Chapter 14

Deep Learning

14.1 Overview

The robot models discussed in the previous chapter are not "intelligent" machines at all; they are, in a sense, merely "automation" machines used to solve some pre-specified tasks, for example, following a planned route repeatedly. Everyone knows that human brains' cognitive processes are completely different. In the real world, classic robot models cannot handle ad-hoc tasks in complex or unstructured environments - tasks that the human brain manages as conditioned reflexes, such as collision avoidance, tracking, and grasping objects of different shapes. It is not realistic or sufficient to predict the next possible pose or trajectory using only the robot's kinematics, and dynamic model without enough perception and learning.

On the other hand, humans have been endeavoring to explore human brains' cognitive processes, while simultaneously applying new discoveries - or algorithms inspired by human intelligence, such as neural network - to improve machine intelligence. This pursuit is known as Artificial intelligence(`AI`), a research field since the 1950s that develops and studies methods and software enabling machines to perceive their environment and take actions that maximize their chance of achieving a defined goal. Machine learning is a branch of AI focused on programs that learn patterns and rules from data. Although its origins can be traced back to the early days of modern computer science, only in the last decade, - with the advent of low-cost and high-performance `CPUs` and graphics processing units (`GPU`) along with advanced intelligent sensors - has a particular subfield of machine learning achieved remarkable success across virtually all domains: deep learning.

It is impossible to explain machine learning(`ML`) in a single chapter to cover its core concepts, practical applications, and surrounding ethical challenges. In this chapter, we will explore some mature machine learning techniques that can be applied to our dual-arm platform. In particular, we will learn how to find, create, and train deep learning models with TensorFlow.

14.2 About TeensorFlow

`TensorFlow` , developed by Google, is an open-source machine learning Library. `TensorFlow` can be effectively used for various aspects of robot gripper control and manipulation, particularly for tasks in complex or unstructured environments where traditional programming methods are insufficient, such as object recognition, grasping, and interaction.

`TensorFlow Lite` can be used to optimize and deploy trained models on embedded systems or robotic platforms with limited computational resources, such as Raspberry Pi or microcontrollers.

14.2.1 TensorFlow Installation

14.2.1.1 Install `TensorFlow Lite` on Raspberry PI 4

If you work with a single robot arm and need to build a ROS app on `TensorFlow` the lightweight version: `TensorFlow Lite` with C++ API only is a better choice. The procedure is very simple. Just clone the GitHub repository and run the two scripts. The commands are listed below. For those who have previously installed TensorFlow Lite on a Raspberry PI, note the subtle difference. Here we use `build_aarch64_lib` because Ubuntu is a 64-bit operating system compared to the 32-bit Raspbian versions, `build_rpi_lib` .

```
$ cd ~
# the dependencies
$ sudo apt-get install wget curl
# download TensorFlow 2.5.0
$ wget -O tensorflow.zip https://github.com/tensorflow/tensorflow
/archive/v2.5.0.zip
# unpack and give the folder a convenient name
$ unzip tensorflow.zip
$ mv tensorflow-2.5.0 tensorflow
$ cd tensorflow
# get the dependencies
$ ./tensorflow/lite/tools/make/download_dependencies.sh
# build TensorFlow Lite
$ ./tensorflow/lite/tools/make/build_aarch64_lib.sh
```

The last step is building the `TensorFlow` Lite flat buffers. Please use the following commands:

```
$ cd ~/tensorflow/tensorflow/lite/tools/make/downloads/flatbuffers
$ mkdir build
$ cd build
$ cmake ..
$ make -j4
$ sudo make install
$ sudo ldconfig
```

If everything went well, you should have the two libraries and two folders with header files under folder

`~/tensorflow/tensorflow/lite/tools/make/gen/linux_aarch64/lib`

`benchmark-lib.a` and `libtensorflow-lite.a`

14.2.1.2 Install `TensorFlow` on Master Machine

Installing `TensorFlow` on Ubuntu 18.04 can be done for either CPU-only or GPU-accelerated versions.

It's important to know that as `TensorFlow` depends on a serial open piece of software, to get around the dependency clashing issue, it is recommended to install `TensorFlow` within `Virtualenv` or `Conda's` environments. They solve the conflicting dependency issue with relatively low overhead. Here, we choose `virtualenv` environment for easy installation. The general process involves setting up a Python environment and then installing `TensorFlow` using `pip` . For CPU-only `TensorFlow` , install `python3-venv` .

```
sudo apt update
sudo apt-get install python3-venv
```

or sudo apt-get install python3.7-venv if you use python 3.7 - Create and activate a Python virtual environment:

```
mkdir ~/tensorflow_env
python3 -m venv ~/tensorflow_env/tf_venv
source ~/tensorflow_env/tf_venv/bin/activate
```

Your shell prompt should look something like this now with (`tf_venv`) preceding your typical shell prompt.

```
(tf_env) peitao@ubuntu:$
```

- Upgrade `pip` and `setuptools` within the virtual environment

```
pip install --upgrade pip setuptools
```

install `Tensorflow`

```
pip install tensorflow
```

For GPU-Accelerated `TensorFlow` (requires NVIDIA GPU), more process is more involved and requires specific NVIDIA drivers, `CUDA` Toolkit, and `cuDNN` libraries to be installed before installing TensorFlow. Install `NVIDIA` Drivers: Add the graphics-drivers PPA and install a compatible NVIDIA driver for your GPU.

```
sudo add-apt-repository ppa:graphics-drivers/ppa
sudo apt update
sudo apt install nvidia-driver-< version> # Replace < version>
                # with a compatible driver
sudo reboot
```

Install `CUDA` Toolkit: Download and install the appropriate `CUDA` Toolkit version compatible with your `NVIDIA` driver and the TensorFlow version you plan to use. Refer to the official `NVIDIA` `CUDA` Toolkit documentation for specific instructions for Ubuntu 18.04. Install `cuDNN` : Download and install the `cuDNN` libraries, which are essential for GPU-accelerated deep learning with TensorFlow. Create and activate a Python virtual environment (same as for CPU-only). Upgrade `pip` and `setuptools` : (same as for CPU-only). Install TensorFlow-GPU:

```
pip install tensorflow-gpu
```

Install Python dependencies `Pandas` , `Matplotlib` used in this chapter:

```
pip install pandas
pip install matplotlib
```

Verification: After installation, you can verify by opening a Python interpreter within your activated virtual environment and running:

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf

print("Pandas version:", pd.__version__)
print("matplotlib version:", plt.__version__);
print("NumPy version:", np.__version__)
print("TensorFlow Version", tf.__version__)
print("GPU installation:", tf.config.list_physical_devices('GPU'))
print(tf.keras.__version__) # for Keras version
```

You may encounter module mismatch errors if the default TensorFlow module cannot be loaded successfully due to older CPU instructions not supporting SSE4.2 etc., you may manually reinstall older versions by:

```
pip uninstall tensorflow
pip install tensflow=2.4.0
```

Or set up `TensorFlow` in a different virtual environment such as Python 3.6 such as for older CPU: Intel Core2 Duo CPU `P8600` , `TensorFlow` has to be 1.5 and Python 3.5-3.7

