Università degli Studi di Padova

Scuola di ingegneria

Corso di Laurea in ingegneria dell'informazione

Hand gesture recognition e controllo del robot LEGO Mindstorms NXT applicato alla robotica educativa

Laureando

Gabriele Pozzato

Relatore

Prof. Emanuele Menegatti

Correlatore

Dott. Stefano Michieletto

Anno Accademico 2012/2013

a Luca

Indice

Elenco delle figure

Elenco delle tabelle

Sommario

L'obiettivo della tesi è quello di permettere ad un dato utente, adulto o bambino, di interagire nel modo più naturale possibile con una macchina e, nello specifico, con un robot.
Lo scopo principale della ricerca sarà unire pose e gesture recognition attraverso ROS (Robot Operating System) per controllare un robot NXT della serie LEGO Mindstorms.
L'utente sarà quindi in grado di interagire col robot, nel caso in esame, giocando con esso una serie di manches di morra cinese.
Tale robot opererà la scelta del gesto da giocare tramite algoritmi di Machine Learning basati su metodi statistici ed in particolare su GMM (Gaussian Mixture Model [15]).

Capitolo 1

Introduzione

Negli ultimi anni, grazie ad un continuo sviluppo tecnologico e alla crescente disponibilità di apparecchiature a basso costo, la frequenza nelle interazioni uomo-macchina è aumentata considerevolmente.
Sensori quali Microsoft Kinect hanno permesso una grande crescita nella Computer Vision (CV) e, in particolare, in campi come lo skeletal tracking e il riconoscimento di pose (pose recognition) e gesti (gesture recognition) dando origine a numerosi approcci basati sull'utilizzo di sensori di profondità, come quello usato da K.K. Biswas e Saurav Kumar Basu [13] o da Yi Li [16].
Oltre a ciò, l'interazione uomo-macchina è divenuta di fondamentale importanza e di grande interesse in particolare quando si parla di robot.
La comunicazione Uomo-Robot permette, non solo di poter far compiere al robot determinate azioni prestabilite, ma anche che sia il robot stesso a comunicare con l'utente umano, operando la scelta migliore in base ad una esperienza passata[14] o cercando di riconoscere lo stato d'animo di una persona a seconda della sua postura per comportarsi di conseguenza (Social Human-Robot interaction [17]).
Da qui si sviluppa la tematica della ricerca di cui si discuterà nei prossimi capitoli, nella quale il controllo del robot LEGO Mindstorms NXT si fonde alla percezione del sensore Microsoft Kinect per sviluppare un'applicazione nel ramo della robotica educativa.
Lo scopo centrale sarà perciò riuscire a coinvolgere utenti inesperti, adulti o bambini, nel campo della robotica, permettendo loro di interagire col robot, nel caso specifico, giocare a morra cinese (Rock-Paper-Scissors).
Gli utenti, pur non conoscendo approfonditamente il funzionamento intrinseco degli strumenti utilizzati, impareranno ad utilizzarli e a comunicare con essi.

Capitolo 2

Descrizione

2.1 Obiettivi

L'obiettivo principale di questa ricerca è quello di permettere ad un dato utente, adulto o bambino, di interagire nel modo più naturale possibile con una macchina e, nello specifico, con un robot.
Da un punto di vista prettamente realizzativo lo scopo è perciò quello di unire l' **hand gesture recognition (HGR)**, ossia il riconoscimento gestuale, effettuato tramite il sensore Microsoft Kinect® con il controllo del robot LEGO Mindstorms NXT®.
Tramite il set predefinito di tre gesti riportato in fig.2.1, l'utente umano sarà in grado di interagire col robot giocando a morra cinese. Sarà inoltre possibile trasmettere al robot delle quantità tramite un set predefinito di tre posizioni riportate in figura 2.2.

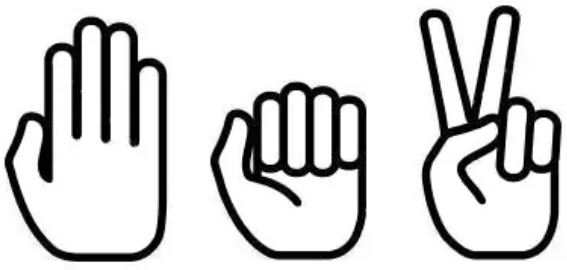


Figura 2.1: Paper, Rock e Scissors (da sinistra verso destra)

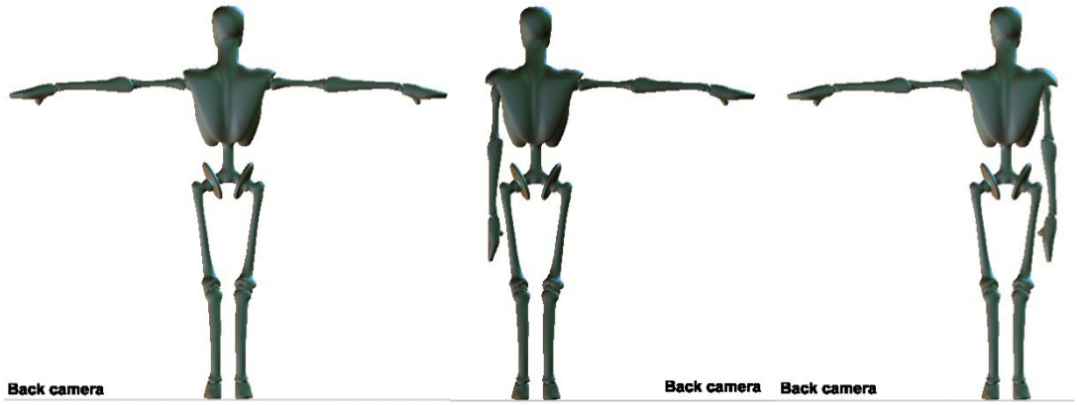


Figura 2.2: Tpose, Rpose e Lpose (Back camera view)

2.2 Strumenti utilizzati

Di seguito viene riportata una breve descrizione degli strumenti hardware e software utilizzati.

2.2.1 Robot LEGO Mindstorms NXT

LEGO Mindstorms NXT[1] è un kit di robotica programmabile rilasciato dall'omonima azienda nel 2006.

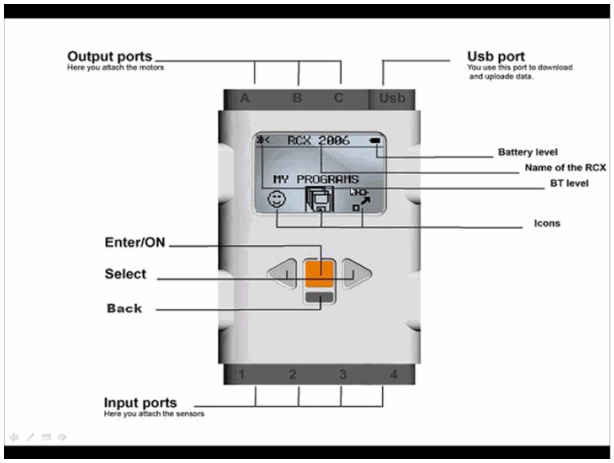


Figura 2.3: NXT

Di seguito sono elencate le specifiche tecniche dell' "NXT brick" (Figura 2.3), il blocco fondamentale del kit, tramite il quale è possibile controllare i

motori (fino ad un massimo di tre) e i sensori che possono essere di prossimità, di luce, di colore, di suono e ad ultrasuoni.

- microprocessore 32-bit ARM7 (256 KB flash memory, 64 KB RAM)
- microcontrollore 8-bit AVR (4 KB FLASH, 512 Bytes RAM)
- Display LCD con matrice da 100×64 pixel
- 1 porta USB 1.1 (12 Mbit/s)
- Connettività wireless Bluetooth
- 4 porte di input
- 3 porte di output

2.2.2 Robot Operating System: ROS

ROS[2] è un framework open source che permette in primo luogo l'astrazione dall'hardware utilizzato, il controllo a basso livello dei dispositivi, l'implementazione mediante linguaggi di programmazione ad alto livello quali C++ (librerie del package roscpp) e Python (librerie del package rospy) e il passaggio di messaggi tra i vari processi in esecuzione.
ROS fornisce inoltre librerie e strumenti per la compilazione e l'esecuzione del codice.

La caratteristica fondamentale di ROS è la modularità, ossia la possibilità di controllare diverse parti del robot tramite nodi ognuno avente una funzione specifica. Grazie all'architettura distribuita di ROS, ogni nodo può interagire con gli altri senza dover prima comunicare con il nodo centrale o master, ciò rende efficiente e robusta la programmazione.
I vari nodi possono infatti comunicare con gli altri tramite messaggi(data structures), pubblicando o sottoscrivendo dei determinati topic. (Per ulteriori informazioni vedi [2].)
Perciò, dopo aver inizializzato (Listing 2.1) un nodo, ROS permette all'utente di controllarne i relativi topic tramite meccanismi di publishing e subscribing.

Tramite i meccanismi di publishing è quindi possibile pubblicare dati in un determinato topic e, per esempio, azionare dei motori. Tramite i meccanismi di subscribing è possibile sottoscrivere uno specifico topic per esempio per conoscere la posizione assunta da un utente.

Distribuzione ROS utilizzata:

- ROS Groovy Galapagos [3] (31-12-2012)

```
...
  ros::init(argc, argv, "nxt_kinect");
...
```

Listing 2.1: Inzializzare un nodo in ROS (C++)

```
...
    //Subscribe
    sub = m.subscribe("pose", 1, Callback);
    //Publish
    pub = n.advertise<std_msgs::Float64>("angle_c", 1);
...
```

Listing 2.2: Publishing e subscribing in ROS (C++)

2.2.3 Microsoft Kinect [12]

Negli ultimi anni la produzione di sensori a basso costo ha permesso un grande sviluppo in campi quali lo skeletal tracking e il riconoscimento gestuale (gesture recognition).
Uno dei dispositivi più conosciuti è il sensore Kinect sviluppato da Microsoft (Figure 2.4) e in commercio dal 2010. É costituito da una telecamera RGB, da un sensore di profondità e da un array di microfoni (Tabella 2.1).

