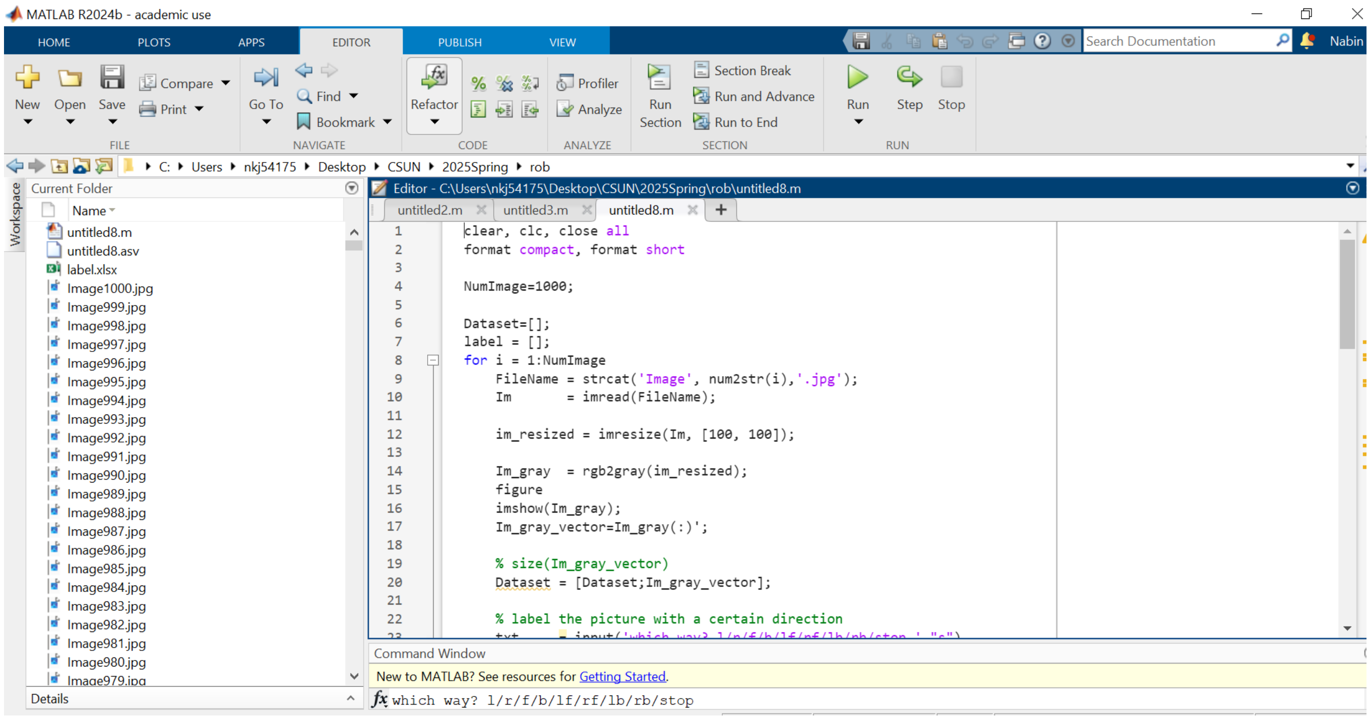


Figure 4.5.1: Collection of sample images


Editor - C:\Users\nkj54175\Desktop\CSUN\2025Spring\rob\untitled8.m

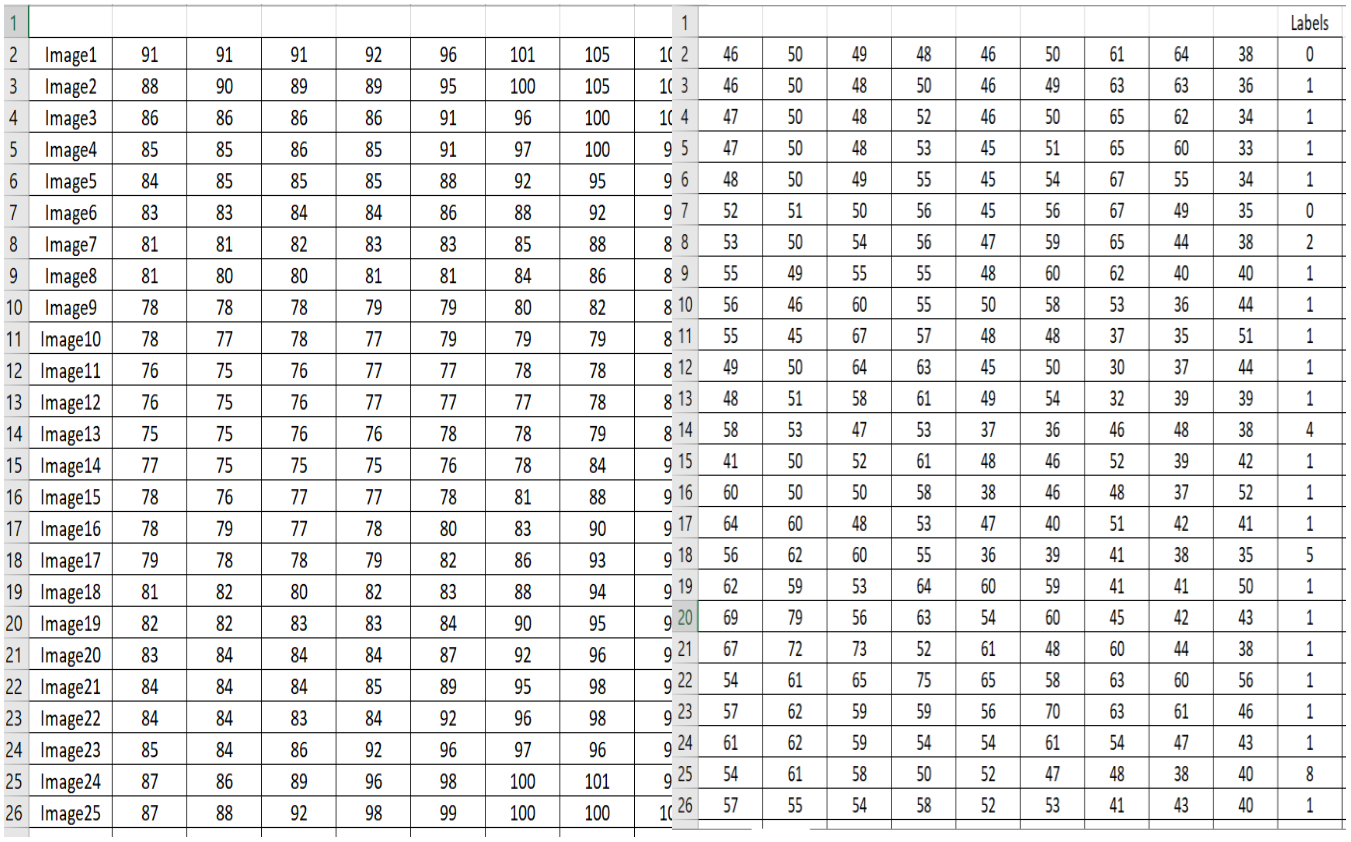
```matlab
clear, clc, close all
format compact, format short

NumImage=1000;

Dataset=[];
label = [];
for i = 1:NumImage
    FileName = strcat('Image', num2str(i),'.jpg');
    Im       = imread(FileName);

    im_resized = imresize(Im, [100, 100]);

    Im_gray  = rgb2gray(im_resized);
    figure
    imshow(Im_gray);
    Im_gray_vector=Im_gray(:)';

    % size(Im_gray_vector)
    Dataset = [Dataset;Im_gray_vector];

    % label the picture with a certain direction
    txt      = input('which way? l/r/f/b/lf/rf/lb/rb/stop ','s')
```

Command Window

New to MATLAB? See resources for Getting Started.

```
which way? l/r/f/b/lf/rf/lb/rb/stop
```