```
(tf15-py36) ptshi@peitao-Latitude-E5500:~$ python test_ver.py
Pandas version: 1.1.5
matplotlib version: 3.3.4
Numpy version: 1.16.4
TensorFlow vesion: 1.5.0
2025-09-01 19:16:02.864468: I tensorflow/core/platform/cpu_feature_guard.cc:137]
     Your CPU supports instructions that this TensorFlow binary was
        not compiled to use: SSE4.1
GPU is not available.
Keras Version: 2.1.2-tf
```

Since `TensorFlow` 2.0+, `Keras` is integrated directly into `TensorFlow` and is accessible via `tf.keras` . If the installation was successful, it will print the versions of `TensorFlow` and `Keras` .

`Keras` essentially acts as a wrapper or a simplified interface (high-level API) for `TensorFlow` . Being built in Python, `Keras` offers a familiar and easy-to-read structure for Python developers. It is ideal for

beginners in deep learning and for quickly implementing and experimenting with models due to its modular design and clear syntax.

14.3 Machine Learning to Deep Learning

There are several types of machine learnings, such as `Supervised learning` , `Unsupervised learning` and `Reinforcement learning` . We will focus on `supervised learning` where we train an inference model with an input dataset, along with the real or expected output for each example. The model will cover a dataset and then be able to predict the output for new inputs that don't exist in the original training dataset.

Historically, many `supervised learning` models were developed from linear models such as `Linear Regression` for continuous models, Classification for discrete and not continuous targets, and later extended to non-linear models such as `Decision Trees` , `Support Vector Machines` , and `Naive Bayes` et al. However, these models have their respective limitations and could not tackle complex or large datasets prior to the invention of the multi-layered Neural Networks model.

The multi-layered neural network was inspired by the structure and function of the human brain. Deep learning's ability to learn complex patterns relies on its depth of layers - the numerous interconnected layers that progressively process and interpret raw data. As a subset of machine learning, deep learning represents an evolution toward more autonomous and sophisticated learning, particularly with large datasets and complex tasks such as image recognition and natural language processing.

There are three typical deep learning models from simple to complex: Feedforward neural network (`FNN`) or multilayer perceptron (`MLP`), convolutional neural networks (`CNN`), and recurrent neural networks (`RNN`).

Feedforward neural network(`FNN`) (Figure 14.1) refers to the recognition-inference architecture of neural networks. It is based on inputs multiplied by weights to obtain outputs (from inputs-to-output) without any feedback from later processing stages to earlier stages, which is distinguished from recurrent neural network.

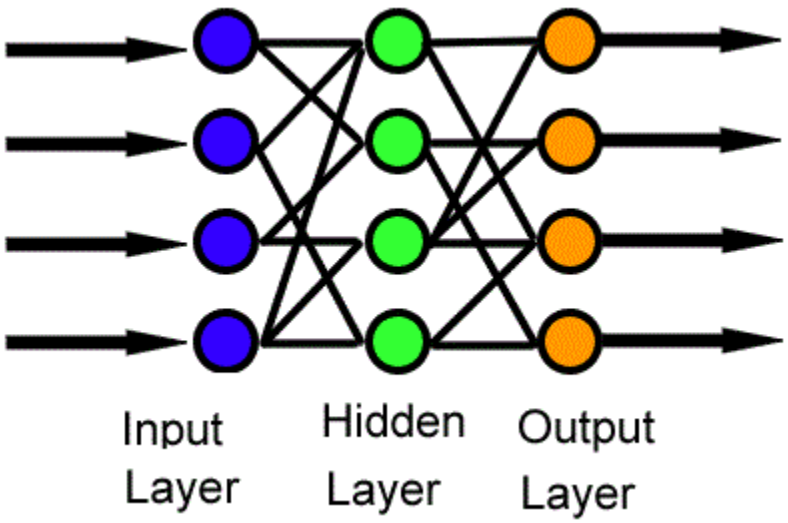

Figure 14.1: Feedforward neural network

In a feedforward network, information always moves in one direction: it never goes backward. In a multiple-layer NN, one node shown in Figure 14.2 (also called `perceptrons` , or `neron`) in the intermediate layer or hidden layer, the node's out is a function of the previous layers node. It has another name activation function.

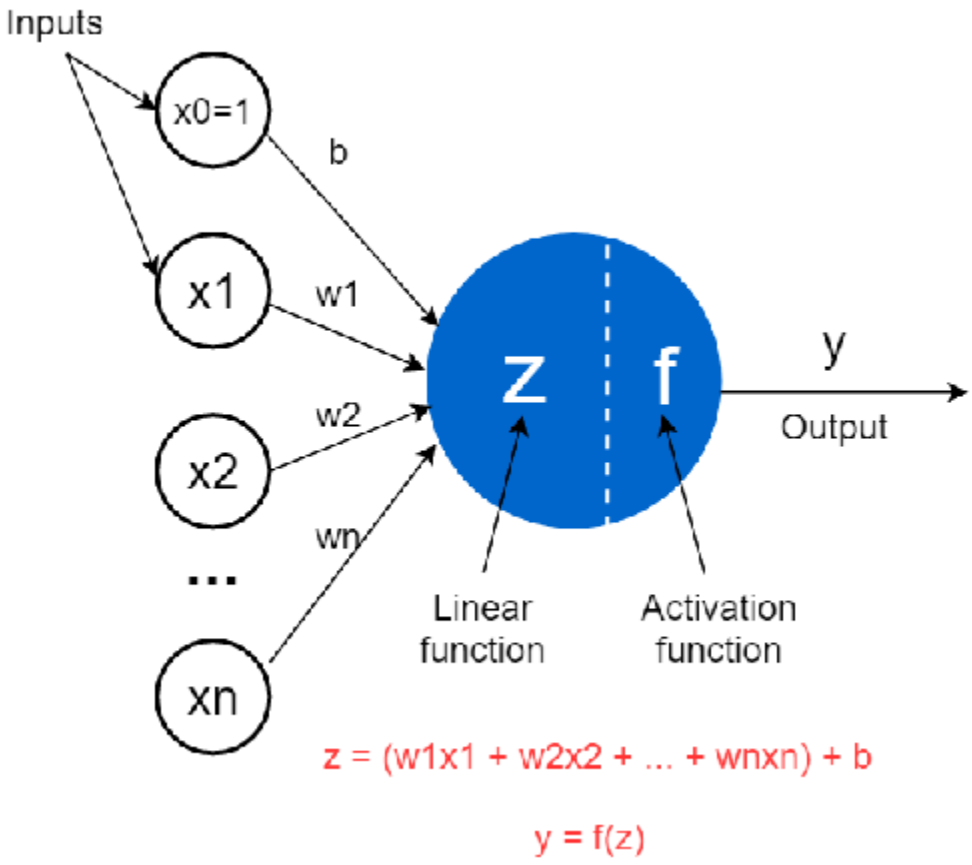

Figure 14.2: The structure of a perceptron

Activation functions introduce non-linearity into neural networks, enabling them to learn complex patterns. There are multiple types of activation functions for different application scenarios (Figure 14.3):

- binary step: or step function:

$$\text{binary step}(z) = \begin{cases} 1, & \text{if } z > 0 \\ 0, & \text{if } z \leq 0 \end{cases}$$

- sigmoid:

$$sigmoid(z) = \frac{1}{1 + e^{-z}}$$

- ReLU: The rectified linear unit function is defined as:

$$ReLU(z) = \begin{cases} z, & \text{if } z > 0 \\ 0, & z \leq 0 \end{cases}$$

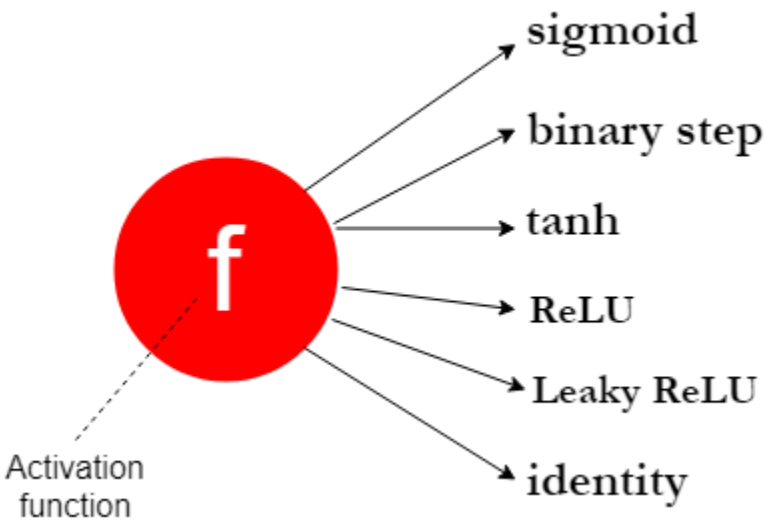

Figure 14.3: Activation function

- tanh:

$$tanh(z) = \frac{sinh(z)}{cosh(z)} = \frac{e^z - e^{-z}}{e^z + e^{-z}} = \frac{e^{2z} - 1}{e^{2z} + 1}$$

If we plot all the above activation functions in one figure (Figure 14.4), we can see that every activation function has different outputs and characteristics: for example, the output of the binary step is valid when $z > 0$ which is suitable for discrete systems, but rarely used due to being non-differentiable at $z = 0$. `Sigmoid` is continuous from between the range [0,1] and primarily for output layers in binary classification tasks. `Tanh` has zero centered output and can be used in hidden layers where zero-centered outputs are required. `ReLU` is the best and most popular activation function in deep learning and is the default choice for hidden layers due to its computational efficiency and ability to mitigate vanishing gradients. Other non-linear activation functions or combinations theeof - are sometimes used in multi-layer perception (`MLP`), a misnomer for a modern feedforward artificial neural network. An MLP consists of fully connected neurons (hence the synonym fully connected network, or `FCN`).

Convolutional neural networks (`CNNs`) are another type of feedforward network, and are used in computer vision and automatic speech recognition (`ASR`). As its name hints, `CNN` processes grid-like data, such as images, by using layers of filters to automatically detect features. These filters slide across the input, computing dot products to create “feature maps” highlighting patterns like edges or shapes. Pooling layers then reduce the size of these maps to make the network more efficient. Finally, fully connected layers take these high-level features to classify the image, such as identifying it as a “dog’ or”cat”.

A recurrent neural network(`RNN`) is a deep learning model that is trained to process and convert sequential data input into a specific sequential data output, in which data can flow in any direction, is used for applications such as language modeling. Long short-term memory is particularly effective for this use. An `RNN` is a software system that consists of many interconnected components mimicking how humans perform sequential data conversion, such as translating text from one language to another. RNNs are largely being replaced by transformer-based artificial intelligence(AI) and large language models (`LLM`) which are much more efficient in sequential data processing.

As an introduction to deep learning, we only talk about FNN since it is simple and more suited for robot

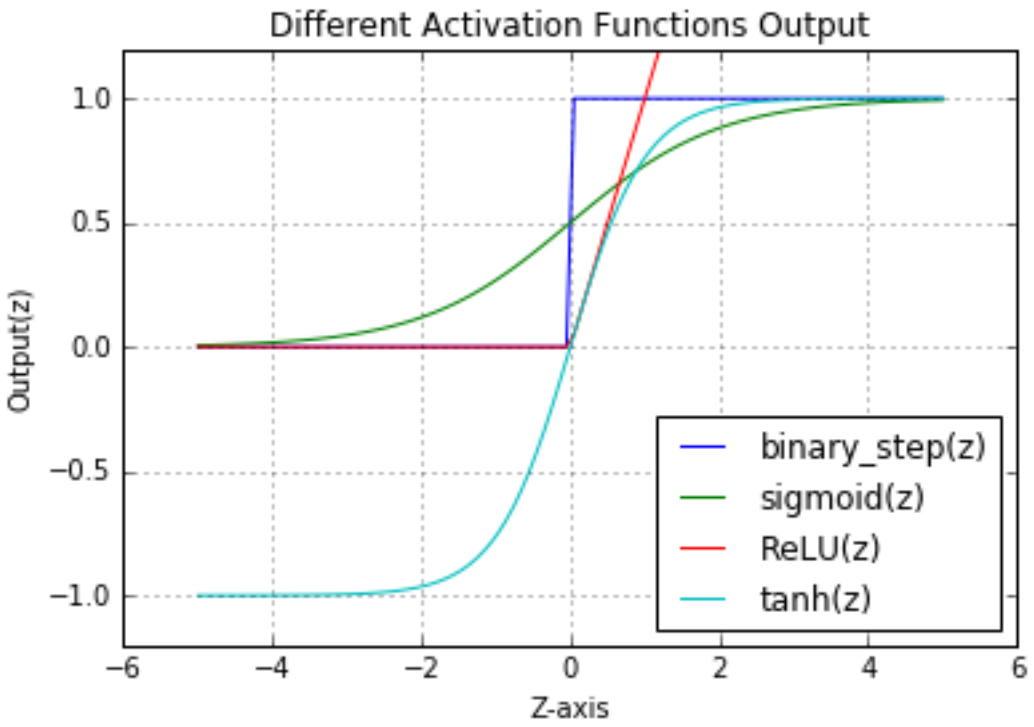

Figure 14.4: Multiple activation functions comparisons

manipulation control or position estimation.

14.4 Deep Learning Process

To implement an `FNN` for a special task, an inference model must first be defined. An inference model is a series of sample operations applied to the input data. In our particular case, these operations involve robot arm's effect positions measured by the depth camera. When we train the inference mode using an input dataset together with the corresponding real or expected output for each example, the model learns the underlying patterns in the dataset. It can then predict outputs for new inputs that were not present in the original training set.

In this section, we learn how a deep learning model is designed and implemented with two fundamental application scenarios: end-effector position estimation and object tracking.

While this can be solved directly with a known kinematic model, a deep learning approach is useful for complex, flexible, or otherwise non-ideal robots where a precise model is unavailable. The process involves data generation, dataset splitting, and model training.

Estimating a robot's end-effector position from its joint variables is a classic forward kinematics problem. A deep learning model for this task is essentially a regression model that learns a mapping from a vector of joint angles to a vector representing the end-effector's pose (position and orientation). After training, we expect that given any joint variables, we can get the end-effector's pose without computing forward kinematics.

Object tracking is the reversed process: giving an object a 3D pose (position and orientation), e.g., detected

and measured by a 3D sensor (e.g., `Realsense` camera), how robot arm joints (6 joint control variables to the robot controller) are controlled to move the end-effector to that position without analytical inverse kinematics.

First, let's take end-effector pose estimation as an example:

14.4.1 Determine Input and Output:

- Input Data:

The primary input consists of the six joint angles (or positions) of a robot arm. These are typically represented as a vector of six floating-point numbers, one for each joint. Example:

$$[q_1, q_2, q_3, q_4, q_5, q_6]$$

Depending on the specific application and desired accuracy, other parameters might be included:

Joint Velocities/Accelerations: If dynamic effects are relevant, these can be incorporated.

Robot Parameters: Link lengths, joint offsets, and other fixed kinematic parameters can be included as part of the model's structure or as additional input features if the model aims to generalize across slightly different robot configurations.

- Output Data: End-Effector Position: This is typically represented as a 3D Cartesian coordinate vector: $[x, y, z]$. End-Effector Orientation: This can be represented in various ways: Euler Angles: $[roll, pitch, yaw]$ or Quaternions: $[qx, qy, qz, qw]$ (preferred for avoiding gimbal lock), or combined 6D pose vector

$$[x, y, z, \phi, \theta, \psi]$$

14.4.2 Dataset Generation

Training requires a large dataset of corresponding joint variable inputs and end-effector pose outputs. The data can be generated through:

- Calculate ground truth end-effector pose by its forward kinematics equations: If the analytical forward kinematics are known, a synthetic dataset can be generated. For a standard 6-joint robot arm like Hex7bot, you can use `Denavit-Hartenberg ( DH )` parameters, calculate the end-effector pose for a given set of joint variables with its the forward kinematics. For each randomly generated set of 6 joint angles, use the forward kinematics (or a simulator) to calculate the corresponding ground truth end-effector pose.
  - Input: 6 joint angles $(\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6)$.
  - Output: The 3D position (x, y, z) and the 3D orientation (e.g., roll, pitch, yaw, or quaternions) of the end-effector. In total, this is a vector of 6 variables.

$$([(\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6), (x, y, z, \text{roll}, \text{pitch}, \text{yaw})])$$

- Combine data: Pair the input joint angles with the output end-effector pose to create your dataset. This dataset will consist of rows, where each row is a training example:

```
## joint_1,joint_2,joint_3,joint_4,joint_5,joint_6,end_effector_x,
       end_effector_y,end_effector_z,end_effector_roll,
           end_effector_pitch,end_effector_yaw
## 0.1,0.2,0.3,0.4,0.5,0.6,1.0,2.0,3.0,0.1,0.2,0.3
```

- Robot Simulation: Simulators like CoppeliaSim or Gazebo can be used to generate data by moving the robot and recording joint states and end-effector poses.
- Real Robot Data: Collecting data from a physical robot, potentially using external tracking systems for ground truth end-effector pose.
- Generate random joint configurations: A simple and effective method is to generate random joint angle values within their specified ranges. The number of samples needed depends on the complexity of the robot's kinematics, but a very large dataset (e.g., hundreds of thousands of samples) is often required for good generalization.

Data Collection

Creating a dataset for robot pose estimation in deep learning involves several key steps to ensure high-quality and diverse data for training robust models.

Real-world Data:

Capture `RGB-D` images (color and depth) of the target objects in various poses, lighting conditions, and backgrounds using 3D cameras such as the Intel RealSense. Accurately determine the ground truth 6D pose (position and orientation) for each object in each image - this often requires specialized tools or manual annotation. Ground truth refers to the factual, verified data used as a benchmark to train and evaluate models. It represents the correct or actual outcome for the system being modeled, providing the "right answers" that the AI learns from and against which it compares it own predictions, thereby ensuring accuracy and reliability.

To facilitate and simplify data collection, we first only collect the left and right arm end-effector position information, not considering its orientation for the moment. Meanwhile, install one point light source (e.g., a tiny red or green LED) on the TCP of the robot arm, and leave the background dark (or test it at night).

Next, we will create a data collection ROS node that performs the following steps:

- Subscribe to all joint states.
- Subscribe to RGB-D 3D point cloud topics and RGB images.
- Randomly jog the left or right arm to any reachable position, especially the overlapped workspace between the left and right arms. For every TCP position, do the following procedure:
  - Record all joint displacements.
  - Calculate `TCP` (end-effector) pose (position and orientation) based on robot arm model (forward kinematics).
  - Perception processing on point clouds (last chapter) by downsampling, outlier removal, segmentation.
  - Image Preprocessing: Identify the 2D pixel (x, y) of source light on the RGB image. Apply techniques like noise reduction, color correction, and normalization to improve image quality and consistency. Average multiple same pixel area to get maximum brightness 2D pixel (x, y).
  - Convert to 3D coordinates: Map the 2D pixel (x, y) and its depth to a 3D point (the distance to that specific pixel) Note the output will be a 3D point (x, y, z) in meters in the camera's coordinate system and needs to be converted to coordinate in robot base frame or world frame.
- Calculate position error between forward kinematics and values derived from `Realsense` depth camera
- Organize the collected and processed data into a structured format suitable for deep learning frameworks. This typically involves storing all joint states (joint1 to joint 6), corresponding position annotations derived from joint state and camera, as well as position errors between as `CSV` format.

Here is a step-by-step process for generating a ground-truth dataset.

- **Step 1:** Define the robot's kinematic model. Before generating any data, you need the analytical forward kinematics model of your 6-joint robot.You can use software like ROS (MoveIt! and Gazebo), `RoboDK` , or kinematics libraries such as `Orocos KDL` . These tools can use a robot's URDF (Unified Robot Description Format) file, which contains the kinematic parameters needed for analytical calculations. If you have developed your robot model library, like we did in the previous chapter. You can develop a simple node for it.
- **Step 2:** Randomly sample joint configurations. To create a robust and comprehensive dataset, you need to generate a wide variety of joint configurations that cover the robot's entire workspace. **Method**: Randomly generate a set of 6 joint angles $(q_1, q_2, q_3, q_4, q_5, q_6)$. Constraints: Ensure that each joint angle is within its physical limits. For example, a joint with a full rotation might range from $-\pi$ to π radians, while one with a limited range might be restricted to $[-\pi/2, \pi/2]$ radians.
- **Step 3:** Compute the ground-truth end-effector pose. For each randomly generated set of 6 joint variables $(q_1, ..., q_6)$, use the analytical forward kinematics model to compute the corresponding end-effector pose. Process: The forward kinematics algorithm takes the joint angles as input and calculates a $4x4$ homogeneous transformation matrix, $T_{base}^{end-effector}$. This matrix describes the end-effector's position and orientation relative to the robot's base frame. Pose components: Position: The position (x, y, z) is found in the fourth column of the transformation matrix. Orientation: The orientation can be extracted from the $3x3$ rotation matrix, which is a sub-matrix of the homogeneous transformation matrix. You can represent this orientation in various formats, such as Euler angles, axis-angle, or quaternions, depending on your model's design.
- **Step 4:** Assemble the dataset. For each random sample, pair the joint variables with the computed ground-truth end-effector pose to create a single training example. Example format:Input: 6 joint angles ([q_1,q_2,q_3,q_4,q_5,q_6]).Output (Ground-truth): End-effector pose, represented as a vector combining position and orientation, such as $[x, y, z, \text{Roll}, \text{Pitch}, \text{Yaw}]$ or a quaternion $[x, y, z, q_x, q_y, q_z, q_w]$.
- **Step 5:** Incorporate randomization (optional, but recommended) to improve the robustness of your deep learning model, you can incorporate slight variations into your dataset. **Sensor noise:** Add a small, normally distributed random value to each joint angle. This simulates the imperfections of real-world joint sensors and can prevent overfitting. **Multiple poses:** For a given set of joint angles, if you need to represent multiple potential end-effector poses (due to redundancy in the inverse kinematics, for example), you can generate data that captures these variations. Why analytical kinematics is ideal for ground truth. Using analytical kinematics to generate training data is superior to using physical sensor readings for several reasons: Perfectly labeled data: Analytical forward kinematics provides a computationally perfect and noise-free ground truth. Sensor readings from a real robot would contain noise and require expensive, highly accurate metrology tools to be used as ground truth for training. **Massive dataset generation:** Analytical methods allow for the creation of vast, randomized datasets very quickly, which is essential for training deep neural networks. **Comprehensive coverage:** You can easily sample the entire joint space and define specific areas of interest, such as singular configurations or the edges of the workspace, to ensure your model learns all conditions.

Dataset split

To train and evaluate your model effectively, split the generated data into training, validation, and testing sets.

- Training set: Use this portion of the data (e.g., 80%) to train the model. The model learns the mapping from joint angles to end-effector pose by minimizing the error on this data.

- Validation set: Use this portion (e.g., 10%~15%) to tune the model's hyperparameters and prevent overfitting during training. The validation error helps decide when to stop the training early to avoid memorizing the training data.

- Testing set: Use this final, unseen portion (e.g., 10%) to evaluate the model's performance after training. The test set provides an unbiased measure of the model's generalization ability to new joint configurations it has never seen before.

Data Diversity

The dataset should cover a wide range of joint configurations and corresponding end-effector poses to ensure the model learns a robust mapping across the robot's workspace. Real-world Data: While simulated data can be used for initial training, incorporating real-world data is essential to account for sensor noise, mechanical inaccuracies, and other real-world complexities.

Randomization: During data collection, it is beneficial to randomize the joint configurations to avoid biases in the dataset.

If the end-effector position estimation is part of a trajectory following or control task, ensuring the temporal coherence of the data within the splits is important. This might involve splitting trajectories rather than individual data points.

Stratification: If certain regions of the workspace or specific joint configurations are more critical, consider stratifying the data split to ensure adequate representation of these areas in all sets.

- Randomization: Shuffle the dataset before splitting to ensure the three sets are a random and representative sample of the full working space.
- Distribution: Ensure the data split maintains a similar distribution of poses across all three sets.

Data Validation

Data validation before the training process. Manually inspect a subset of the data to ensure the accuracy of annotations and the quality of generated images.

By following these steps on the left-arm and right-arm, a comprehensive and high-quality dataset can be created for training deep learning models for accurate robot pose estimation.

Code snippet of data collection ros node:

- Data collector node header file. It contains two classes, `DataCollector` and composite class `CsvWriter` for writing CSV files.