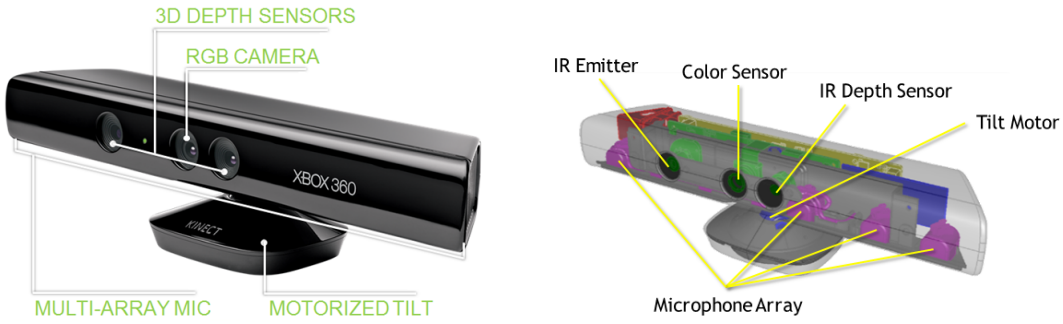


Figura 2.4: Microsoft Kinect

Come si può immaginare, lo sviluppo di algoritmi per Hand Gesture Recognition è legato principalmente al sensore di profondità che stima la posizione rispetto ad un oggetto, animato o meno, attraverso le sue coordinate 3D (x, y, z).

Sono disponibili diversi framework software per l'utilizzo di Kinect in piattaforme differenti dall' Xbox 360 per cui era stato originariamente ideato:

- Microsoft SDK [4]
- OpenNI [5]
- OpenKinect [6]

In tale ricerca il riconoscimento gestuale attuato tramite Kinect è stato sviluppato utilizzando le librerie OpenNI.

Kinect	**Specifiche**
Campo visivo	43° verticale per 57° orizzontale
Capacità di inclinazione	± 27°
Frame rate (profondità e colore)	30 frame al secondo (FPS)
Formato audio	16kHz, 24bit mono pulse code modulation (PCM)
Caratteristiche audio di input	Array di quattro microfoni con convertitore analogico-digitale a 24bit (ADC) ed elaborazione del segnale compresa la cancellazione dell'eco acustico e la soppressione del rumore
Caratteristiche accelerometro	Accelerometro 2G/4G/8G configurato per 2G, con limite superiore di precisione dell'1%

Tabella 2.1: Specifiche Kinect

Capitolo 3

Pose & Hand Gesture Recognition

Come discusso in precedenza l'obiettivo primo della ricerca è quello di controllo del robot LEGO Mindstorms NXT, al fine di farlo giocare a morra cinese (Rock-paper-scissors) contro un avversario umano.
Per raggiungere tale scopo si è partiti da skeletal tracking e pose recognition, necessari per trasmettere delle quantità al robot, per poi passare in successione alla parte centrale del progetto ossia l'HGR.

3.1 Pose Recognition

Per prima cosa, quando un nuovo utente si posiziona davanti al sensore Kinect ad una distanza di circa 2m, per calibrare il sistema e permettere il tracking, questi deve assumere la Psi Pose (Figura 3.1). In questo modo vengono riconosciuti lo scheletro umano e i suoi 24 giunti (Listing 3.1) rendendo possibile il riconoscimento dei movimenti corporei.

```
1  jointFrameMap.insert(boost::bimap<int,std::string>::
2      value_type(headSkeletonJoint,"head"));
3  jointFrameMap.insert(boost::bimap<int,std::string>::
4      value_type(neckSkeletonJoint,"neck"));
5  jointFrameMap.insert(boost::bimap<int,std::string>::
6      value_type(torsoSkeletonJoint,"torso"));
7  jointFrameMap.insert(boost::bimap<int,std::string>::
8      value_type(waistSkeletonJoint,"waist"));
9
10 jointFrameMap.insert(boost::bimap<int,std::string>::
11     value_type(leftCollarSkeletonJoint,"left_collar"));
12 jointFrameMap.insert(boost::bimap<int,std::string>::
13     value_type(leftShoulderSkeletonJoint,"left_shoulder"));
14 jointFrameMap.insert(boost::bimap<int,std::string>::
```

```
15      value_type(leftElbowSkeletonJoint,"left_elbow"));
16  jointFrameMap.insert(boost::bimap<int,std::string>::
17      value_type(leftWistSkeletonJoint,"left_wist"));
18  jointFrameMap.insert(boost::bimap<int,std::string>::
19      value_type(leftHandSkeletonJoint,"left_hand"));
20  jointFrameMap.insert(boost::bimap<int,std::string>::
21      value_type(leftFingertipSkeletonJoint,"left_fingertip"));
22
23  jointFrameMap.insert(boost::bimap<int,std::string>::
24      value_type(rightCollarSkeletonJoint,"right_collar"));
25  jointFrameMap.insert(boost::bimap<int,std::string>::
26      value_type(rightShoulderSkeletonJoint,"right_shoulder"));
27  jointFrameMap.insert(boost::bimap<int,std::string>::
28      value_type(rightElbowSkeletonJoint,"right_elbow"));
29  jointFrameMap.insert(boost::bimap<int,std::string>::
30      value_type(rightWistSkeletonJoint,"right_wist"));
31  jointFrameMap.insert(boost::bimap<int,std::string>::
32      value_type(rightHandSkeletonJoint,"right_hand"));
33  jointFrameMap.insert(boost::bimap<int,std::string>::
34      value_type(rightFingertipSkeletonJoint,"right_fingertip"));
35
36  jointFrameMap.insert(boost::bimap<int,std::string>::
37      value_type(leftHipSkeletonJoint,"left_hip"));
38  jointFrameMap.insert(boost::bimap<int,std::string>::
39      value_type(leftKneeSkeletonJoint,"left_knee"));
40  jointFrameMap.insert(boost::bimap<int,std::string>::
41      value_type(leftAnkleSkeletonJoint,"left_ankle"));
42  jointFrameMap.insert(boost::bimap<int,std::string>::
43      value_type(leftFootSkeletonJoint,"left_foot"));
44
45  jointFrameMap.insert(boost::bimap<int,std::string>::
46      value_type(rightHipSkeletonJoint,"right_hip"));
47  jointFrameMap.insert(boost::bimap<int,std::string>::
48      value_type(rightKneeSkeletonJoint,"right_knee"));
49  jointFrameMap.insert(boost::bimap<int,std::string>::
50      value_type(rightAnkleSkeletonJoint,"right_ankle"));
51  jointFrameMap.insert(boost::bimap<int,std::string>::
52      value_type(rightFootSkeletonJoint,"right_foot"));
```

Listing 3.1: jointFrameMap dello scheletro umano, classe SkeletonPublisher.cpp (C++)

Dopo avere effettuato il tracking dell'utente è stato quindi necessario implementare delle funzioni al fine di riconoscere le quattro posizioni prestabilite definite nel seguente modo :

- Tpose (Figura 3.5),
- Ipose (Figura 3.4)

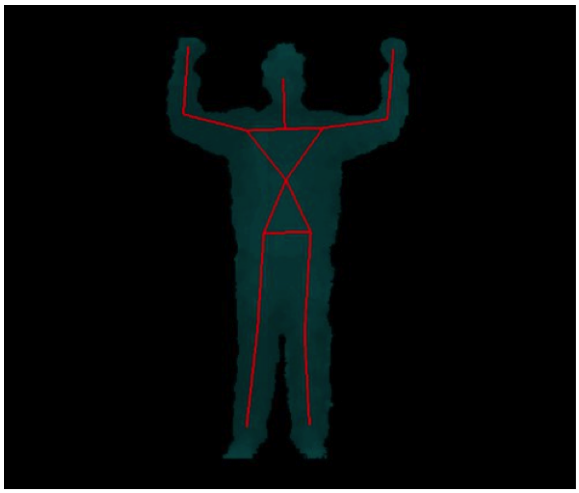


Figura 3.1: Psi (Ψ) Pose

- Lpose (Figura 3.7),
- Rpose (Figura 3.6),

con $S = \{$Tpose,,Ipose,Rpose,Lpose$\}$ set delle posizioni considerate.
Per l'insieme considerato è stato di fatto necessario comprendere la posizione di entrambe le braccia di ogni individuo presente nella scena rispetto al sistema *\openni_rgb_optical_frame* preso come riferimento universale U. (Figura 3.2)

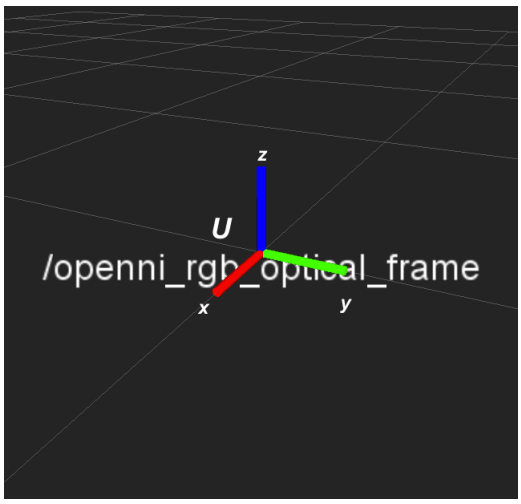


Figura 3.2: \openni_ rgb_ optical_ frame

Sono state quindi utilizzate le posizioni relative rispetto U dei seguenti frame:

- braccio sinistro:
  - $\backslash skeleton\_i\_left\_hand$
  - $\backslash skeleton\_i\_left\_elbow$
  - $\backslash skeleton\_i\_left\_shoulder$
- braccio destro:
  - $\backslash skeleton\_i\_right\_hand$
  - $\backslash skeleton\_i\_right\_elbow$
  - $\backslash skeleton\_i\_right\_shoulder$

con $i \in \mathbb{N}$ indice indicante lo scheletro in tracking.
Ad ogni frame corrisponde un identificatore ID di tipo **integer** che corrisponde, a sua volta, al giunto di interesse, si ha perciò una corrispondenza biunivoca tra frame e ID del giunto, 24 frames distinti numerati da 1 a 24.
In Tabella 3.1 sono riportati gli indici corrispondenti ai giunti di interesse per questo lavoro.