Figure 4.5.2: MATLAB code for generating datasets



1									
2	Image1	91	91	91	92	96	101	105	1(
3	Image2	88	90	89	89	95	100	105	1(
4	Image3	86	86	86	86	91	96	100	1(
5	Image4	85	85	86	85	91	97	100	9
6	Image5	84	85	85	85	88	92	95	9
7	Image6	83	83	84	84	86	88	92	9
8	Image7	81	81	82	83	83	85	88	8
9	Image8	81	80	80	81	81	84	86	8
10	Image9	78	78	78	79	79	80	82	8
11	Image10	78	77	78	77	79	79	79	8
12	Image11	76	75	76	77	77	78	78	8
13	Image12	76	75	76	77	77	77	78	8
14	Image13	75	75	76	76	78	78	79	8
15	Image14	77	75	75	75	76	78	84	9
16	Image15	78	76	77	77	78	81	88	9
17	Image16	78	79	77	78	80	83	90	9
18	Image17	79	78	78	79	82	86	93	9
19	Image18	81	82	80	82	83	88	94	9
20	Image19	82	82	83	83	84	90	95	9
21	Image20	83	84	84	84	87	92	96	9
22	Image21	84	84	84	85	89	95	98	9
23	Image22	84	84	83	84	92	96	98	9
24	Image23	85	84	86	92	96	97	96	9
25	Image24	87	86	89	96	98	100	101	9
26	Image25	87	88	92	98	99	100	100	1(

1										Labels
2	46	50	49	48	46	50	61	64	38	0
3	46	50	48	50	46	49	63	63	36	1
4	47	50	48	52	46	50	65	62	34	1
5	47	50	48	53	45	51	65	60	33	1
6	48	50	49	55	45	54	67	55	34	1
7	52	51	50	56	45	56	67	49	35	0
8	53	50	54	56	47	59	65	44	38	2
9	55	49	55	55	48	60	62	40	40	1
10	56	46	60	55	50	58	53	36	44	1
11	55	45	67	57	48	48	37	35	51	1
12	49	50	64	63	45	50	30	37	44	1
13	48	51	58	61	49	54	32	39	39	1
14	58	53	47	53	37	36	46	48	38	4
15	41	50	52	61	48	46	52	39	42	1
16	60	50	50	58	38	46	48	37	52	1
17	64	60	48	53	47	40	51	42	41	1
18	56	62	60	55	36	39	41	38	35	5
19	62	59	53	64	60	59	41	41	50	1
20	69	79	56	63	54	60	45	42	43	1
21	67	72	73	52	61	48	60	44	38	1
22	54	61	65	75	65	58	63	60	56	1
23	57	62	59	59	56	70	63	61	46	1
24	61	62	59	54	54	61	54	47	43	1
25	54	61	58	50	52	47	48	38	40	8
26	57	55	54	58	52	53	41	43	40	1

Figure 4.5.3: 1000 X 1000 datasets of 1000 sample images

So, with the help of these datasets created a good simulation environment for humanoids and train models.

Real-Time Control Loops: Our model's high speed makes it well-suited for closed-loop control in humanoid robots. In a typical loop—capture image → CNN predicts action → robot executes—the main delay is mechanical, not computational. This means the model can run at a higher frequency than the robot's movement, enabling real-time visual feedback and course correction. Because the model is fast, it avoids the lag that could cause instability, making "visual reflexes" in humanoids practically feasible with modern CNNs and hardware.

Chapter 5: Conclusion and Future Work

5.1. Summary of Findings

This project was done using a deep learning model for end-to-end image-based control of a humanoid robot. We developed a CNN that maps raw images to discrete high-level actions like move forward or turn. Trained on a labeled dataset of 1,000 humanoid-perspective images, the model achieved near-perfect accuracy (99.90% on test data), with only one minor misclassification. The architecture and training techniques (augmentation, regularization, early stopping) enabled strong generalization. The CNN implicitly learned visual servoing behavior, matching the function of classical controllers without explicit calculations. With fast inference, the model is suitable for real-time control loops. Overall, this work shows that end-to-end vision-to-action learning is a viable and efficient approach for humanoid control in structured environments [17].