```
#ifndef __DATA_COLLECT_H__
#define __DATA_COLLECT_H__

#include <fstream>
#include <string>
#include <vector>
#include <sstream>
#include <iostream>
#include <chrono>
#include <iomanip> // For Std::put_time

// ROS
#include <ros/ros.h>
#include <std_msgs/String.h>
#include <sensor_msgs/JointState.h>
#include <geometry_msgs/PointStamped.h>

// robotcommon
#include "rbt_models/rbt_common.h"
#include "rbt_models/base_model.h"
#include "rbt_models/hex7bot_model.h"

class CsvWriter {
public:
  CsvWriter(const string& filename)
    : file_(filename){}

 ~CsvWriter() {
     if (file_.is_open()) {
        file_.close();
     }
  }

  CsvWriter() {}
```

- Open a CSV and write head line.

```
  bool Open(const string& filename);
  bool isOpen();
  void Close();
  //Write a header row from a vector of strings
  void writeHeader(const vector<string>& headers);
  template <typename T>
  void writeRow(const vector<T>& col);
  // Function to write a row from a mix of types
  template<typename T, typename... Args>
  void writeRow(const T&first, const Args&... args);
private:
  ofstream file_;
  //Helper function to write a single element
  template <typename T>
  void writeElement(const T& element);
  // Helper function for the last element in a variadic pack
  template <typename T>
  void writeRest(const T& element) ;

  //Recursive helper function to write multiple elements
  template <typename T, typename... Args>
  void writeRest(const T& first, const Args&... args);
};
```

- class DataCollector Header definition (part 1)

```
class DataCollector
{
public:
  DataCollector(ros::NodeHandle& nh);
  ~DataCollector();

  enum {
     STATUS_OK = 0,
     ARM_TYPE_NOT_DEFINED = 1,
     ARM_TYPE_NOT_SUPPORTED = 2,
     ROBOT_NAME_NOT_FOUND = 3,
     ROBOT_MODEL_IS_NULL =4,
     INVALID_ROBOT_DESCRIPTION_NAMESPACE = 5,
     URDF_INVALID_ROOT_LINK = 6,
     CSV_FILE_IS_NOT_OPEN = 7};

  bool  CsvFileWriteStart();
  int   getStatus() { return status_; };

private:
  /* \brief Callback for joint subscription */
  void jointStatesCB(const sensor_msgs::JointState::ConstPtr&  msg);
  /* \brief callback for camera position subscription*/
  void camera3DLocatorCB(const geometry_msgs::PointStamped::ConstPtr& msg);
  /* \brieff create Robot Model object based on robot_name */
  BaseModel* createRobotModel(const std::string& robot_name);
  /* \brief read robot name from its robot_description urdf file */
  string getRobotName (const string& robot_description);
  /* \brief get base to world frame parameters*/
  void getBaseLinkParameters(const string robot_description,
       vector<double>& xyz, vector<double>& rpy);
```

- class DataCollector private members (part 1)

```
  /* \brief Write one line record */
  void csvWriteRow();
  ros::NodeHandle nh_;
  CsvWriter csv_file_;
  // (mutex protected below)
  mutable std::mutex data_mutex_;
  // ROS
  ros::Subscriber joint_state_sub_;
  ros::Subscriber camera_pos_sub_;
  std::string arm_type_;
  //Robot module pointer
  BaseModel* robot_model_;
  //CSV data record:
  vector<double> jointPositions_;
  //geometry_msgs::Pose endEffector_Pose_;
  vector<double> endEffector_Pose_; //x,y,z, roll, pitch, yaw
  //geometry_msgs::Pose cameraSensePosision_;
  // Only position.x, msg->position.y, msg->position.z valid
  vector<double> depth3DPosition_;

  bool     jointStateUpdated_;
  bool     cameraPositionUpdated_;
  int      row_counter_;
  int      status_;
  double   allow_rmse_error_;
  double   allow_x_range_;
};
#endif // end of __DATA_COLLECT_H__
```

- DataCollector: Get current Robot name, create its robot model and load world to base frame information to that model.

```
string DataCollector::getRobotName (const string& robot_description){
  string robot_name = "";
  urdf::Model model;

  if (model.initParam(robot_description)) {
    // Access the robot name from the root element
    robot_name = model.getName() ;
  }
  return robot_name;
}

BaseModel* DataCollector::createRobotModel(const std::string& robot_name) {
  if (robot_name == "hex7bot" || robot_name =="left_arm" ||
        robot_name == "right_arm") {
    ROS_DEBUG("create Hex7BotModel from %s\n", robot_name.c_str());
    return new Hex7BotModel(MAX_7BOT_JNTS, robot_name);
  }
  else {
    ROS_ERROR("Not supported robot name: %s", robot_name.c_str());
    return nullptr;
  }
}

void DataCollector::getBaseLinkParameters(const string robot_description,
                vector<double>& xyz, vector<double>& rpy) {
 urdf::Model model;
 if (model.initParam(robot_description)) {
   // Get a specific joint by its name
   urdf::JointConstSharedPtr joint = model.getJoint("joint_base");
   if (joint) {
       // Access the origin's xyz and rpy
       xyz[0] = joint->parent_to_joint_origin_transform.position.x;
       xyz[1] = joint->parent_to_joint_origin_transform.position.y;
       xyz[2] = joint->parent_to_joint_origin_transform.position.z;
       double roll, pitch, yaw;
       joint->parent_to_joint_origin_transform.rotation.getRPY(roll,pitch, yaw);
       rpy[0] = roll;
       rpy[1] = pitch;
       rpy[2] = yaw;
   }
   else{
       ROS_ERROR("joint_base not found from robot_description %s",
                   robot_description.c_str());
   }
 } // end of if(root_link)
 else {
     ROS_ERROR("Could not find robot_description %s ", robot_description.c_str());
 }
}
```

- DataCollector Object constructor function: Reading its parameters from launch file, creating Robot model for forward kinematics. Subscribing robot JointStates and camera sensed positions topics.

```
DataCollector::DataCollector(ros::NodeHandle& nh)
  : nh_(nh) , robot_model_(nullptr)
  , arm_type_(""), jointPositions_(vector<double>(MAX_7BOT_JNTS,0))
  , endEffector_Pose_(vector<double>(6,0)) //x,y,z, roll, pitch, yaw
  , depth3DPosition_(vector<double>(3,0)) // Only position x,y,z valid
  , jointStateUpdated_(false)
  , cameraPositionUpdated_(false)
  , row_counter_(0) , status_(STATUS_OK){

  string arm_robot_description ;
  string node_namespace = ros::this_node::getNamespace();
  if (nh.getParam("arm_type", arm_type_)){
      //Get this arm's robot name from its robot_description file
      string robotName =  getRobotName(arm_robot_description);
      //Create robot model based on robot name
      robot_model_=  createRobotModel (robotName);
  }
  //Get allowed distance error
  allow_rmse_error_ = 0.025; //2.5mm
  if (nh.getParam("allow_rmse", allow_rmse_error_)){
      ROS_INFO("maximum allowed RMSE error: %g m\n", allow_rmse_error_);
  }
  //Get allow x distance range
  allow_x_range_ = 1.0; //100cm
  if (nh.getParam("x_range", allow_x_range_)){
      ROS_INFO("maximum allowed X range : %g m\n", allow_x_range_);
  }
  //Get this arm's baselink information(robot base to world frame)
  if (robot_model_){
    vector<double> base_xyz = { 0.0, 0.0, 0.0};
    vector<double> base_rpy = { 0.0, 0.0, 0.0};
    string cam_robot_description = node_namespace + "/camera/robot_description";
    getBaseLinkParameters((const string) arm_robot_description, base_xyz, base_rpy);
    //Set baselink frame information to Robot model:
    robot_model_->setBase_wrt_world(base_xyz, base_rpy);
    //Create a subscriber
    std::string joint_state= "/" + arm_type_ + "/joint_states";
    joint_state_sub_ = nh_.subscribe<sensor_msgs::JointState>
            (joint_state, 10, &DataCollector::jointStatesCB, this);
    //subscriber camera detected position
    camera_pos_sub_ = nh_.subscribe<geometry_msgs::PointStamped>
        ("/camera/deprojection_position", 10,
            &DataCollector::camera3DLocatorCB, this);
  }
}
```

- DataCollector JointState callback function. This function calls the robot model's forward kinematics to get the end-effector position with respect to the world frame. Meanwhile, this function calculates end-effector Euler angles (roll, pitch, yaw). Please note that we do not use ROS provided `tf2` to get end-effector position because D-H parameters in the robot model is customized for robot control, such as frame arrangement for controlled joint angle and joint offset, etc., which is different from URDF definitions. If the camera-sensed position data is ready, it will save both positions to the CSV file.