Frame	**ID giunto**
$skeleton\_i\_left\_shoulder$	6
$skeleton\_i\_left\_elbow$	7
$skeleton\_i\_left\_hand$	9
$skeleton\_i\_right\_shoulder$	12
$skeleton\_i\_right\_elbow$	13
$skeleton\_i\_right\_hand$	15

Tabella 3.1: Corrispondenza frame-ID nei casi di interesse

3.1.1 Coordinate dei frame di interesse

Utilizzando i metodi forniti dalle classi *ActionModel::SkeletonPublisher* e *ActionModel::SkeletonReader* per la gestione dei giunti dell'utente e la libreria *tf* [7] di ROS necessaria per tenere traccia delle coordinate dei frame nel tempo, si sono determinate le posizioni relative dei frames di interesse rispetto

al sistema di riferimento U. (Listing 3.2)

```
...
void indexer(ActionModel::SkeletonPublisher& pub,
    std::vector<int>& indexL, std::vector<int>& indexR){
    indexL.push_back(pub.getJointFromFrame(std::string(L) + S));
    indexL.push_back(pub.getJointFromFrame(std::string(L) + E));
    indexL.push_back(pub.getJointFromFrame(std::string(L) + H));
    indexR.push_back(pub.getJointFromFrame(std::string(R) + S));
    indexR.push_back(pub.getJointFromFrame(std::string(R) + E));
    indexR.push_back(pub.getJointFromFrame(std::string(R) + H));
}
...
void getCoord(ActionModel::SkeletonReader& read,
    std::vector<int>& index, std::vector<float>& x,
    std::vector<float>& y){
  tf::StampedTransform transform;
  while(!index.empty()){
      transform = read.readTransform(index.back());
      x.push_back(transform.getOrigin().x());
      y.push_back(transform.getOrigin().y());
      index.pop_back();}
  }
...
```

Listing 3.2: pose_LS.cpp (C++)

Il metodo ***void indexer(<parameters>)*** implementato nel file eseguibile *pose_LS.cpp* permette di ricavare l'ID dei giunti a partire dai frame di interesse. Il metodo ***void getCoord(<parameters>)***, implementato anch'esso nel file eseguibile sopra citato, permette invece di ottenere le coordinate $(x, y) \in \mathbb{R}^2$ dell'origine dei sei giunti, omettendo di fatto la coordinata z poichè indicante la distanza tra utente e piano (x,y) di U, approssimativamente uguale per tutti i giunti corporei.
In figura 3.3 viene evidenziata la proiezione dei punti tridimensionali A, B e C nel piano bidimensionale di U. I punti vengono ridenominati con A', B' e C'.

Dopo esserci portati nel piano bidimensionale per la determinazione delle pose si è applicata la procedura riportata nello pseudocodice 3.1 e discussa nel paragrafo seguente.

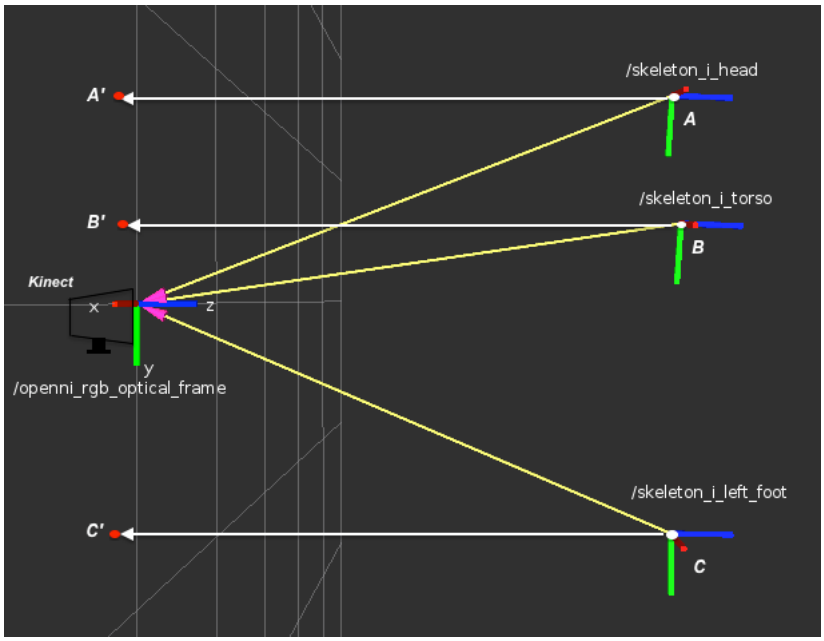


Figura 3.3: Proiezione dei punti sul piano (x,y) di U

3.1.2 Determinazione delle pose

A partire da $\mathbf{x}_R$ e $\mathbf{y}_R$, vettori di coordinate rispettivamente x e y dei tre giunti del braccio destro, si è applicato il modello di stima ai minimi quadrati non pesati (Least Squares) per fare un fit lineare secondo il modello:

$$y(t) = g(t, \mathbf{p}), \tag{3.1}$$

dove $\mathbf{p}$ = [m,q] è il vettore dei parametri incogniti da cui segue

$$y(t) = mt + q. \tag{3.2}$$

Dando per noto il metodo dei minimi quadrati possiamo procedere con la stima parametrica:

$$m = \frac{\frac{1}{N}\sum_{i=1}^{N} z_i \sum_{i=1}^{N} t_i - \sum_{i=1}^{N} z_i t_i}{\frac{1}{N}(\sum_{i=1}^{N} t_i)^2 - \sum_{i=1}^{N} t_i^2}, \tag{3.3}$$

$$q = \frac{1}{N}\sum_{i=1}^{N} z_i - m\frac{1}{N}\sum_{i=1}^{N} t_i, \tag{3.4}$$

dove t_i e z_i sono noti e corrispondono rispettivamente alle coordinate $x_i \in \mathbf{x}_R$ e $y_i \in \mathbf{y}_R$ ed $N = |\mathbf{x}_R| = |\mathbf{y}_R| = 3$.

Esprimiamo quindi la 3.3 e la 3.4 in funzione dei parametri considerati e otteniamo:

$$\begin{aligned} m_R &= \frac{\frac{1}{3}\sum_{i=1}^{3} y_i^R \sum_{i=1}^{3} x_i^R - \sum_{i=1}^{3} y_i^R x_i^R}{\frac{1}{3}(\sum_{i=1}^{3} x_i^R)^2 - \sum_{i=1}^{3}(x_i^R)^2}, \\ q_R &= \frac{1}{3}\sum_{i=1}^{3} y_i^R - m\frac{1}{3}\sum_{i=1}^{3} x_i^R. \end{aligned} \tag{3.5}$$

Equivalentemente per $\mathbf{x}_L$ e $\mathbf{y}_L$, le rispettive coordinate del braccio sinistro, otteniamo:

$$\begin{aligned} m_L &= \frac{\frac{1}{3}\sum_{i=1}^{3} y_i^L \sum_{i=1}^{3} x_i^L - \sum_{i=1}^{3} y_i^L x_i^L}{\frac{1}{3}(\sum_{i=1}^{3} x_i^L)^2 - \sum_{i=1}^{3}(x_i^L)^2}, \\ q_L &= \frac{1}{3}\sum_{i=1}^{3} y_i^L - m\frac{1}{3}\sum_{i=1}^{3} x_i^L. \end{aligned} \tag{3.6}$$

Figura 3.4: Ipose

Algoritmo PoseRecognition_LS

input: quattro array:

- $\mathbf{x}_R$ e $\mathbf{y}_R$ array contenenti le coordinate (x,y) dei tre giunti corrispondenti al braccio destro
- $\mathbf{x}_L$ e $\mathbf{y}_L$ array contenenti le coordinate (x,y) dei tre giunti corrispondenti al braccio sinistro

output: la posizione riconosciuta, altrimenti NotPose

$m_R \leftarrow linearFit(x_R, y_R)$
$m_L \leftarrow linearFit(x_L, y_L)$
$\alpha_R \leftarrow angleDeg(m_R)$
$\alpha_L \leftarrow angleDeg(m_L)$

if $\mid \alpha_R - \alpha_L \mid < 20$ **then**
  return Tpose;
end
if $\mid \alpha_R - \alpha_L \mid < 180$ ***and*** $\mid \alpha_R - \alpha_L \mid > 140$ **then**
  return Ipose;
end
if $\mid \alpha_R - \alpha_L \mid > 60$ ***and*** $\mid \alpha_R - \alpha_L \mid < 90$ ***and*** $\mid \alpha_R \mid < 10$ **then**
  return Rpose;
end
if $\mid \alpha_R - \alpha_L \mid > 60$ ***and*** $\mid \alpha_R - \alpha_L \mid < 90$ ***and*** $\mid \alpha_L \mid < 10$ **then**
  return Lpose;
else
  return NotPose;
end

Algorithm 3.1: Pseudocodice pose recognition

Una volta determinati i coefficienti angolari possiamo perciò comprendere la posizione delle braccia l'una rispetto all'altra, studiando l'angolo in gradi compreso tra le due rette appena trovate.
Detti α_R e α_L gli angoli in gradi ricavati dai coefficienti angolari m_R ed m_L allora:

- se $\mid \alpha_R - \alpha_L \mid < 20 \Rightarrow$ Tpose (Figura 3.5),
- se $140 < \mid \alpha_R - \alpha_L \mid < 180 \Rightarrow$ Ipose (Figura 3.4 a pagina precedente),

- se $60 <| \alpha_R - \alpha_L |< 90$ e $| \alpha_R |< 10 \Rightarrow$ Rpose (Figura 3.7),
- se $60 <| \alpha_R - \alpha_L |< 90$ e $| \alpha_L |< 10 \Rightarrow$ Lpose (Figura 3.6).

Come si può notare può essere riconosciuta anche la posizione *Ipose* che però non è utile ai fini dell'applicazione in esame. I metodi implementati per eseguire il riconoscimento delle pose sono riportati nel codice di header 3.3.

```
#ifndef POSERECOGNITIONLS_H_
#define POSERECOGNITIONLS_H_
#include <vector>
#include <string>

namespace PoseRecognition{

class poseRecognition_LS {
public:
  poseRecognition_LS();
  void linearFit(std::vector<float>&,std::vector<float>&);
  std::string checkPos(std::vector<float>&,
    std::vector<float>&,std::vector<float>&,std::vector<float>&);
  bool isAvailable(std::vector<float>&,std::vector<float>&);
  float angleDeg(float);
  virtual ~poseRecognition_LS();
private:
  std::vector<float> m;
  std::vector<float> q;
  float getMR();
  float getML();
};
};
#endif
```

Listing 3.3: poseRecognition_LS.h (C++)

Legenda:

: retta 'left'

: retta 'right'