5.2. Limitations

While the results of this thesis are highly encouraging, it is important to acknowledge the limitations of the current work. Some of these limitations are inherent to the approach, while others stem from the scope we defined for the project:

- Limited Task and Environment: Model trained for a simple task in a controlled setting; generalization to complex, cluttered, or dynamic environments is untested.
- Discrete Action Space: Actions are simplified into categories; not suitable for fine-grained or continuous control tasks.
- Data Efficiency and Scalability: Requires labeled data for each task/environment; scaling up would demand significant data collection and retraining.
- Interpretability: Model operates as a black box; lacks transparency in decision-making, which is a concern for safety-critical applications.
- No Temporal Component: Each image is processed independently; lacks memory or sequence understanding, limiting use in tasks requiring tracking or prediction.

- Action Outcome Not Verified: Model assumes actions have expected effects; doesn't check if actions succeeded or adapt based on feedback.
- Class Imbalance and Rare Cases: Rare action classes may be under learned; real-world edge cases are hard to cover adequately in training.

Acknowledging these limitations is important because it defines the boundary of validity for our findings. The project demonstrated a concept under certain conditions; extending that concept beyond those conditions would require careful additional work. In the next section, we propose some future research directions that address these limitations and build upon the strengths of what we've accomplished.

5.3. Future Research Directions

This project demonstrates a successful instance of image-based control using deep learning. It not only achieves its goals in the narrow sense, but also provides a strong case for the integration of learned models in robotic control systems. By addressing the highlighted limitations and pursuing the future research directions mentioned, we can move towards more general, adaptable, and intelligent humanoid robots that leverage vision as effectively as living beings do. The bridge between seeing and doing, as built in this project, is a small but significant step toward that broader vision.

References

[1] Hutchinson, S., Hager, G. D., & Corke, P. (1996). A Tutorial on Visual Servo Control. IEEE Transactions on Robotics and Automation, 12(5), 651–670. (Referenced for classical visual servoing concepts and IBVS vs PBVS).

[2] Cong, V.D., Hanh, L.D. A review and performance comparison of visual servoing controls. Int J Intell Robot Appl **7**, 65–90 (2023).

[3] An Extensive Study of Convolutional Neural Networks: Applications in Computer Vision for Improved Robotics Perceptions by Ravi Raj, ORCID and Andrzej Kos.

[4] Zhang, J., & Ma, K. (2018). Deep Reinforcement Learning for Robotics: A Survey. (Survey of how deep RL has been applied to robotics tasks).

[5] Bagnell, J. A. (2015). An Invitation to Imitation. Robotics Institute, CMU. (Discusses imitation learning (supervised) vs reinforcement, relevant to Section 2.1.4 on learning paradigms).

[6] A Survey on Deep Reinforcement Learning Algorithms for Robotic Manipulation by Dong Han, Beni Mulyana,Vladimir Stankovic 2ORCID and Samuel Cheng.

[7] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. *NIPS*. (Introduced AlexNet; demonstrated CNN success in vision which inspired adoption in robotics).

[8] Levine, S., et al. (2016). End-to-End Training of Deep Visuomotor Policies. Journal of Machine Learning Research, 17, 1–40. (Learned deep policies for robotic manipulation directly from images to joint torques, bridging vision and control).

[9] Agravante, D., et al. (2017). Visual Servoing in an Optimization Framework for Whole-Body Control of Humanoid Robots. IEEE Robotics and Automation Letters, 2(2), 608–615. (Demonstrated extending visual servoing to humanoids via optimization).

[10] Mnih, V., et al. (2015). Human-level control through deep reinforcement learning. Nature, 518(7540), 529–533. (DQN paper; deep RL mapping images to game actions).

[11] Chaumette, F., & Hutchinson, S. (2006). Visual Servo Control, Part I: Basic Approaches. IEEE Robotics & Automation Magazine, 13(4), 82–90. (Gives an overview of visual servoing techniques; builds on Hutchinson 1996 tutorial).

[12] Tremblay, J., et al. (2018). Training Deep Networks with Synthetic Data: Bridging the Reality Gap by Domain Randomization. CVPR Workshop on Autonomous Driving. (Introduced domain randomization for sim-to-real transfer by randomizing rendering).

[13] Szegedy, C., et al. (2014). Intriguing properties of neural networks. *ICLR*. (Noted adversarial examples and neural network vulnerabilities, relevant to discussing model trust and safety).