```
void DataCollector::jointStatesCB(const sensor_msgs::JointState::ConstPtr&  msg){
 const std::lock_guard<std::mutex> state_lock(data_mutex_);
  for (size_t msg_j = 0; msg_j < MAX_7BOT_JNTS; msg_j++) {
      jointPositions_[msg_j] = msg->position[msg_j];
  }

  /* modified DH Forward Kinematics */
  vector<vector<double> > Tpose = robot_model_->
                                    GetEndEffectorPose_wrt_world(jointPositions_);

  endEffector_Pose_[0] = Tpose[0][3];
  endEffector_Pose_[1] = Tpose[1][3];
  endEffector_Pose_[2] = Tpose[2][3];
  // Get new orienetation
  double roll_x=0, pitch_y=0, yaw_z=0;
  pitch_y = asin (-Tpose[2][0]);
  //pitch_y as M_PI/2 or -M_PI/2
  if ((fabs(Tpose[0][0]) > CALC_TOLERANCE)||(fabs(Tpose[1][0]) > CALC_TOLERANCE)){
     pitch_y = atan2(-Tpose[2][0], sqrt(Tpose[0][0] * Tpose[0][0] +
                              Tpose[1][0] * Tpose[1][0]));
     yaw_z  = atan2(Tpose[1][0] / cos(pitch_y), Tpose[0][0] /cos(pitch_y));
     roll_x = atan2(Tpose[2][1] / cos(pitch_y), Tpose[2][2] /cos(pitch_y));
  }
  else {
     yaw_z = 0.0;
     if ( Tpose[2][0] > 0){
          pitch_y = -M_PI/2;
     } else{
          pitch_y = M_PI/2;
     }
     if ( fabs(Tpose[1][1]) > CALC_TOLERANCE)
        roll_x = -atan2(Tpose[0][1], Tpose[1][1]);
     else
        roll_x = (pitch_y > 0)? asin(Tpose[0][1]): -asin(Tpose[0][1]);
  }
  endEffector_Pose_[3] = roll_x;
  endEffector_Pose_[4] = pitch_y;
  endEffector_Pose_[5] = yaw_z;

  //Write to CSV file
  if ( cameraPositionUpdated_) {
      csvWriteRow();
  }
  else {
     jointStateUpdated_ = true;
  }
}
```

- camera3DLocatorCB callback function: camera sensed 3D position has been converted from Camera

link frame to the world frame (by ROS `tf2`). The tf2 is very convenient on converting fixed frame information position.

```
void DataCollector::camera3DLocatorCB
  (const geometry_msgs::PointStamped::ConstPtr& msg){
  if ( msg->point.x > allow_x_range_) {
    ROS_WARN("DataCollector::Camera detected Position(world): x=%.2f >
            maximum range: %.2f",
            msg->point.x, allow_x_range_);
    cameraPositionUpdated_ = false;
    return;
  }
  //convert camera link to world frame
  depth3DPosition_[0] =  msg->point.x;
  depth3DPosition_[1] =  msg->point.y;
  depth3DPosition_[2] =  msg->point.z;

  if ( jointStateUpdated_ ){
      csvWriteRow();
  }
  else {
      cameraPositionUpdated_ = true;
  }
}
```

- DataCollect CSV write interface functions:

```
bool DataCollector::CsvFileWriteStart() {
  //Get the current time
  auto now = chrono::system_clock::now();
  //Convert to local time
  time_t now_c = chrono::system_clock::to_time_t(now);
  //Convert to a local time structure
  tm* local_tm = localtime(&now_c);
  //Format the timestamp as as tring
  ostringstream oss;
  oss << std::put_time(local_tm, "%Y%m%d") << ".csv";
  string csv_file_name = arm_type_ +"_" + oss.str();
  //Open the CSV file as: "robot_name_YYYMMDD.csv" file
  csv_file_.Open(csv_file_name);

  if (!csv_file_.isOpen()) {
     ROS_ERROR("Failed to open CSV file for writing!");
     status_ = CSV_FILE_IS_NOT_OPEN;
     return false; //Exist if file can not e opened
  }
  //Write CSV header(optional)
  vector<string> csv_headers =
     { "joint_1", "joint_2", "joint_3", "joint_4", "joint_5", "joint_6",
       "end_effector_x", "end_effector_y", "end_effector_z", "end_effector_roll",
       "end_effector_pitch", "end_effector_yaw",
       "depth_3D_x","depth_3D_y", "depth_3D_z"};
  csv_file_.writeHeader(csv_headers);
}
```

- DataCollect: Write one line record to CSV file

```
void DataCollector::csvWriteRow(){
  if (csv_file_.isOpen()) { //calculate RMSE
     double rmse =
        std::sqrt( (std::pow(endEffector_Pose_[0] - depth3DPosition_[0], 2)
            + std::pow(endEffector_Pose_[1] - depth3DPosition_[1], 2) +
            +  std::pow(endEffector_Pose_[2] - depth3DPosition_[2], 2)) / 3.0);
    if (rmse <= allow_rmse_error_) {
      const std::lock_guard<std::mutex> state_lock(data_mutex_);
      csv_file_.writeRow(jointPositions_[0], jointPositions_[1], //Joint 1-2
                  jointPositions_[2], jointPositions_[3],//Joint 3-4
                  jointPositions_[4], jointPositions_[5], //Joint 5-6
                  endEffector_Pose_[0], endEffector_Pose_[1],
                  endEffector_Pose_[2], //TCP x, y,z
                  endEffector_Pose_[3], endEffector_Pose_[4],
                  endEffector_Pose_[5], // roll, pitch, yaw
                  depth3DPosition_[0], depth3DPosition_[1],
                  depth3DPosition_[2]);  //TCP x, y, z from camera deprojection
      row_counter_ ++;
      ROS_INFO(" Data record count %d: RMSE: %g, written to csv file.",
                 row_counter_, rmse);
    } else{
       ROS_WARN("Data RMSE_error: %g is larger than %g, skip writting \n",
             std::abs(distance), allow_rmse_error_);
    }

    cameraPositionUpdated_ = false;
    jointStateUpdated_ = false;
  }
}
```

- Data Collector main function

```
int main(int argc, char **argv){
  ros::init(argc, argv, NODE);
  //Private NodeHandle to access node-specific parameter
  ros::NodeHandle nh("~");

  DataCollector  dataLogger(nh);

  dataLogger.CsvFileWriteStart();

  if (dataLogger.getStatus() != DataCollector::STATUS_OK) {
      ROS_ERROR("Data Collector has error: %d , abort! ",
          dataLogger.getStatus());
      return -1;
  }
  // Keep the node alive and process callbacks
  ros::spin(); .

  return 0;
 }
```

- Start Data collector node with a launch file, such as data_collector_left_arm.launch:

```
<?xml version="1.0"?>
<launch>
  <arg name="camera_name" value="camera" />
  <arg name="arm_type" value = "left_arm"/>
  <!-- Include a launch file from another package -->
  <include file="$(find realsense2_camera)/launch/rs_aligned_depth.launch" />
  <group ns="$(arg camera_name)">
     <!-- Send the robot description to the parameter server -->
     <param name="robot_description" command="$(find xacro)/xacro
            --inorder $(find rbt_descriptions)/urdf/dual_arm_d435.urdf.xacro" />
     <!-- Start camera deprojection processing -->
     <node name="depth3Dlocation" pkg="rbt_percept_process"
           type="rs2Point3DLocator"  output="screen"/>
     <!-- publish a static transform from camera_optical_frame to world_frame -->
     <node name="world_to_camera_optical_tf" pkg="tf"
           type="static_transform_publisher"
           args="0 0.018 0.435 0 0 0  world camera_link 10" output="screen">
     </node>  </group>
  <group ns="$(arg arm_type)">
      <!-- keyboard  publishes twist or jointTrajectory command -->
      <node name="keyboard_jog_publisher" pkg="teleop"
            type="keyboard_jog" output="screen">
        <param name="ros_queue_size" type="int" value="10" />
        <param name="step_gain" type="double" value="1.25" />
        <param name="joint_vel" type="double" value="0.5" />
        <param name="linear_pos" type="double" value="0.2" />
      </node>
     <!-- Start the robot state publisher to publish the /tf frames -->
     <node name="robot_state_publisher"  pkg="robot_state_publisher"
           type="robot_state_publisher" >
        <remap from="/robot_description" to="/$(arg arm_type)/robot_description"/>
        <remap from="/joint_states" to="/$(arg arm_type)/joint_states" />
     </node></group>
  <!-- Write data to CSV file -->
  <node name="Data_collector" pkg="rbt_learning" type="data_collect_node"
          output="screen">
      <param name="arm_type" type="string" value="$(arg arm_type)"/>
      <param name="allowed_rmse" type="double" value="0.05" />
      <param name="x_range" type = "double" value = "1.0" />
  </node>
</launch>
```

Attach one LED (red) source at TCP (Tool center point), for the Hex7bot big claw gripper, we let gripper two jaws close to hold it. Please note that you have to measure the real distance from the LED the to wrist origin (intersection of frame 4, 5, 6) as d_6 of D_H parameter to compute forward kinematics.

Manipulate the robot arm in the launched window with the keyboard.(Please refer to teleop-Arm chapter for robot arm control), the data collector node will auto-collect position for that LED, save the log to the

CSV file if root mean square error (RMSE) between camera-sensed position and the derived position from joint states is less than the predefined threshold.

$$RMSE = \sqrt{\frac{(x_d - x_c)^2 + (y_d - y_c)^2 + (z_d - z_c)^2)}{3}} \tag{14.4.1}$$

Console window outputs some more debugging information, such as:

```
[ INFO] [1763348529.899507838]:  Depth_raw on image: (48,451)=326

[ INFO] [1763348529.901618640]: Point in camera_frame (-0.15, 0.11, 0.33) transformed
                    to world_frame (0.32, 0.18, 0.32)
[ INFO] [1763348529.902123171]: DataCollector::Camera detected Position(world):
                    x=0.32, y=0.18, z=0.32
[ INFO] [1763348529.902289742]: 3D position of pixel (48, 451) is:
                    (-0.1458, 0.1122, 0.3260) meters
[ INFO] [1763348529.942734144]: Best red pixel found at: (38, 453)
[ INFO] [1763348529.948009440]: DataCollector::jointStateCB: joint1 = -0.104544
[ INFO] [1763348529.948039890]: DataCollector::jointStateCB: joint2 = -1.05104
[ INFO] [1763348529.948069084]: DataCollector::jointStateCB: joint3 = -0.272112
[ INFO] [1763348529.948098906]: DataCollector::jointStateCB: joint4 = -0.004602
[ INFO] [1763348529.948127960]: DataCollector::jointStateCB: joint5 = -0.001534
[ INFO] [1763348529.948156315]: DataCollector::jointStateCB: joint6 = -0.012272
[ INFO] [1763348529.948284403]:  Currrent Pose:
[ INFO] [1763348529.948322397]:     0.704849    0.116268    0.699764    0.357657
[ INFO] [1763348529.948356968]:     0.703502   0.0119131   -0.710593    0.202983
[ INFO] [1763348529.948390282]:   -0.0909558    0.993146   -0.073398    0.343189
[ INFO] [1763348529.948424714]: Currrent Euler angles:
                            pitch(y)=0.0910817, yaw(z)=0.784442, roll(x) = 1.64457
[ INFO] [1763348529.948453209]: DataCollector::jointStatesCB:
                            cameraPositionUpdated_ =true: call CSVWriteRow
[ INFO] [1763348529.948485825]:  Data record counter 147 : delta x=0.0340849
[ INFO] [1763348529.948520466]:      delta y=0.020521
[ INFO] [1763348529.948548472]:      delta z=0.0199845
[ INFO] [1763348529.948590446]:  Data record counter 148 :
                            RMSE: 0.0257, written to csv file
```

and one point data record in the csv:

```
joint_1 joint_2 joint_3 joint_4 joint_5 joint_6
    end_effector_x  end_effector_y  end_effector_z
    end_effector_roll   end_effector_pitch  end_effector_yaw
    depth_3D_x  depth_3D_y  depth_3D_z
-0.104544   -1.05104    -0.272122   -0.004602   -0.001534   -0.012272
        0.357657    0.202983    0.343189
            1.64457 0.0910817   0.84442
                0.32 0.18      0.32
```

We can get that the RMSE for the above record is 0.026m (2.6cm). It has been verified the most valid RMSE ranges are between 0.02m to 0.03m and only very small portion data (certain locations within the workspace) can reach below 0.02m and can not achieve more accurate precision. It has been shown by measurement and verification that camera-sensed data is much more accurate by taking the average of multiple values at some location (typical error distribution: $\Delta x < 0.01m$; $\Delta y < 0.01m$, $\Delta z < 0.01m$). The main errors come from motor servo accuracy and large assembly tolerance. In addition, other errors come from open-loop joint state error (all motors are in position control mode rather than force/torque mode), robot links' mass and weight at different poses introducing different torques to the base of the arm, inaccurate forward kinematic computations due to inaccurate D-H parameters, as well as the hysteresis effect of actuators, etc. The combination of all these errors often causes the end-effector to be wiggled over 1 cm in the same controlled position. After all, Hex7Bot is a low-cost desktop robot arm without closed-loop control, not a high-precision industry manipulator. The author still thinks it is still an ideal platform for learning robotics fundamentals like kinematics by building, programming, and controlling an arm, verifying advanced algorithms, including deep learning and computer vision, with open-source software like ROS.

14.4.3 Training Process

Model Architecture The core of the deep learning approach involves designing, compiling, and training a neural network for regression. A common and effective choice for end-effector pose estimation is a standard feedforward neural network (also known as a multilayer perceptron or `MLP`).

- Input layer: A layer with 6 neurons, corresponding to the 6 joint variables.
- Hidden layers: One or more fully connected hidden layers. For a 6-joint arm, a deeper network might be necessary to capture complex, non-linear relationships. The number of neurons and layers can be tuned with the validation set.
- Output layer: A layer with 6 neurons, representing the 3D position ((x,y,z)) and 3D orientation.

$$[x, y, z, \text{Roll}, \text{Pitch}, \text{Yaw}]$$

- Activation functions: Use an activation function like ReLU in the hidden layers. A linear activation function is typically used in the output layer for regression tasks.

An Example Architecture is shown in Figure 14.5:

```
Input Layer (6 units - joint angles)
    -> Dense Layer (e.g., 128 units, ReLU activation)
    -> Dense Layer (e.g., 64 units, ReLU activation)
    -> Output Layer (e.g., 6 ouput - x, y, z, roll, pitch, yaw
         - linear activation)
```

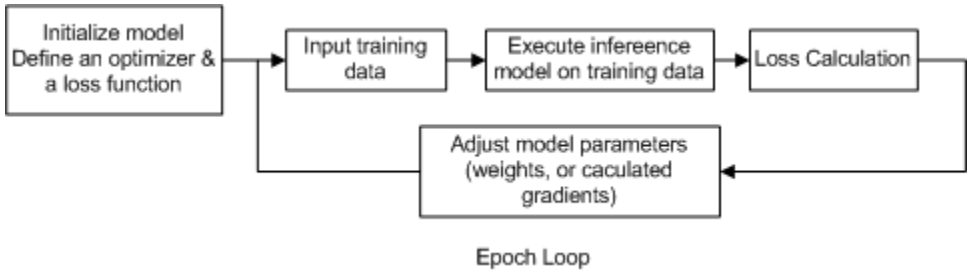

Figure 14.5: Training loop

A typical deep learning training loop can be visualized as a cyclical process, repeatedly refining a model's parameters to minimize a defined loss function. The core steps involved are:

Initialization - Initialize model parameters (such as weights and biases) , often randomly - Define a Loss Function: The loss function quantifies the discrepancy between the model's predictions and the actual target values. Mean Squared Error (`MSE`) is commonly used to minimize the difference between predicted and actual end-effector poses. - Define an Optimizer - In deep learning, an optimizer is an algorithm or method used to adjust the parameters (weights and biases) of a neural network during the training process to minimize the loss function, thereby improving the model's performance and accuracy. Adam(Adaptive Moment Estimation) or RMSprop (Root Mean Square Propagation) are popular choices. Both employ adaptive learning rates. RMSprop maintains an exponentially decaying average of squared gradients for each parameter and the update rule devices the learning rate by the square root of this moving average. Adam builds upon RMSprop by incorporating momentum, which accelerates convergence in relevant directions and dampens oscillations during training.

Epoch Loop Train the model on the training data for a specified number of epochs, using the validation set to monitor for overfitting, where each epoch represents one full pass over the entire training dataset. Until the specified number of epochs is completed or a stopping criterion(e.g., early stopping based on validation performance) is met.

- Forward Pass: Execute inference model on training data by feeding data through the neural network, lay by layer to produce an output prediction.
- Loss Calculation: The predicted output is compared wot he true target label, and a loss function quantifies the discrepancy between them. The loss represents how "wrong" the model's prediction was. Use a regression loss function, such as Mean Squared Error (`MSE`), to measure the difference between the model's predictions and the true end-effector pose.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\widehat{y}_i - y_i)^2$$

where y_i is the ground truth pose and $\widehat{y}_i$ is the model's prediction.

Compare the model's predictions with the true labels (ground truth) using the defined loss function. This calculates a numerical value representing the discrepancy between predictions and actual.

- Parameter Updates: Adjust the model's parameters using the optimizer and the calculated gradients. An optimization algorithm, like Adam, is used to adjust the model's weights and minimize the loss function.

  The optimizer determines how the parameters are updated based on the learning rate and other optimization strategies. For training, a regression loss function is used to minimize the difference between the model's predictions and the true end-effector pose from the training data. Common choices include:

  Mean Squared Error (MSE): Calculates the average squared difference between the predicted and true values. It is sensitive to large errors. Mean Absolute Error (`MAE`): Calculates the average absolute difference. It is more robust to outliers than MSE.

  An optimizer, such as Adam or RMSprop, is used to update the network's weights during training based on the calculated loss.

Please note that the model's performance heavily depends on the size and quality of the training dataset. A diverse set of configurations is needed to cover the entire workspace.

Hyperparameter tuning The number of layers, neurons per layer, and learning rate are all hyperparameters that require careful tuning for optimal performance. Generalization: The model may not generalize well to different robot geometries or arm designs without retraining.

14.4.4 Evaluation and Application

After training, evaluate the final model on the test set using a metric like `MSE` or Root Mean Squared Error (RMSE) to quantify its accuracy.

$$RMCE = \sqrt{\frac{\sum_{i=1}^{n} \left(\text{Predicted}_i - \text{Actual}_i\right)^2}{n}}$$

A lower MSE or RMSE value indicates that the model's prediction are on average, closer to the actual values, meaning higher accuracy.

14.5 Implement Mode with Tensorflow

We can implement this mode in Tensorflow.

- Define function to load and preprocess the train the mode