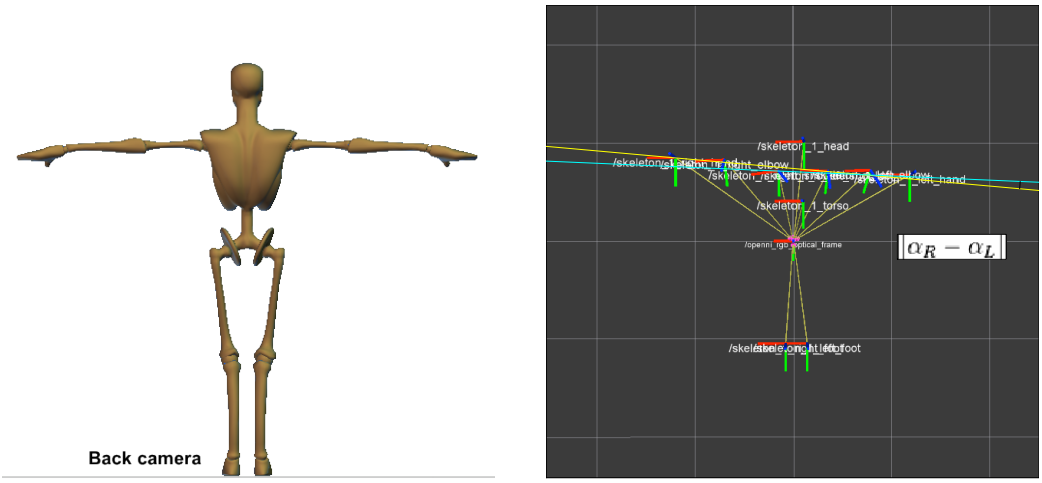


Figura 3.5: Tpose

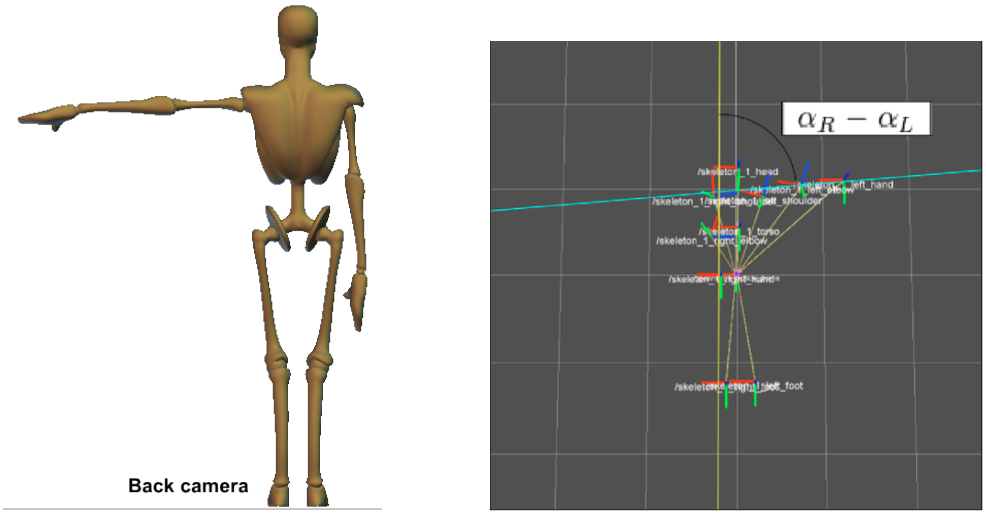


Figura 3.6: Lpose

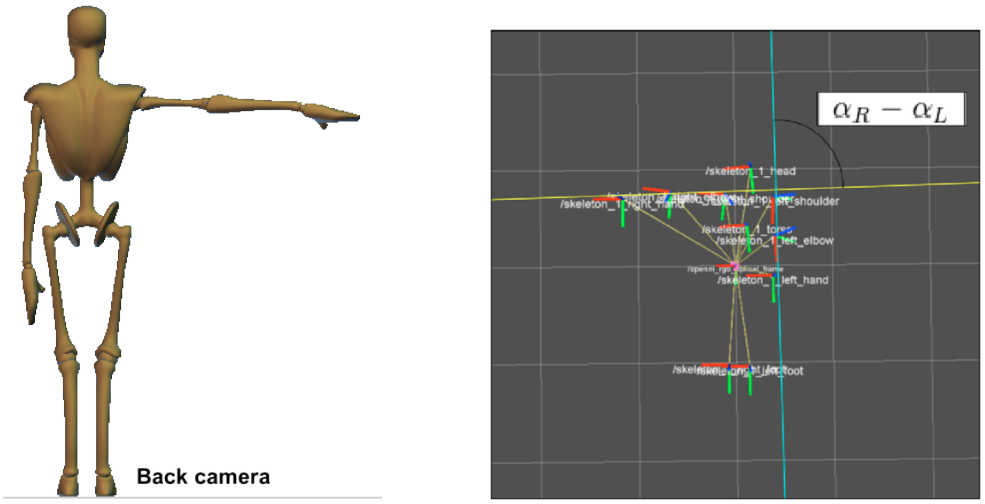


Figura 3.7: Rpose

Da notare che i frame *_right_* corrispondono al braccio sinistro (retta 'left') mentre i frame *_left_* al braccio destro (retta 'right').
Determinata la posizione dell'utente questa trasmetterà una quantità al robot:

- Tpose: 3 manches da tre giocate l'una,
- Rpose: 2 manches da tre giocate l'una,
- Lpose: 1 manches da tre giocate l'una.

3.2 Hand Gesture Recognition

Conclusa l'implementazione del Pose Recognition di cui si è discusso nel precedente paragrafo, si è passati al riconoscimento gestuale effettuato con l'utilizzo delle librerie di Hand Gesture Recognition fornite dal *Multimedia Technology and Telecommunications Lab* di Padova.
I gesti riconosciuti per l'implementazione successiva del gioco della morra cinese sono di conseguenza tre: carta, sasso e forbice. (Figura 3.8)

3.2.1 Protocollo TCP-IP

L'implementazione del Pose Recognition e dell'algoritmo rock-paper-scissors, nonchè il controllo del robot LEGO Mindstorms NXT sono stati sviluppati in ambiente Linux perciò, essendo le librerie per l'Hand Gesture Recognition sviluppate in Microsoft Visual Studio 2010[8], è stato necessario prima di

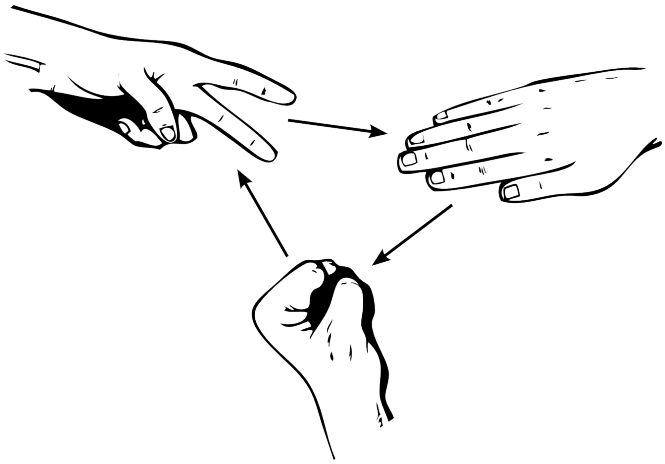


Figura 3.8: Carta, sasso e forbice

tutto sviluppare un protocollo di comunicazione TCP-IP tra due computer distinti.
Una volta posizionato l'utente ad 1-1.5m di distanza dal sensore Kinect connesso al pc Linux, si trasmette una *depth image* al pc Windows.
Con depth image si indica un'immagine contenente informazioni relative alla distanza delle superfici dell'oggetto rispetto un certo punto di vista. Nel caso in esame è stato quindi necessario estrapolare dal sensore Kinect la depthMap, cioè una matrice contenente la distanza di ogni punto dal sistema di riferimento coincidente con il sensore Kinect .
Tale matrice, con numero di righe pari a 480 e numero di colonne pari a 640, si presenta come un vettore *std::vector<uint8_t> data* pubblicato nel topic *camera/depth/image_raw* tramite messaggi di tipo *sensor_msgs/Image.msg* (vedi fugura 3.9).
I valori sono codificati nel formato 16UC1, perciò ogni valore della matrice è un *unsigned short* di lunghezza 2 Byte cioè 16 bit.
L'intero array *data* trasmesso dal server ha quindi dimensione pari a $ROWS \times COLS \times sizeof(short) = 480 \times 640 \times 2Byte = 614400$ Bytes. Al lato client i byte ricevuti vengono letti a coppie di due per formare così l'array *unsigned short depth_image* di lunghezza 480×640.
Infine tale array viene trasformato in un vettore *std::vector<unsigned short> depthMap* da fornire come parametro ai metodi di analisi dei frame forniti dalla libreria per l'Hand Gesture Recognition citata nella prefazione a questo paragrafo (3.2).

Di seguito viene riportata la depth map (3.7) del frame i-esimo dove:

- m = 480 (rows),
- n = 640 (columns).

$$DM_i = \begin{bmatrix} p_{1,1} & p_{1,2} & \cdots & p_{1,n} \\ p_{2,1} & p_{2,2} & \cdots & p_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ p_{m,1} & p_{m,2} & \cdots & p_{m,n} \end{bmatrix} \tag{3.7}$$

che in C++ diventa:

$$\begin{aligned} \textit{unsigned short}^* \ DM = \textit{new unsigned short}[ARR\_LEN]; \\ DM = \{p_{1,1}, p_{1,2}, \cdots, p_{1,n}, \cdots, p_{m,1}, p_{m,2}, \cdots, p_{m,n}\}; \end{aligned} \tag{3.8}$$

con ARR_LEN = 480 × 640 poi trasformato in *std::vector<unsigned short> depthMap*.
Vengono riportate alcune righe di codice del protocollo Tcp-IP nel codice 3.4.

```
# This message contains an uncompressed image
# (0, 0) is at top-left corner of image
#

Header header        # Header timestamp should be acquisition time of image
                     # Header frame_id should be optical frame of camera
                     # origin of frame should be optical center of cameara
                     # +x should point to the right in the image
                     # +y should point down in the image
                     # +z should point into to plane of the image
                     # If the frame_id here and the frame_id of the CameraInfo
                     # message associated with the image conflict
                     # the behavior is undefined

uint32 height         # image height, that is, number of rows
uint32 width          # image width, that is, number of columns

# The legal values for encoding are in file src/image_encodings.cpp
# If you want to standardize a new string format, join
# ros-users@lists.sourceforge.net and send an email proposing a new encoding.

string encoding       # Encoding of pixels -- channel meaning, ordering, size
                      # taken from the list of strings in include/sensor_msgs/image_encodings.h

uint8 is_bigendian    # is this data bigendian?
uint32 step           # Full row length in bytes
uint8[] data          # actual matrix data, size is (step * rows)
```

Figura 3.9: sensor_msgs/Image.msg

```
 /*
  *server.cpp
  *Invio della depth map
  */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
...
send(client_sockfd,depthImage,bytes,0);
...

 /*
  *client.cpp ()
  *Ricezione della depth map
  */
#include <winsock.h>
...
depth_image = new unsigned short[ARR_LEN];
rc = recv(clientsocket,(char*)depth_image,bytes,MSG_WAITALL);

for(int i=0; i < ARR_LEN; i++)
  depthMap[i] = depth_image[i];
//Determinazione del gesto
...
```