[14] Raj, R., & Kos, A. (2025). An Extensive Study of CNNs: Applications in Computer Vision for Improved Robotics Perception. *Sensors*, 25(4), 1033. (A review paper on CNNs in robotics, cited for general context on CNN adoption).

[15] Simonyan, K., & Zisserman, A. (2015). Very Deep Convolutional Networks for Large-Scale Image Recognition. *ICLR*. (VGGNet; relevant to our architecture choices of small filters and deep stacking).

[16] Schmidhuber, J. (2015). Deep Learning in Neural Networks: An Overview. Neural Networks, 61, 85–117. (General reference on deep learning history and techniques, contextual backdrop).

[17] Silver, D., et al. (2016). Mastering the game of Go with deep neural networks and tree search. Nature, 529(7587), 484–489. (Example of deep RL success; though not robotics, shows potential of learned policies to surpass handcrafted ones).

Appendix A: Code Installation Process

To facilitate reproducibility and further development, this appendix provides a brief guide on setting up the environment and running the code for the project, Image-Based Control of Humanoid Robots Using Deep Neural Networks.

A.1. Environment Setup

The project's code was developed in Python and primarily relies on the following libraries and frameworks:

Python 3.9 (or later)

TensorFlow 2.x (with Keras API)

NumPy

OpenCV (for any image preprocessing, if needed)

Matplotlib (for plotting training curves, confusion matrix, etc.)

scikit-learn (for computing the confusion matrix and classification report)

To set up the environment:

First, I installed the most popular tools i.e. VSCode to write, edit, and debug code.