```
import pandas as pd
import tensorflow as tf
from tensorflow.python.keras import backend as K
from tensorflow import keras
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import numpy as np
import json

def hex7bot_model(csv_file_path):
  # Load data: CSV file into a pandas DataFrame
  df = pd.read_csv(csv_file_path)
  # Assuming your columns are named 'feature1', 'feature2', ..., 'rotZ'
  feature_columns = ['joint_1', 'joint_2', 'joint_3', 'joint_4',
                     'joint_5', 'joint_6']
  label_columns = ['end_effector_x', 'end_effector_y', 'end_effector_z',
        'end_effector_roll', 'end_effector_pitch', 'end_effector_yaw']
  features = df[feature_columns]
  labels = df[label_columns]
  # Split data into training and testing sets
  X_train, X_test, y_train, y_test = train_test_split(features, labels,
        test_size=0.2, random_state=42)
  # Standardize features
  scaler = StandardScaler()
  X_train_scaled = scaler.fit_transform(X_train)
  X_test_scaled = scaler.transform(X_test)
  # Extract parameters JSON file
  scaler_params = {'mean':scaler.mean_.tolist(), 'scale':scaler.scale_.tolist()
     ,'var':scaler.var_.tolist(), 'n_samples_seen':int(scaler.n_samples_seen_)}
  with open('./model/right_arm/scaler_params.json','w') as f:
      json.dump(scaler_params, f, indent=4)
  # Build the Model (a simple DNN regressor) ---
  inputs = tf.keras.Input(shape=(X_train_scaled.shape[1],),
                          name="input_joints")
  # Define hidden layers with specific names
  hidden1 = tf.keras.layers.Dense(128, activation='relu',
                          name="hidden_layer_1")(inputs)
  hidden2 = tf.keras.layers.Dense(64, activation='relu',
                name="hidden_layer_2")(hidden1)
  # Define the output layer with a specific name
  outputs = tf.keras.layers.Dense(6,  name="output_position")(hidden2)
  # Create and compile the mode with 'mean_squared_error' loss
  model = tf.keras.Model(inputs=inputs, outputs=outputs name="position_model")
  model.compile(optimizer='adam', loss='mean_squared_error',
                metrics=['mean_absolute_error'])
  return model, X_train_scaled, X_test_scaled, y_train, y_test
```

- Define a function to freeze the state of a session into a pruned computation graph

```
        # Define a function to load, preprocess, and train the model
        def freeze_session(session, keep_var_names=None, output_names=None,
              clear_devices=True):
        """
        @param session The TensorFlow session to be frozen.
        @param keep_var_names A list of variable names that should not be frozen,
                              or None to freeze all the variables in the graph.
        @param output_names Names of the relevant graph outputs.
        @param clear_devices Remove the device directives from the graph for
                better portability.
        @return The frozen graph definition.
        """
        graph = session.graph
        with graph.as_default():
            freeze_var_names = list(set(v.op.name for v in
                tf.global_variables()).difference(keep_var_names or []))
            print(freeze_var_names)
            output_names = output_names or []
            output_names += [v.op.name for v in tf.global_variables()]
            print(output_names)
            input_graph_def = graph.as_graph_def()
            if clear_devices:
                for node in input_graph_def.node:
                    node.device = ""
            frozen_graph = tf.graph_util.convert_variables_to_constants(
                session, input_graph_def, output_names, freeze_var_names)
            return frozen_graph
```

- Run the train module and then evaluate and predict results.

```
if __name__ == '__main__':
  csv_path="./data_collect/right_arm_20251116.csv"
  model, x_train_data, x_test_data, y_train_data, y_test_data
         = hex7bot_model(csv_path)
  model.summary()
  #Train the Model ---
  history = model.fit(x_train_data, y_train_data, epochs=50,
              batch_size=32, validation_split=0.2, verbose=1)
  # Evaluate the model on the test set
  test_loss, test_mae = model.evaluate(x_test_data, y_test_data, verbose=0)
  print(f"\nTest Mean Absolute Error: {test_mae}")
  # Example prediction on the first 3 test samples
  predictions = model.predict(x_test_data[:3])
  print("\nPredictions (first 3 samples):")
  print(predictions)
  print("Ground Truth (first 3 samples):")
  print(y_test_data[:3])
```

- Save the model and freeze the graph.

```
# Save the entrie model to an HDF5 file that TF1.X supported
model_path = './model/right_arm_tf1_keras_model.h5'
model.save(model_path)
#load pretrained Model
model = tf.keras.models.load_model(model_pat)
model.summary()
# set learning_phase=0 to change to evaluation mode:
K.set_learning_phase(0)
export_path = './model/right_arm'
saver = tf.train.Saver()
# freeze graph
with K.get_session() as sess:
    saver.save(sess, './model/right_arm')
    frozen_graph = freeze_session(sess,
        output_names=[out.op.name for out in model.outputs])
    tf.train.write_graph(frozen_graph, export_path,
            "right_arm.pb", as_text=False)
```

- Verify the trained results by reloading the trained model.

```
# debugging code : reload exported model
 with tf.gfile.GFile(export_path + '/right_arm.pb', "rb") as f:
    graph_def = tf.GraphDef()
    graph_def.ParseFromString(f.read())

 with tf.Graph().as_default() as graph:
    tf.import_graph_def(graph_def)

 for op in graph.get_operations():
    if op.type == "Placeholder":
        print(op.name)
    print(op.name, [inp for inp in op.inputs])
```

Run this script in a virtual environment (Python3.6 and Tensorflow1.5):

- DNN model

```
Layer (type)                 Output Shape              Param #
=================================================================
input_joints (InputLayer)    (None, 6)                 0
_________________________________________________________________
hidden_layer_1 (Dense)       (None, 128)               896
_________________________________________________________________
hidden_layer_2 (Dense)       (None, 64)                8256
_________________________________________________________________
output_position (Dense)      (None, 6)                 390
=================================================================
```

- Training loops (Epoch=50)

```
...
Epoch 48/50
32/90 [=========>....................]32/90 [=========>....................]
    - ETA: 0s - loss: 90/90 [==============================]90/90
    - 0s 144us/step - loss: 3.1175e-04 - mean_absolute_error: 0.0082
    - val_loss: 0.0180 - val_mean_absolute_error: 0.0303

Epoch 49/50
32/90 [=========>....................]32/90 [=========>....................]
    - ETA: 0s - loss: 3.0621e-04 - mean_90/90
    - 0s 201us/step - loss: 2.9756e-04 - mean_absolute_error: 0.0081
    - val_loss: 0.0179 - val_mean_absolute_error: 0.0300

Epoch 50/50
32/90 [=========>....................]32/90 [=========>....................]
    - ETA: 0s - loss: 3.9546e-04 - mean_90/90
    - 0s 98us/step - loss: 2.8487e-04 - mean_absolute_error: 0.0079
    - val_loss: 0.0179 - val_mean_absolute_error: 0.0297


Test Mean Absolute Error: 0.009665458463132381
```

- Predict first 3 samples in the test set and compare with ground truth values.

```
Predictions (first 3 samples):
[[ 0.367989   -0.06149229  0.38556835 -1.6377324   0.01782972 -0.5023024 ]
 [ 0.35028106 -0.11659154  0.35767555 -1.5904398   0.02635893 -0.573172  ]
 [ 0.35219586 -0.11499584  0.36076573 -1.5944114   0.02198265 -0.55812407]]
Ground Truth (first 3 samples):
[[ 0.340126  -0.0710356  0.362745  -1.58957    0.0262983 -0.456872 ]
 [ 0.3577    -0.117241   0.361988  -1.59398    0.0238024 -0.574992 ]
 [ 0.35469   -0.112814   0.362117  -1.59337    0.0241906 -0.558119 ]]
```

The above model was run on an old machine with Ubuntu 18.04 and TensorFlow 1.5 installed. It not only saves the model in SavedModel format but also freezes the trained TensorFlow graph by saving the entire network structure and weights into a single self-contained Protocol Buffers (.pb) file. This file is optimized for inference (prediction) and deployment, as it removes unnecessary training-related nodes and simplifies the model structure and it is easier to deploy the model on different platforms, especially C++ environments.

Please note that TensorFlow 1.x needs to run the extra freeze_graph.py script so that the final frozen pb file can be used by different platforms. The script function is realized in the `freeze_session` function to generate the final frozen .pb file directly.

For TensorFlow 2. Saving the model in Python directly will be much easier:

```
import tensorflow as tf
from tensorflow import keras
#... (define and train your model)...
model.save('path/to/saved_model_folder')
```

14.5.1 Export Trained Model For ROS

If your ROS nodes are developed with Python, you can embed the TensorFlow code directly in the ROS node. To load a TensorFlow model trained in Python into a C++ program within ROS, you need to export the model from Python in a compatible format (like the SavedModel format) and then use the TensorFlow C++ API or a wrapper library (like tensorflow_ros_cp) to load and run it.

You can simplify the complex build process of the native C++ TensorFlow API by using existing ROS packages. The `tensorflow_ros_cpp` package is recommended, as it handles much of the configuration for you. However, the `tensorflow_ros_cpp` node on Melodic has C++ ABI mismatch issue when using the pip-installed TensorFlow, which can cause linking errors. Another common problem is dependency or mismatched versions between Python, TensorFlow, protobuf, numpy etc. To overcome these issues, we are seeking to install all compatible packages and their dependencies in an isolated environment, such as Docker. For our code, we use a virtual Python environment to install Python 3.6 and TensorFlow 1.5.0:

- Download and compile TensorFlow 1.5.0 source code for Python, C++ libraries. To build old TensorFlow 1.X, we sometimes must downgrade compiler tools such as Bazel down to 0.10.0 first.

```
git clone https://github.com/tensorflow/tensorflow.git tensorflow
cd tensorflow
git checkout v1.5.0
./configure
```

- Compile and build all TensorFlow libraries for C , C++ and python (libtensorflow.so, libtensorflow_cc.so, libtensorflow_framework.so)

```
bazel build -c opt  //tensorflow:libtensorflow_cc.so
bazel build -c opt  //tensorflow:libtensorflow
bazel build -c opt  //tensorflow:libtensorflow_framework.so
bazel build -c opt //tensorflow/core/distributed_runtime/rpc:grpc_tensorflow_server
    //tensorflow/tools/pip_package:build_pip_package
```

- Install all these libraries to virtual environment (such as *~/tensorflow_env/tf15_py36/lib*) instead of system folder (*/usr/local/include*).

- Ensure you have the TensorFlow C++ libraries(*libtensorflow_cc.so* and *libtensorflow_frame.so*) are linked in the proper location (e.g. *~/tensorflow_env/tf15_py36/lib*) and all included header files are pointed to the folder in the virtual environment. For example, when we create a C++ node to load and run the mode, use the following script to compile your C++ node or add them into *CMakefile.txt* if you use CMake.