Listing 3.4: Client-Server

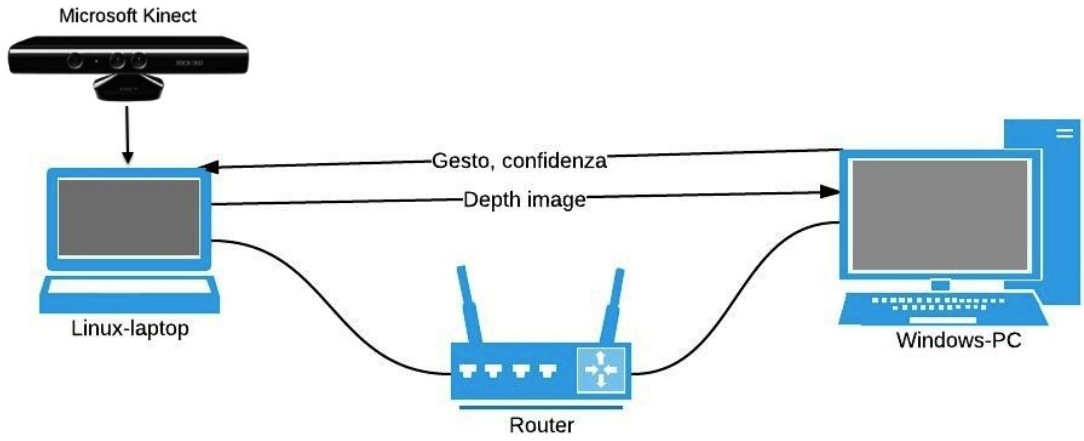
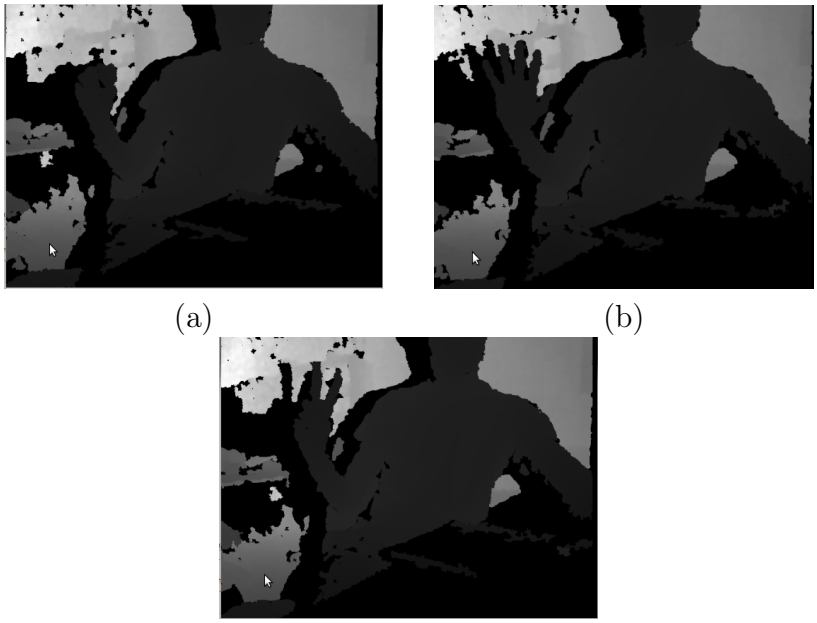


Figura 3.10: Network

A questo punto, con l'utilizzo delle apposite librerie, viene assegnata una probabilità ad ognuno dei tre gesti indicante quale gesto sia il più plausibile. I tre gesti con le relative confidenze vengono quindi trasmessi al pc Linux. (Figura 3.10)
Nella figura sottostante (3.11) vengono riportate tre depth map di esempio relative ad ognuno dei tre gesti.



(a) (b)

(c)

Figura 3.11: Depth images: rock (a), paper (b), scissors (c)

3.2.2 Determinazione del gesto

Tramite un messaggio di tipo *Gesture.msg* (3.5), pubblicato nel topic */Gesture*, si ottiene il gesto più probabile per la depth map inviata.
Per la determinazione del gesto si attendono i messaggi relativi a 3 depth map consecutive, identificate dal campo *seq* dell'header presente nel messaggio *Gesture.msg*.

Passi per la determinazione del gesto:

1. **Attesa di tre messaggi**
2. **Scelta del gesto**

1) **Attesa di tre messaggi**
Per prima cosa definisco l'array *double c[]* = $\{0, 0, 0\}$ nel quale l'indice i indica uno dei tre gesti e ogni componente c[i] contiene la somma delle probabilità per il gesto i .

i	Gesto
0	r
1	p
2	s

Tabella 3.2: Corrispondenze indice i e gesto

Supponendo che sia stato letto l'$n - esimo$ messaggio:

- - *seq* = n
  - *gesture* = r $\Rightarrow$ 0 = i
  - *confidence* = 0.6

procedo all'aggiornamento della componente c[i]:

$$c[i] = c[i] + confidence, \tag{3.9}$$

che nel caso specifico diventa:

$$c[0] = c[0] + 0.6 = 0 + 0.6 = 0.6. \tag{3.10}$$

Per la determinazione del gesto è necessario attendere i messaggi $(n + 1) - esimo$ e $(n + 2) - esimo$ che suppongo essere:

- - *seq* = n+1
  - *gesture* = p $\Rightarrow$ 1 = i
  - *confidence* = 0.5

  Aggiorno l'elemento c[1] :
  c[1] = c[1] + 0.5 = 0 + 0.5 = 0.5.

- - *seq* = n+2
  - *gesture* = r $\Rightarrow$ 0 = i

- *confidence* = 0.6

Aggiorno l'elemento c[0] :
c[0] = c[0] + 0.6 = 0.6 + 0.6 = 1.2.

2) **Scelta del gesto**

Un volta ottenuto l'array finale *c[]* viene determinato l'elemento maggiore *max*. Di tale elemento viene memorizzato l'indice *i_max*, corrispondente al gesto giocato dall'utente.

In questo caso $c[] = \{1.2, 0.5, 0\}$ perciò:
- $max = 1.2$,
- $i\_max = 0$,

poichè all'indice 0 corrisponde il simbolo r il gesto giocato dall'utente è r cioè rock.

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id

char gesture
float64 confidence
```

Listing 3.5: Messaggio Gesture.msg di ROS contenete la coppia (gesto,confidenza)

Capitolo 4

Rock-Paper-Scissors

4.1 Controllo dei motori del robot LEGO Mindstorms NXT

Per permettere al robot di emulare i gesti carta, sasso e forbice (Figura 4.2), si sono dovuti controllare singolarmente due motori (vedi 4.1).
Perciò dopo aver inizializzato i nodi *r_ wheel_ controller* e *nxt_ central_ controller*, pubblicando un angolo in radianti rispettivamente nei topic *angle_ r* ed *angle_ c*, è possibile portare i motori, in questo caso denominati r_wheel_joint e central_joint (vedi 4.2), ad una data angolazione.
Per comunicare al topic il dato angolo vengono utilizzati messaggi del tipo *std_ msgs::Float64*.

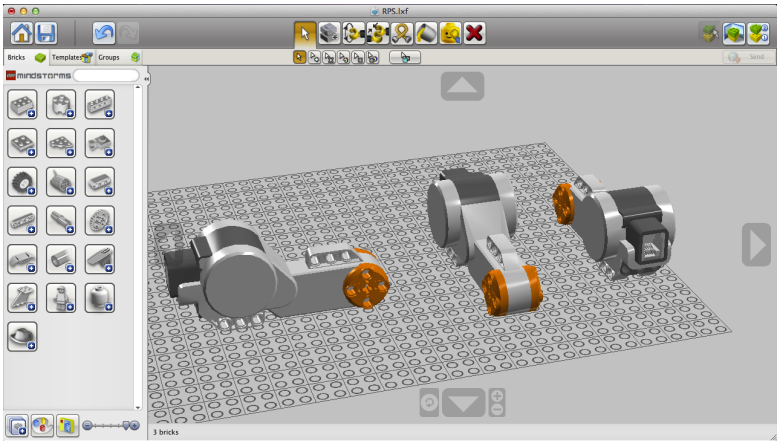


Figura 4.1: Motore LEGO Mindstorms NXT (Screenshot LEGO Digital Designer [9])

Il controllo fisico dei due motori viene effettuato tramite gli script in Python *central_joint_controller.py* e *r_wheel_joint_controller.py*.
Il codice riportato in 4.1 è adibito al controllo dell'angolazione dei motori.

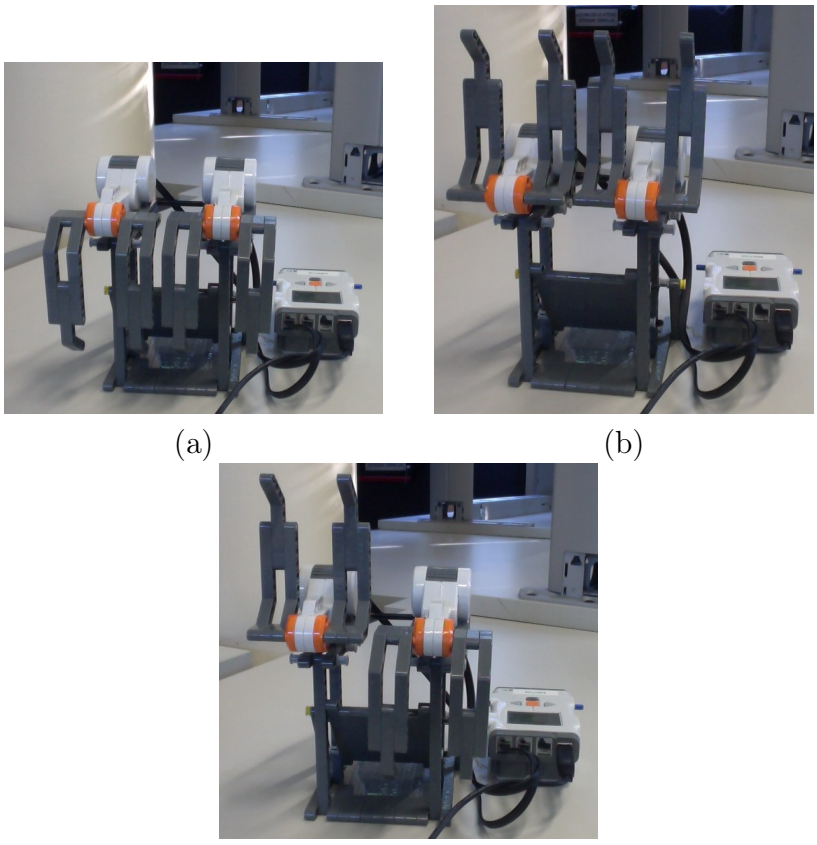
```
...
        cmd = JointCommand()
        cmd.name = self.joint
        cmd.effort = 0.0
        if self.angle_desi - angles[self.joint] > 0.2:
            cmd.effort = max(0.55, 0.55 +
              (self.angle_desi - angles[self.joint]) / 32.0)
        elif self.angle_desi - angles[self.joint] < -0.2:
            cmd.effort = min(-0.55, -0.55 +
              (self.angle_desi - angles[self.joint]) / 32.0)
        print cmd.effort
        self.pub.publish(cmd)
...
```

Listing 4.1: Codice adibito al controllo dell'angolazione del motore (central_joint_controller.py)