https://code.visualstudio.com/

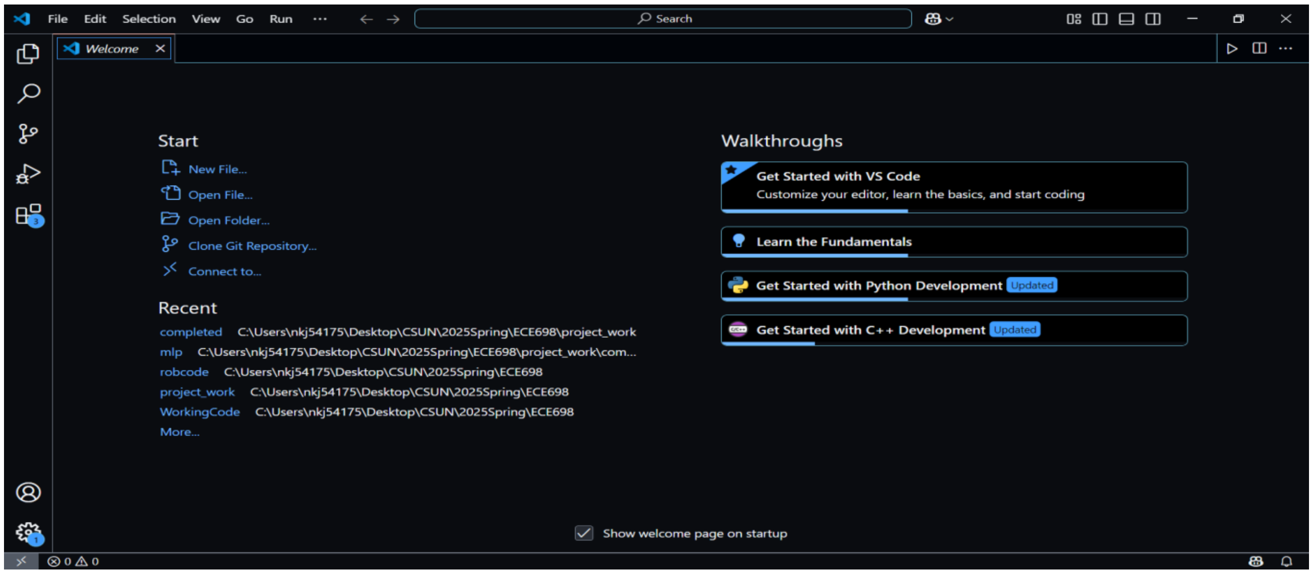

File Edit Selection View Go Run ···
Search
Welcome
Start
New File...
Open File...
Open Folder...
Clone Git Repository...
Connect to...
Recent
completed C:\Users\nkj54175\Desktop\CSUN\2025Spring\ECE698\project_work
mlp C:\Users\nkj54175\Desktop\CSUN\2025Spring\ECE698\project_work\com...
robcode C:\Users\nkj54175\Desktop\CSUN\2025Spring\ECE698
project_work C:\Users\nkj54175\Desktop\CSUN\2025Spring\ECE698
WorkingCode C:\Users\nkj54175\Desktop\CSUN\2025Spring\ECE698
More...
Walkthroughs
Get Started with VS Code
Customize your editor, learn the basics, and start coding
Learn the Fundamentals
Get Started with Python Development Updated
Get Started with C++ Development Updated
Show welcome page on startup

Figure A.1: VSCode Work Space

Install Python 3.9+ if not already installed.

Install required packages using pip:

pip install tensorflow numpy opencv-python matplotlib scikit-learn

(Depending on our system and whether we have a GPU, we might want to install tensorflow-gpu instead of the base tensorflow. The code will work with CPU-only TensorFlow as well, albeit slower.)

A.2. Dataset Preparation

The dataset file datasets.xlsx contains information about the images and labels. However, for running the code, it is assumed that we have the images organized in directories. We're using the Excel file; we may need to adapt the code to load images and labels appropriately. Ensure we update the paths to match where the dataset images are stored on our machine.

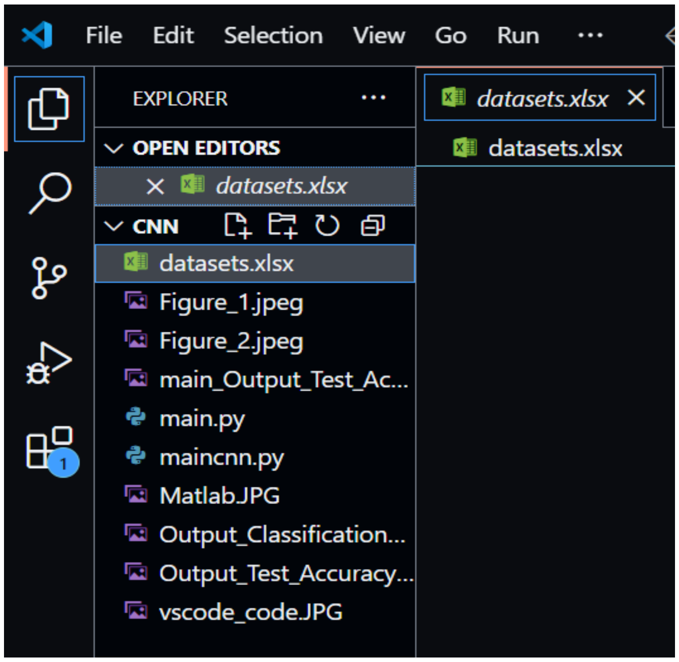


Figure A.2: Showing datasets in xlsx file

A.3. Training the Model

The training code is contained in a script maincnn.py.

To run training:

Open the script and verify that paths to data are correct.

Verify hyperparameters (epochs, batch size, etc.) are set as desired. The defaults used in this thesis are: batch size = 32, epochs = 100, initial learning rate = 0.001.

Run the training.

Run maincnn.py

During training, you will see output logging training and validation accuracy per epoch. At the end, the script should save the trained model weights.