```
#!/bin/bash
BUILD_CONFIG_PATH="/home/ptshi/tensorflow_env/tf15-py36/lib:/usr/local/lib"
export LD_CONFIG_PATH="$BUILD_CONFIG_PATH"
g++ -I /home/ptshi/tensorflow_env/tf15-py36/lib/python3.6/
    site-packages/tensorflow/include
    -I /home/ptshi/tensorflow/
    -I /home/ptshi/tensorflow_env/tf15-py36/lib/python3.6/
    site-packages/tensorflow/include/external/nsync/public/
    -L/home/ptshi/tensorflow_env/tf15-py36/lib
    trained_model.cpp
    -ltensorflow_cc
    -ltensorflow_framework
    -Wall -g  -std=c++11
    -o trained_model
```

2. Develop your C++ node to run inference

There are two typical methods to load trained models in TensorFlow C++. One is a higher-level function for loading models that are generated with Python `model.save()` or `tf.saved_model.save()` and popular in TensorFlow 2.x. However, for older TensorFlow 1.x on Ubuntu 18.04, the supported workflow is to load a GraphDef (which defines the model's architecture)from a standalone *.pb* file over `ReadBinaryProto` . This is a low-level utility function used to read arbitrary binary-formatted Protocol Buffer(protobuf) messages from the filesystem and is supported by TensorFlow 2.x.

Let's create a C++ node to load the frozen graph that we created and run the inference on it.

- Load frozen graph (*.pb*) file :

```
#include <tensorflow/cc/saved_model/tag_constants.h>
#include "tensorflow/core/framework/types.h"
#include <tensorflow/cc/saved_model/loader.h>
#include <tensorflow/core/public/session.h>
#include "tensorflow/core/platform/env.h"
#include <iostream>
#include <fstream>
#include <string>
#include <nlohmann/json.hpp> // Include json header

using namespace tensorflow;
using json = nlohmann::json;

const char* right_arm_pb=
         "/home/ptshi/projects/tensorflow/src_py/model/right_arm/right_arm.pb";
int main()
{
  const int N_Joints = 6;
   // 1. Create a Session
   tensorflow::Session* session;
   tensorflow::Status status = tensorflow::NewSession
               (tensorflow::SessionOptions(), &session);
   if (!status.ok()) {
      std::cerr << "Failed to create new Session:  "
              << status.ToString() << std::endl;
      return 1;
   }
   // 2. Read the frozen graph
   tensorflow::GraphDef graph_def;

   status = tensorflow::ReadBinaryProto(tensorflow::Env::Default(),
               right_arm_pb, &graph_def);
   if (!status.ok()) {
       std::cerr << "Failed to read the frozen graph:  "
                 << status.ToString() << std::endl;
       return 1;
   }
```

We should know that a frozen graph only stores the structure of the graph itself and does not handle normalized input variables, various weights, or model metadata (like signatures for inputs/outputs), we have to implement additional code to process the input/weights prior to calling the session for inference. The last Python code snippet, dumping `StandardScaler` parameters to a JSON file, is for this purpose:

```
{
   "mean": [
       0.042761946902654856,
       -0.4326670088495576,
       0.14390151769911508,
       -0.010738,
       -0.012041221238938056,
       -0.0015340000000000004
   ],
   "scale": [
       0.001192303265460046,
       0.13856382626467167,
       0.1612119212216625,
       1.0,
       0.0013416824580764145,
       4.336808689942018e-19
   ],
   "var": [
       1.4215870768266887e-06,
       0.01919993394910612,
       0.025989283543979513,
       0.0,
       1.8001118183099696e-06,
       1.88079096131566e-37
   ],
    "n_samples_seen": 113
}
```


- Create the module and input tensor.

```
//  Load the graph into the session
  status = session->Create(graph_def);
  if (!status.ok()) {
      std::cerr << "Failed to load graph into the session:  "
      << status.ToString() << std::endl;
      return 1;
  }
 //get standard scaler used by python
 std::ifstream spf("./model/right_arm/scaler_params.json");
 json jf =json::parse(spf);

 //Get mean values
 auto means { jf["mean"].get<std::vector<float>>()};
 //Get scales
 auto scales { jf["scale"].get<std::vector<float>>()};

 //Prepare Input Tensors
 std::string input_node_name = "input_joints_1:0";
 // Define the shape: 1 sample (batch size), 6 features
 tensorflow::TensorShape shape({1, N_Joints});
 //Define the shape of your input sensor (e.g., a 1XN vector)
 Tensor input_tensor(DT_FLOAT, shape);
 //Fill the tensor with your actual data
 auto input_tensor_data=input_tensor.flat<float>();
 if ( means.size() >= N_Joints && scales.size() >= N_Joints) {
    input_tensor_data(0) = ( 0.10 - means[0]) / scales[0]; //joint 1
    input_tensor_data(1) = (-1.05 - means[1]) / scales[1]; //joint 2
    input_tensor_data(2) = (-0.27 - means[2]) / scales[2]; //joint 3
    input_tensor_data(3) = (-0.004 - means[3]) / scales[3]; //joint 4
    input_tensor_data(4) = (-0.001 - means[4]) / scales[4]; //joint 5
    input_tensor_data(5) = (-0.012 - means[5]) / scales[5]; //joint 6
  }
  else {
      std::cerr << "Failed to read Standard Scaler used by model" << std::endl;
  }
```

- Run the model for inference.

You can perform inference by preparing inputs as tensorflow::Tensor objects and running the session with the appropriate input and output tensor names.

```
  //Define input and output names
   std::vector<std::pair<std::string, Tensor>> inputs = {{input_node_name,
        input_tensor}};
   std::vector<std::string> output_nodes_names = {"output_position_2/BiasAdd:0"};
   std::vector<tensorflow::Tensor> outputs;
   // Run the session for Inference:
   status = session->Run(inputs, output_nodes_names, {}, &outputs);
   if (!status.ok()) {
      std::cerr << "Error running model: " << status.ToString() << std::endl;
   }
   for (size_t i = 0; i < outputs.size(); ++i) {
      const tensorflow::Tensor& tensor = outputs[i];
      std::cout << "Tensor " << i << " (Shape: " << tensor.DebugString() << "):"
                << std::endl;
    }
   //Remember to close the session
   session->Close();

   return 0;
}
```

Using same normalized inputs (X_test_scaled) you can get the similar result as the last predicted data.

```
Input (shape: [1,6]) = 0.1594 1.52844 0.494296 0  -0.172007  1
Running session
Load session-> Run
Tensor 0 (Shape: Tensor< type: float shape: [1,6]
0.352852 -0.091336 0.368253 -1.65802 0.0238131 -0.492769
```

Please note that since there are no enough samples (only 113 samples for learning) in the last example, we use StandardScaler parameters from the JSON file to get normalized data.

$$x' = \frac{x - \bar{x}}{\sigma} \tag{14.5.1}$$

It will create very large errors since the standard deviation of training samples is very small (such as the scale at joint6 close to zero). The next job is to improve our model with new network modules, and activation functions without normalized inputs or train the model with more normal distribution data to get more accurate/robust results.

14.6 Deep Learning For Position Tracking

Deep learning is also a highly effective approach for solving the inverse kinematics (`IK`) problem, which is the task of estimating the 6 joint positions of a robotic arm for a given end-effector position. Traditional analytical and numerical methods for IK can be computationally expensive and struggle with complex, high degree-of-freedom (`DOF`) systems and redundancies. Deep learning offers several advantages by directly learning the complex, nonlinear mapping from end-effector pose to joint angles.

Deep learning for estimating the 6 joint positions from a given end-effector position involves training a neural network with a dataset of end-effector positions and their corresponding joint angles, a process known as inverse kinematics. The network learns the complex, non-linear relationship between the 3D end-effector pose and the 6 joint angles, which are then predicted by the trained model. This deep learning approach offers a more robust and often a more efficient solution than traditional analytical or numerical methods, especially for complex robotic systems.

1. Data Acquisition: We can reuse same dataset of robot configurations, mapping various end-effector (x, y, z) positions and orientations (features) to the precise 6 joint angles (ground truth) as a data set.

2. Model Architecture: Design and select a deep neural network to learn the relationship between inputs and outputs, such as a feedforward neural network, one of the most common and straightforward approaches, or a more complex hybrid model like reinforcement learning (RL). The input layer of the network should correspond to the end-effector pose (e.g., a 6-dimensional vector for position and Euler angles). The output layer should match the number of joints, with each output representing a joint angle.

3. Training: Train the network using the collected data to minimize the error between the predicted and actual joint angles. Feed the generated dataset into the neural network. Use a loss function, such as Mean Squared Error (`MSE`), to measure the difference between the network's predicted joint angles and the actual joint angles from the dataset. Train the model using an optimizer (e.g., `Adam`) to minimize the loss, iteratively adjusting the network's weights to improve accuracy.

4. Prediction: After training, the network can take a new end-effector position as input and predict the corresponding 6 joint positions.

5. Evaluate and deploy: Validate the trained model using a separate test set of data to evaluate its performance on unseen end-effector positions. The trained model can then be deployed for real-time control, where a desired end-effector pose is fed into the network, and the predicted joint angles are used to move the robot.

We can update last chapter's robot tracking example by combining the deep learning example in this chapter, which will eliminate the issue of resolving robot inverse kinematics and in-accurate parameters issue. The author plans to update all ROS projects to latest ROS2 , Ubuntu OS and TensorFlow first and then upgrade these samples (in the future). This task will leave interested readers.

14.7 Summary

In this chapter, we learned some fundamental concepts of deep learning and learned how to develop and implement a deep learning model (i.e., FNN) with TensorFlow into our dual-arm system in two basic areas: one is to estimate robot arm end-effector pose (position and orientation) without solving forward kinematics. Another scenario is target tracking: given an object's 3D position in the robot workspace, how to control all joint positions so that the robot's end-effector can reach and grab the object without computing inverse kinematics.

Unlike some analytical methods, deep learning does not require a closed-form solution, making it applicable to a wider range of robot morphologies. The system can learn unique mechanical characteristics and be calibrated in a highly specialized way, potentially identifying critical arm positions and singularities, which is especially suitable for developing advanced rehabilitation robots that can precisely track desired arm trajectories for therapeutic purposes.

Deep learning technology continues to advance with faster hardware and more sophisticated algorithms like reinforcement learning or models. Especially with the advancement in AI, more and more self-learning robots have emerged with abilities of bipedal running or gymnastic maneuvers. This chapter is just a prelude or starting point for you to explore next-level intelligence.