```
...
  - type: motor
    name: central_joint
    port: PORT_A
    desired_frequency: 20.0

  - type: motor
    name: r_wheel_joint
    port: PORT_B
    desired_frequency: 20.0
...
```

Listing 4.2: File di configurazione nxt_lab.yaml per il robot LEGO Mindstorms NXT



(a) (b)

(c)

Figura 4.2: Gesti robot: rock (a), paper (b), scissors (c)

Nota: per l'utilizzo del robot NXT è necessario lanciare i nodi tramite il comando da terminale *roslaunch rps nxt_ gesture.launch*. Per il corretto funzionamento è necessario che il robot assuma la posizione 4.3 prima dell'esecuzione del comando.

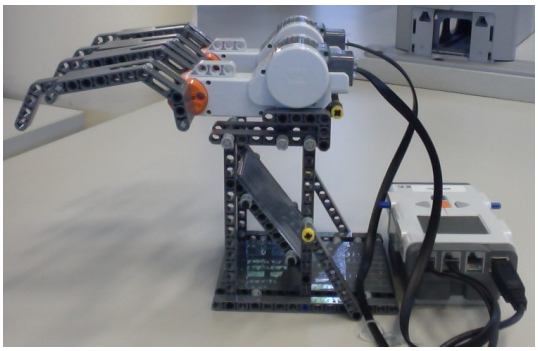


Figura 4.3: Posizione iniziale LEGO Mindstorms NXT

4.2 Intelligenza Artificiale: Gaussian Mixture Model

Per l'implementazione dell'intelligenza artificiale (IA) si è utilizzato un approccio basato su Gaussian Mixture Models (GMM). Una GMM è una pdf (probability density function) parametrica rappresentata come una somma pesata di K componenti gaussiane.
I parametri della GMM vengono stimati dai dati di training tramite l'algoritmo EM (Expectation-Maximization Algorithm).

4.2.1 Gaussian Mixture Model (GMM)

Riprendendo l'introduzione al capitolo, una GMM è una somma pesata di K densità gaussiane data dall'equazione:

$$p(\boldsymbol{x} \mid \theta) = \sum_{i=1}^{K} \omega_i g(\boldsymbol{x} \mid \boldsymbol{\mu}_i, \Sigma_i), \tag{4.1}$$

dove $\boldsymbol{x}$ è un vettore $\boldsymbol{x} = \{x_1, \ldots, x_d\}$ t.c. $\mid \boldsymbol{x} \mid = d$ che corrisponde a un punto. Gli ω_i sono i pesi relativi alle varie componenti $g(\boldsymbol{x} \mid \boldsymbol{\mu}_i, \Sigma_i)$ con i = 1, ..., K.

Ogni componente è un funzione di densità di probabilità d-variata nella forma:

$$g(\boldsymbol{x} \mid \boldsymbol{\mu}_i, \Sigma_i) = \frac{1}{(2\pi)^{d/2} \mid \Sigma_i \mid^{1/2}} exp\Big\{ -\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu}_i)' \Sigma_i^{-1} (\boldsymbol{x} - \boldsymbol{\mu}_i) \Big\} \tag{4.2}$$

con vettore delle aspettazioni $\boldsymbol{\mu}_i$ e matrice di covarianza Σ_i. I pesi ω_i soddisfano la proprietà di normalizzazione $\sum_{i=1}^{K} \omega_i = 1$.

La GMM è perciò parametrizzata secondo i vettori delle aspettazioni, le matrici di covarianza e i pesi di tutte le Kcomponenti; tali parametri sono rappresentati collettivamente nel seguente modo:

$$\theta = \{\omega_i, \boldsymbol{\mu}_i, \Sigma_i\}, i = 1, \ldots, K. \tag{4.3}$$

Per ulteriori informazioni consultare [15], [18], [19].

4.2.2 Stima parametrica e EM Algorithm

Di fondamentale importanza è quindi la fase di training e quindi di stima parametrica secondo il criterio di Maximum Likelihood (ML).
La stima viene effettuata tramite l'agoritmo di Expectation-Maximization (EM) che ha come scopo principale quello di trovare i parametri del modello che massimizzano la verosimiglianza della GMM dato il dataset di training.

Per ulteriori informazioni consultare [15] e [18].

4.2.3 BIC: Bayesian Information Criterion

Come si evince dalla sezione 4.2.1 di fondamentale importanza è la scelta del numero K di componenti della GMM.
Per stimare il valore ottimale di K si è ricorso al *Bayesian Information Criterion* (Criterio di informazione Bayesiano). Il *BIC* è un criterio per la selezione di un modello tra una classe di modelli parametrici con un diverso numero di parametri.
Stimando i parametri del modello mediante il metodo della massima verosimiglianza (maximum likelihood) è possibile aumentare la verosimiglianza attraverso l'aggiunta di parametri, la qual cosa può provocare over-fitting, cioè portare ad una riduzione della WRSS (Weighted Residual Sum of Squares) e ad un miglioramento del pattern di residui ma, nel contempo, a stime parametriche molto sensibili dai dati.
Tramite il *BIC* si determina un modello con il minor numero di parametri che abbia un buon fit rispetto ai dati.
Per ulteriori informazioni consultare [11].

4.3 Intelligenza Artificiale: Rock-Paper-Scissors & GMM

4.3.1 Training

Nel caso in esame la fase di training, e quindi di stima parametrica, viene effettuata tramite un dataset $X = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$ con $N = 850$ punti in cui ogni punto $\boldsymbol{x}_j$ con j = 1, ..., N è di dimensione $d = 8$ ed è così definito:

$$\boldsymbol{x}_j = \{x_1, x_2, x_3, x_4, x_5, x_6, x_8\} \tag{4.4}$$

con $x_i \in \{0, 1, 2\}$ dove rock = 0, paper = 1, scissors = 2 e i = 1, ..., 8.

Gli otto simboli indicano 4 giocate, i simboli contrassegnati da indice dispari indicano i gesti giocati dal robot (R), quelli contrassegnati da indice pari i gesti giocati dall'utente umano (H).
I primi sei simboli sono le tre giocate precedenti.
Il settimo e l'ottavo simbolo rappresentano l'ultima giocata data la storia di tre precedenti. In particolare il settimo è il gesto che il robot gioca per cercare di vincere (*R to win*) su *H last throw*, l'ultimo gesto giocato dall'utente (vedi 4.1).
Il dataset utilizzato è stato memorizzato per usi futuri in *dataRPS.txt*

	R	H	R	H	R	H	R to Win	H last throw
$\boldsymbol{x}_j =$	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8

Tabella 4.1: $\boldsymbol{x}_j$, R = Robot, H = Human (le tre giocate sono suddivise per colore)

4.3.2 Determinazione del gesto del robot

Dopo la fase di training e l'esecuzione dell'algoritmo *BIC* per la determinazione del numero di stati ottimale, si è potuti procedere con la scelta vera e propria del gesto che il robot dovrà giocare per vincere.

Partendo da una storia di tre giocate consecutive, contenuta in *lastRow.txt*, si completa la sequenza con ognuna delle tre coppie (gesto_R, gesto_H) 'favorevoli' cioè che portano il robot alla vittoria.
Le tre configurazioni favorevoli sono:

- (0,2),
- (1,0),
- (2,1),

dove il primo gesto è *R to Win* mentre il secondo è *H last throw*.
Si ottengono perciò le 3 probabilità *pr, pp e ps* corrispondenti alla possibilità di avere la sequenza di sei simboli seguita rispettivamente da una delle coppie sopra elencate.
Una volta ottenute le tre probabilità si determina casualmente il gesto che il robot dovrà giocare e si procede quindi col confronto per determinare il vincitore.

A questo punto, avendo a disposizione la coppia (robot_gesture, human_gesture), si procede all'aggiornamento del dataset *dataRPS.txt* aggiungendo una nuova riga, tale dataset verrà poi utilizzato per il training del'iterazione successiva permettendo così l'aggiornamento della GMM.
Allo stesso modo viene sovrascritto *lastRow.txt* con una riga contenente la storia delle ultime 3 giocate.
Mentre si adotta un aggiornamento della GMM ad ogni iterazione, per quanto riguarda il calcolo del numero ottimale di stati tramite il metodo *BIC* si opta per un aggiornamento ogni 2^n iterazioni con $n \in \mathbb{N}$ descritto dall'algoritmo seguente 4.1.

Algoritmo testBIC()

k_iter = 0, identificatore iterazione alla quale stimare K
num_iter = 0, identificatore iterazione corrente

if $k\_iter = num\_iter$ ***and*** $k\_iter = 0$ **then**
    k_iter $\leftarrow$ 1;
    $K \leftarrow$ BIC();
end
if $k\_iter = num\_iter$ **then**
    k_iter $\leftarrow$ k_iter $\times$ 2;
    $K \leftarrow$ BIC();
end

Algorithm 4.1: Pseudocodice testBIC() implementato in RPSclass.cpp

num_iter	**k_iter**
0	0
1	$1 = 2^0$
2	2^1
3	2^2
4	2^2
5	2^3
6	2^3
7	2^3
8	2^3
9	2^4
⋮	⋮

Tabella 4.2: Modifica di *k_iter* e *num_iter*, in blu quando viene eseguito il *BIC*

Di seguito viene descritto in dettaglio il procedimento, diviso in quattro punti, di determinazione del gesto e di aggiornamento dei dataset, supponendo di torvarsi all'iterazione $1000 + (i + 1) - esima$.

Passi:

1. **Caricamento dataset**
2. **Aggiornamento delle probabilità**
3. **Determinazione del gesto che sarà giocato dal robot**
4. **Aggiornamento dataset**

1) **Caricamento dataset**

	R	H	R	H	R	H	R to Win	H last throw
$\boldsymbol{x}_1 =$	1	2	1	0	1	2	1	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
$\boldsymbol{x}_{1000} =$	2	1	1	1	0	2	1	2
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
$\boldsymbol{x}_{1000+i} =$	1	2	0	2	1	2	2	1

Tabella 4.3: *dataRPS.txt*

R	H	R	H	R	H
0	2	1	2	2	1

Tabella 4.4: *lastRow.txt*

Come si può notare la tabella 4.4 contiene la storia delle 3 precedenti giocate che saranno necessarie per determinare le nuove *pr, pp e ps* e quindi il prossimo gesto del robot.
Confrontando 4.3 e 4.4 si può notare come la riga di *lastRow.txt* corrisponda agli ultimi 6 simboli di $\boldsymbol{x}_{1000+i}$ in *dataRPS.txt*.
In rosso viene indicata l'ultima giocata.

2) **Aggiornamento delle probabilità**

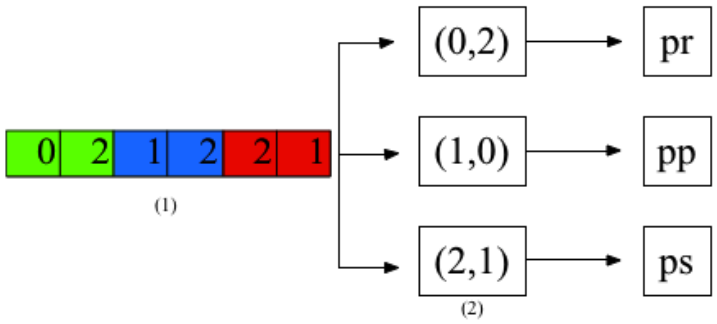


Figura 4.4: Aggiornamento di pr, pp e ps

Data la sequenza delle tre giocate precedenti (Figura 4.4 (1)) si procede col completamento tramite ognuna delle tre sequenze favorevoli (Figura 4.4 (2)). Ottenute le tre sequenze complete denominate, in relazione al primo simbolo di completamento, $\boldsymbol{x}_0$, $\boldsymbol{x}_1$ e $\boldsymbol{x}_2$ si ottengono le tre probabilità generali delle sequenze rispetto alle K componenti della GMM:

$$\begin{aligned} pr = P(\boldsymbol{x}_0) &= \sum_{m=1}^{K} P(C = m \mid \boldsymbol{x}_0), \\ pp = P(\boldsymbol{x}_1) &= \sum_{m=1}^{K} P(C = m \mid \boldsymbol{x}_1), \\ ps = P(\boldsymbol{x}_2) &= \sum_{m=1}^{K} P(C = m \mid \boldsymbol{x}_2), \end{aligned} \tag{4.5}$$

dove C indica la componente.
Il metodo ***double GaussianMixture::p(const Point& point)***, utilizzato per ottenere le probabilità di cui si è appena discusso, in realtà ritorna il valore della pdf (probability density function) calcolato nel punto *point*. Perciò, per ottenere *pr*, *pp* e *ps*, è stato necessario normalizzare i valori come indicato nel codice seguente (4.3).