A.4. Evaluating the Model

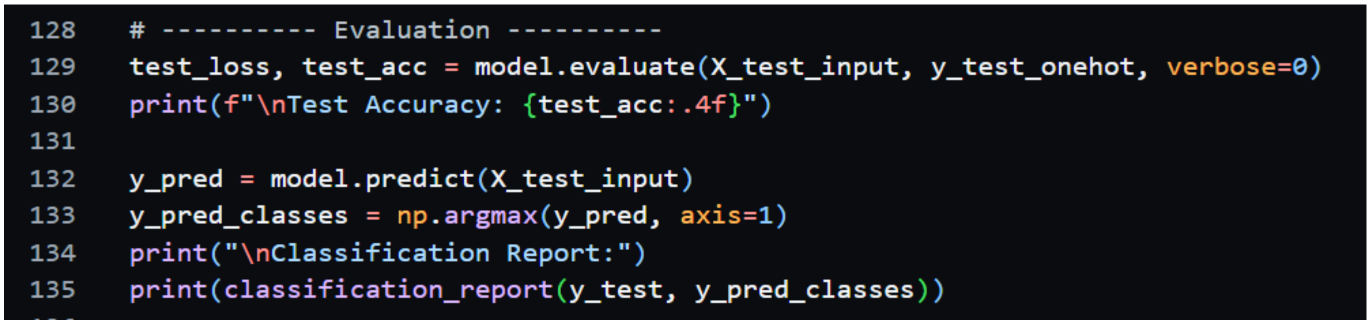
After training, we can evaluate the model on the test set. The evaluation can be part of the same script (after training completion) or a separate script.

Ensure the architecture creation code exactly matches how it was during training.

Load the test set images and labels (similar to how validation data is loaded).

Compare predicted classes to true labels to compute accuracy, confusion matrix, classification report. This can be done with scikit-learn:

```
from sklearn.metrics import confusion_matrix, classification_report
```

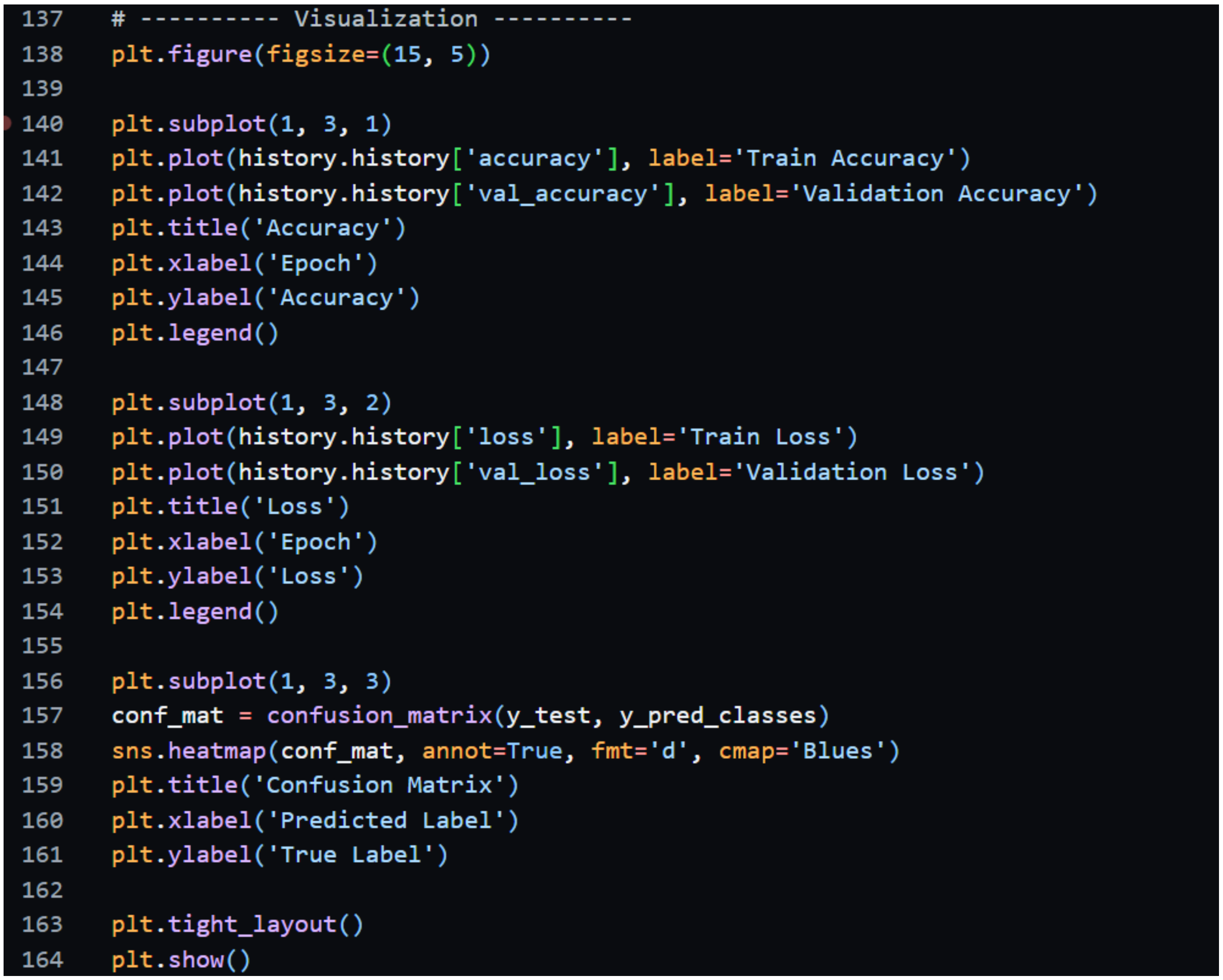


```
# ---------- Evaluation ----------
test_loss, test_acc = model.evaluate(X_test_input, y_test_onehot, verbose=0)
print(f"\nTest Accuracy: {test_acc:.4f}")

y_pred = model.predict(X_test_input)
y_pred_classes = np.argmax(y_pred, axis=1)
print("\nClassification Report:")
print(classification_report(y_test, y_pred_classes))
```

Figure A.4: Code for evaluating the model

The code can also include plotting commands to visualize the confusion matrix and to plot the training history of accuracy and loss which was recorded during training. This can be done with matplotlib.

```
import matplotlib.pyplot as plt
```



```
# ---------- Visualization ----------
plt.figure(figsize=(15, 5))

plt.subplot(1, 3, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

plt.subplot(1, 3, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.subplot(1, 3, 3)
conf_mat = confusion_matrix(y_test, y_pred_classes)
sns.heatmap(conf_mat, annot=True, fmt='d', cmap='Blues')
plt.title('Confusion Matrix')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')

plt.tight_layout()
plt.show()
```

Figure A.4.1: Code for visualizing the model

A.5. Troubleshooting

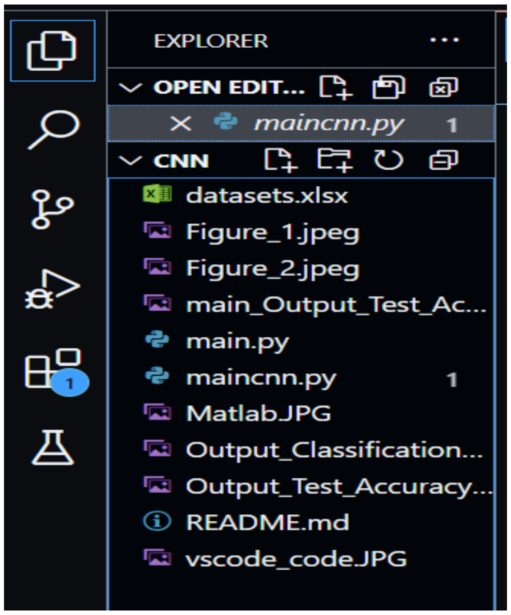
If you encounter memory issues (especially on GPU), you might need to reduce batch size or image size.

If TensorFlow is not utilizing the GPU, check that you installed the GPU version and that your CUDA drivers are set up properly.

Ensure that your Keras image data generator (if used) or custom loader is shuffling training data; otherwise, the training might be suboptimal.

A.6. File Organization

Typical project repository structure:



```
EXPLORER
OPEN EDIT...
  maincnn.py   1
CNN
  datasets.xlsx
  Figure_1.jpeg
  Figure_2.jpeg
  main_Output_Test_Ac...
  main.py
  maincnn.py   1
  Matlab.JPG
  Output_Classification...
  Output_Test_Accuracy...
  README.md
  vscode_code.JPG
```

Figure A.6: Project repository structure

By following these instructions, one should be able to set up the environment, train the vision-to-action model, and replicate the results reported in this project. Future users can build upon this codebase to explore new tasks or improve the model as suggested in the future work section.