```
12 ...
13  //Normalization
14  double norm = 1.0/(p_0 + p_1 + p_2);
15
16  pr = p_0*norm;
17  pp = p_1*norm;
18  ps = p_2*norm;
19 ...
```

Listing 4.3: RPSclass.cpp

In riferimento a 4.3 con *p_0*, *p_1* e *p_2* vengono indicati i valori della pdf e con *pr*, *pp* e *ps* le probabilità.

3) **Determinazione del gesto che sarà giocato dal robot**

Una volta ottenute le nuove *pr, pp, e ps* tramite il metodo **void RPSclass::getProbability()** implementato in *RPSclass.cpp*, si può procedere con la scelta del gesto che il robot dovrà giocare in base alla seguente procedura:

- Generazione di un numero casuale *random* nell'intervallo [0,1] tramite il metodo **double RPSclass::rndNumber()** implementato sempre in *RPSclass.cpp*
- Suddivisione dell'intervallo [0,1] in tre intervalli disgiunti e determinazione del gesto :
  - se *random* $\in$ [0,pr) $\Rightarrow$ Rock
  - se *random* $\in$ [pr,pr+pp) $\Rightarrow$ Paper
  - se *random* $\in$ [pr+pp,1] $\Rightarrow$ Scissors

Di seguito viene riportato lo pseudocodice relativo al metodo **char RPSclass::nxtGesture(<parameters>)**.

Algoritmo nxtGesture()

output: gesto g

getProbability();

rnd $\leftarrow$ rndNumber();
if $rnd \geq 0$ ***and*** $rnd < pr$ **then**
  return Rock;
end
if $rnd \geq pr$ ***and*** $rnd < (pp + pr)$ **then**
  return Paper;
end
return Scissors;

Algorithm 4.2: Pseudocodice nxtGesture()

4) **Aggiornamento dataset**

Aggiornamento dei dataset, **g_r** e **g_h** indicano rispettivamente il gesto giocato dal robot e quello giocato dall'utente nell'ultima giocata.

	R	H	R	H	R	H	R to Win	H last throw
$\boldsymbol{x}_1 =$	1	2	1	0	1	2	1	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
$\boldsymbol{x}_{1000} =$	2	1	1	1	0	2	1	2
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
$\boldsymbol{x}_{1000+i} =$	1	2	0	2	1	2	2	1
$\boldsymbol{x}_{1000+(i+1)} =$	0	2	1	2	2	1	**g_r**	**g_h**

Tabella 4.5: *dataRPS.txt*

R	H	R	H	R	H
1	2	2	1	**g_r**	**g_h**

Tabella 4.6: *lastRow.txt*

4.4 Test

Nella fase di test si è utilizzata una sequenza di 200 simboli (dataset di test) non generata casualmente ma realmente giocata da utenti umani.
Per verificare l'efficienza dell'IA (Intelligenza Artificiale) si è dapprima utilizzato un algoritmo che scegliesse il gesto giocato dal robot casualmente, cioè basandosi unicamente sulla probabilità di avere uno dei tre gesti:

1. pr = $\frac{1}{3}$ per sasso,
2. pp = $\frac{1}{3}$ per carta,
3. ps = $\frac{1}{3}$ per forbice.

Una volta determinata la percentuale di vittoria del robot in questo contesto si è testato il medesimo dataset con l'algoritmo utilizzante la GMM e si è determinata la percentuale di vittoria del robot.
Il dataset contenente la sequenza di 200 gesti è *human_ gesture.txt*.

4.4.1 Risultati primo test

Come descritto nell'introduzione del paragrafo, il test viene effettuato con pr = pp = ps = $\frac{1}{3}$ costanti.
In questa tipologia di test non è necessaria la fase di training perchè ogni giocata è considerata singolarmente e la scelta del gesto giocato dal robot viene operata casualmente.
Per la determinazione della percentuale di vittoria nelle 200 giocate del dataset di test si è iterato il procedimento per 500 volte ottenendo, in media, le seguenti percentuali:

$$p\_win = \mathbf{33.258\%}$$
$$p\_l = 33.371\%$$
$$p\_t = 33.371\%$$

Dove p_win è la percentuale di vittoria del robot mentre p_l e p_t sono rispettivamente la percentuale di giocate perse (lose) e pareggiate (tie).

Per permettere il confronto con i risultati successivi si è determinata la p_win anche con un numero di iterazioni pari a 20 ottenendo una percentuale di vittoria pari al 32.55%.

4.4.2 Risultati secondo test

In questo caso pr, pp e ps sono determinate grazie all'utilizzo della GMM e dipendono dal dataset di training utilizzato.
Per questo si sono eseguiti più test sempre su 200 giocate ma su dataset di training di dimensione diversa: da 150 a 650 punti.
In ogni variante per determinare la percentuale di vittoria si è iterato il procedimento per 20 volte.

Dati:

- 200 gesti di utenti umani,
- MAXK = 50, K massimo,
- 20 iterazioni.

I risultati sono riportati nella tabella 4.7 sottostante.

Dimensione dataset di training (dataRPS.txt)	**Perc. vittoria**	**Perc. partite perse**	**Perc. pareggio**
150	**35.225%**	30.3%	34.475%
250	**35.45%**	29.275%	35.275%
350	**33.2%**	29.125%	37.675%
450	**35.4%**	31.275%	33.325%
550	**35.825%**	30.825%	33.35%
650	**36.65%**	31.15%	32.2%

Tabella 4.7: Risultati secondo test

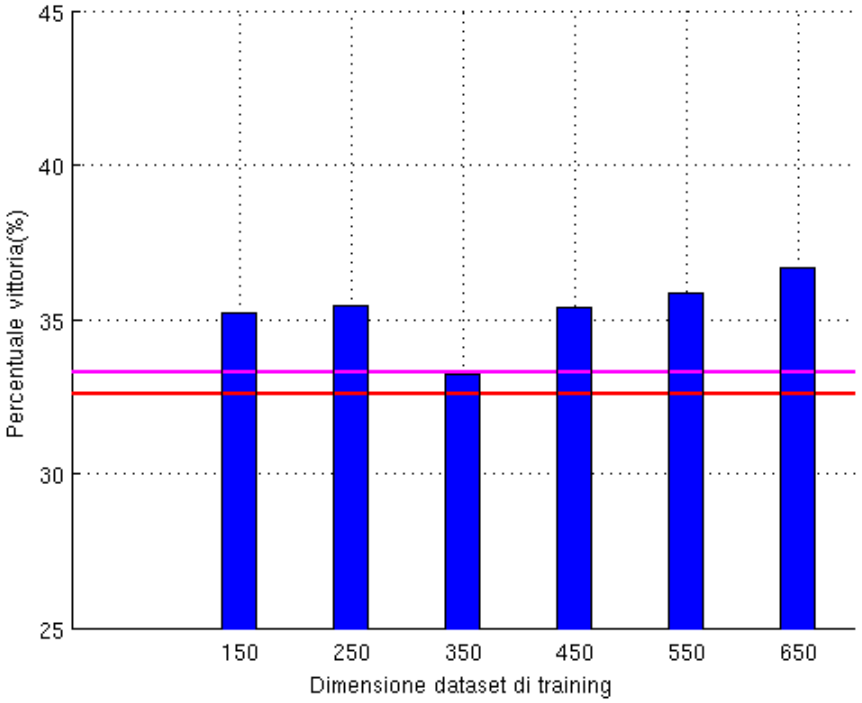


Figura 4.5: *Grafico risultati secondo test.* Le due rette indicano le percentuali di vittoria relative al primo test: 33.258% (magenta), basato su 500 iterazioni, e 32.55% (rosso), basato su 20 iterazioni. In blu sono indicate le percentuali di vittoria per ogni dimensione del dataset di training considerata.

4.4.3 Considerazioni

Come si può notare dal grafico 4.5 la percentuale di vittoria assume valori vicini al 35% già con un dataset di training di 150 punti.
Per 250 punti otteniamo la percentuale di vittoria del 35.45% mentre per il dataset successivo, di dimensione 350, la p_win è pari a 33.2%. Questo calo della percentuale di vittoria può essere spiegato con l'alternarsi di giocatori diversi. Il dataset di 250 punti contiene infatti 150 giocate di uno stesso utente; perciò è presumibile che sia stato più semplice individuare pattern ripetuti e portare il robot alla vittoria.
I cento simboli aggiuntivi presenti nel dataset di 350 campioni sono invece di giocatori differenti, questo ha portato alla diminuzione della percentuale di vittoria che ha ricominciato a crescere, portandosi a 36.65%, quando il modello è riuscito a generalizzare e ha cominciato a gestire un numero più ampio di utenti.

Con l'utilizzo dell'IA si è passati da una percentuale di vittoria per venti iterazioni del 32.55% ad una percentuale del 36.65%. Il miglioramento ottenuto nella percentuale di vittorie è consistente già con 650 giocate, ma prevediamo che il modello possa ulteriormente migliorare le performance con un insieme maggiore di dati di training.
Si nota comunque come la percentuale di vincite del robot sia, per ogni dimensione del dataset, sempre superiore alla percentuale di partite perse portando il robot, nel complesso, alla vittoria.

Unendo il dataset di training di dimensione maggiore di 650 punti e il dataset di test di 200 si sono stimate le percentuali di utilizzo dei tre gesti:

- 29.06% sasso,
- 30.82% carta,
- 40.12% forbice.

Su un campione di 850 giocate quindi la percentuale di utilizzo non è pari ad $\frac{1}{3}$ e fa notare come il simbolo più giocato sia la forbice. Nel grafico 4.6 vengono evidenziate le percentuali di utilizzo.

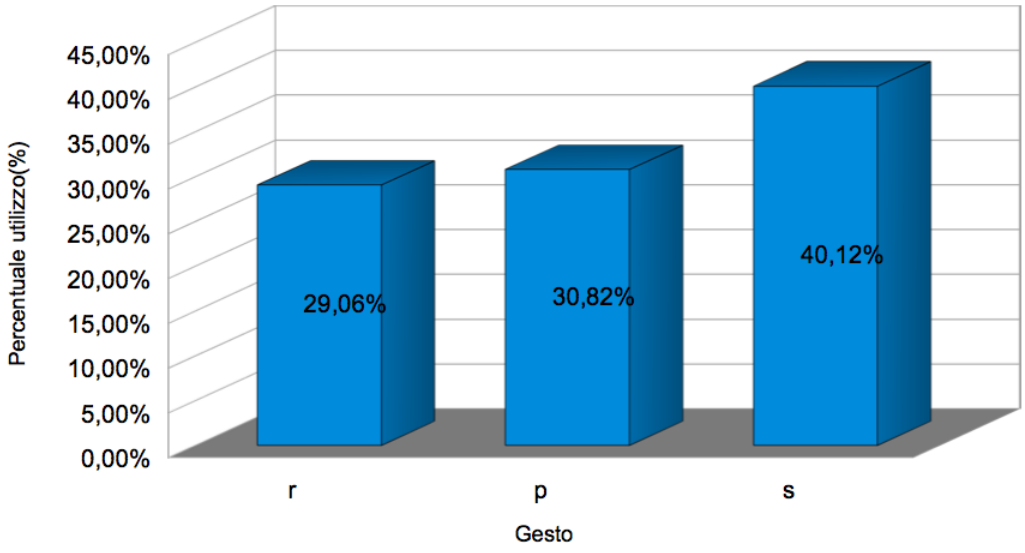


Figura 4.6: Percentuale di utilizzo dei tre gesti

Capitolo 5

Conclusione

Il presente lavoro di tesi ha descritto la realizzazione di un'applicazione per unire hand gesture e pose recognition al controllo del robot LEGO Mindstorms NXT .
Ciò che si è ottenuto è la possibilità di interagire col robot in modo estremamente semplice tramite un set predefinito di pose per comunicare una quantità, nella fattispecie il numero di manches da giocare, e un set predefinito di gesti per sfidarlo a morra cinese.
Come descritto nel paragrafo (4.3) non ci si è limitati ad una mera scelta casuale del gesto del robot, ma si è implementata una IA (Intelligenza Artificiale) basata su GMM (Gaussian Mixture Model).
In questo modo il gesto viene scelto in base ad un pattern di giocate precedenti e, come si è visto nel paragrafo (4.4), ciò ha dato buoni risultati portando il robot a vincere nel 36.65% dei casi, considerando il dataset di training di dimensione massima 650.
Rispetto alla scelta casuale del gesto, dove anche per un numero elevato di iterazioni la percentuale di vittoria si aggira intorno al 33% (al meglio 33.26% per le prove che abbiamo fatto), abbiamo ottenuto un esito positivo e possiamo affermare che il robot opera una scelta intelligente. L'utente infatti non gioca i tre gesti casualmente, cioè con pr = pp = ps = $\frac{1}{3}$, ma le probabilità sono influenzate dai gesti precedenti, dalla conoscenza dell'avversario, dal genere (maschio o femmina) e da altri aspetti e, secondo alcuni studi (vedi [10]) , sono pari a:

- pr = 0.296 sasso,
- pp = 0.354 carta,
- ps = 0.35 forbice.

Ciò porta il robot senza IA a vincere con percentuali più basse di quello allenato con GMM.

Di fondamentale importanza per la stima della percentuale di vittoria è il dataset utilizzato in fase di training. In futuro perciò si potrebbe procedere con una analisi più dettagliata tramite l'utilizzo di dataset di dimensioni superiori.
Si potrebbe migliorare ulteriormente la percentuale di vittoria se il robot utilizzasse un modello casuale per le prime sfide con un soggetto, ma poi generasse un modello diverso per ogni giocatore in modo da adattarsi al modo di giocare del singolo.
Con questi ulteriori studi si potrebbe ottenere una migliore analisi dell'IA stimando con più precisione la percentuale di vittoria.

Appendice A

Teleoperazione del robot NXT

Di seguito viene riportato un secondo utilizzo della libreria di pose recognition di cui si è discusso nel paragrafo 3.1.
Sottoscrivendo il topic */pose* si possono ottenere le quattro posizioni riconosciute:

- Tpose (figura 3.5),
- Ipose (figura 3.4),
- Rpose (figura 3.7),
- Lpose (figura 3.6).

In questo caso lo scopo è quello di teleoperare cioè di guidare a distanza il robot (figura A.1), associando ad ognuno dei quattro gesti una direzione da prendere.

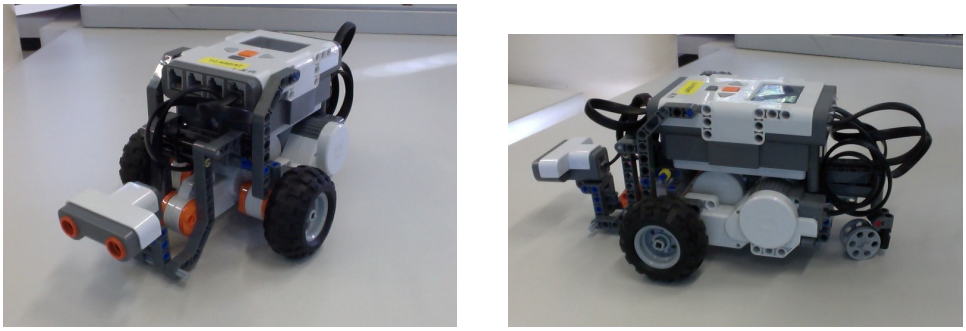


Figura A.1: LEGO Mindstorms NXT

Dopo aver lanciato i nodi del robot tramite il comando *roslaunch nxt_unipd nxt_lab.launch*, una volta ottenuta una determinata posa, si procede con l'invio della velocità al topic *cmd_vel* tramite messaggi di tipo *geometry_msgs/Twist*. La velocità, come risulta dal codice A.1, risulta divisa in due componenti *angolare* e *lineare*.

Corrispondenze posa-direzione (figura A.2):

- se Tpose ⇒ indietro,
- se Ipose ⇒ avanti,
- se Rpose ⇒ destra,
- se Lpose ⇒ sinistra.

```
20 geometry_msgs/Vector3 linear
21
22 geometry_msgs/Vector3 anguar
```

Listing A.1: geometry_msgs/Twist

```
23 ...
24   - type: motor
25     name: r_wheel_joint
26     port: PORT_B
27     desired_frequency: 20.0
28
29   - type: motor
30     name: l_wheel_joint
31     port: PORT_C
32     desired_frequency: 20.0
33 ...
```

Listing A.2: File di configurazione nxt_lab.yaml, vengono indicate le porte per i due motori

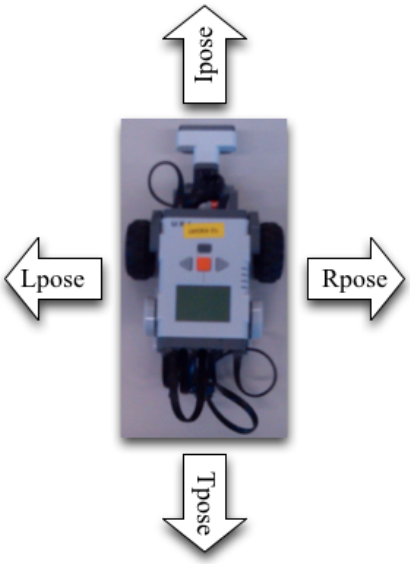


Figura A.2: Movimenti robot LEGO Mindstorms NXT

Bibliografia

[1] http://mindstorms.lego.com/en-us/default.aspx.

[2] http://www.ros.org/wiki/.

[3] http://www.ros.org/wiki/groovy.

[4] http://www.microsoft.com/en-us/kinectforwindows/.

[5] http://www.openni.org.

[6] http://www.openkinect.org.

[7] http://www.ros.org/wiki/tf.

[8] http://www.microsoft.com/visualstudio/eng/2013-preview.

[9] http://ldd.lego.com/en-us/.

[10] http://www.worldrps.com/.

[11] Bayesian information criteria. In *Information Criteria and Statistical Modeling*, Springer Series in Statistics, pages 211–237. Springer New York, 2008.

[12] M.R. Andersen, T. Jensen, P. Lisouski, A.K. Mortensen, M.K. Hansen, T. Gregersen, and P. Ahrendt. Kinect depth sensor evaluation for computer vision applications. *Department of Engineering, Aarhus University. Denmark. - Technical report ECE-TR-6*, page 37, 2012.

[13] K. K. Biswas and S.K. Basu. Gesture recognition using microsoft kinect. In *Automation, Robotics and Applications (ICARA), 2011 5th International Conference on*, pages 100–103, 2011.

[14] S. Calinon, F. Guenter, and A. Billard. On learning, representing, and generalizing a task in a humanoid robot. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 37(2):286–298, 2007.

[15] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

[16] Yi Li. Hand gesture recognition using kinect. In *Software Engineering and Service Science (ICSESS), 2012 IEEE 3rd International Conference on*, pages 196–199, 2012.

[17] D. McColl and G. Nejat. Affect detection from body language during social hri. In *RO-MAN, 2012 IEEE*, pages 1013–1018, 2012.

[18] D.A. Reynolds. Gaussian mixture models. *Encyclopedia of Biometric Recognition*, 2008.

[19] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003.