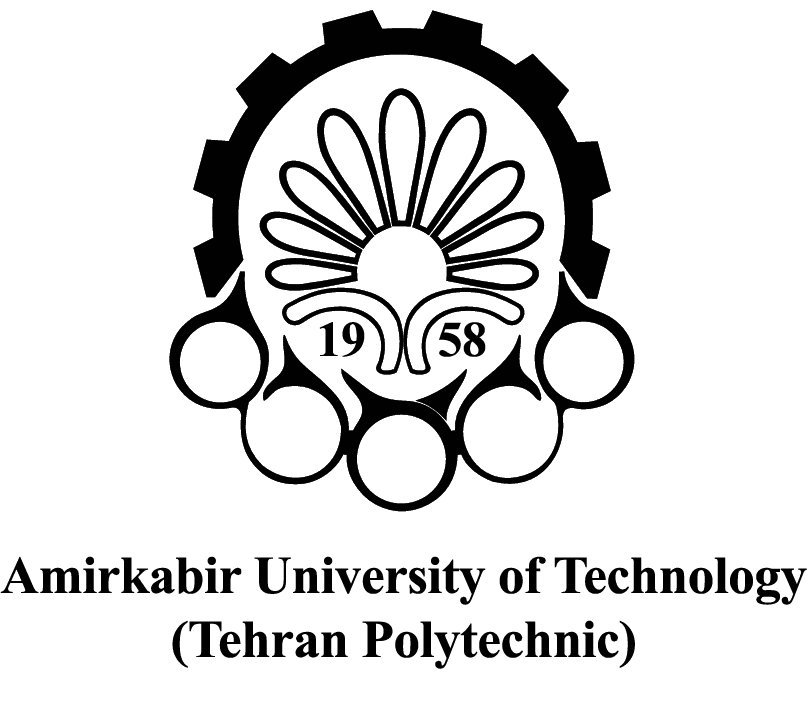
****

a

**کنترل ربات**

سینا ربیعی، علیرضا فقیه علی‌آبادی، معین نصیری

دکتر عبداللهی

پاییز 1401

**فهرست مطالب**

[1. داده‌سازی ۳](#_Toc124920888)

[**1. داده‌های سنسور**  ۳](#_Toc124920889)

[**2. فیلتر کردن و بهینه‌سازی داده‌ها** ۳](#_Toc124920890)

[**3. وارد کردن داده‌ها به لیست و Data Frame** ۴](#_Toc124920891)

[**4. معادلات سینماتیکی ربات** ۶](#_Toc124920892)

[**5. رسم حرکت ربات و تشکیل دیتاست مورد نیاز برای شناسایی‌کننده**  ۷](#_Toc124920893)

[**6. نرمال‌سازی**  ۱۴](#_Toc124920894)

[2. شناسایی ۱۵](#_Toc124920895)

[**1. قرار دادن داده‌ها در حالت سری زمانی**  ۱۵](#_Toc124920896)

[**2. مدل شبکه و آموزش**  ۱۷](#_Toc124920897)

[**3. تست شبکه**  ۱۹](#_Toc124920898)

[**4. ذخیره سازی مدل و وزن‌ها**  ۲۱](#_Toc124920899)

[3. کنترلر ۲۱](#_Toc124920900)

[**1. قرار دادن داده‌ها در حالت سری زمانی**  ۲۱](#_Toc124920901)

[**2. مدل شبکه و آموزش**  ۲۴](#_Toc124920902)

[**3. تست شبکه**  ۲۵](#_Toc124920903)

[**4. ذخیره سازی مدل و وزن‌ها**  ۲۷](#_Toc124920904)

# **داده‌سازی**

در این بخش داده‌های مناسب برای آموزش شبکه‌ها از ربات دریافت شده و فیلترهای مناسب روی آن‌ها پیاده می‌شود.

## **داده‌های سنسور**

برای دریافت داده‌های سنسور، در کد نوشته شده برای راه‌‌اندازی ربات به کمک برد raspberry pi 3، کد شماره‌ی 1 را اضافه می‌کنیم. در این چند خط، داده‌های دریافت شده از سنسورها و ارسال شده برای موتور‌ها به عنوان PWM در لیست‌های مربوطه اضافه شده و این لیست در فایل‌های .txt نوشته می‌شوند. سپس این فایل‌ها را برای برداشتن داده‌ها و ایجاد Data Frame در کد اصلی می‌خوانیم.

|  |
| --- |
| left\_wheel.**append**(robot1.ctr)  right\_wheel.**append**(robot1.ctr)  left\_srf.**append**(robot1.sensor\_left)  right\_srf.**append**(robot1.sensor\_right)    f = **open**("left\_srf.txt","w")  f.**write**(str(left\_srf))  f.**close**()  f = **open**("right\_srf.txt","w")  f.**write**(str(right\_srf))  f.**close**()  f = **open**("pwm\_left.txt","w")  f.**write**(str(left\_wheel))  f.**close**()  f = **open**("pwm\_right.txt","w")  f.**write**(str(right\_wheel))  f.**close**() |

کد شماره‌ی 1: نوشتن داده‌های سنسورها و موتورها در فایل درونraspberry pi

## **فیلتر کردن و بهینه‌سازی داده‌ها**

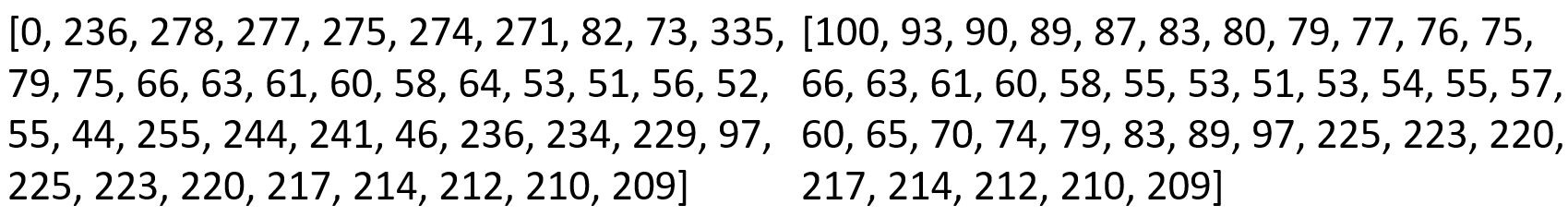
داده‌های بدست آمده از سنسورها دارای نویز و داده‌های پرت فراوانی هستند. زمانی که تمامی داده‌های آموزشی شبکه اینگونه باشند، آموزش آن تقریبا غیرممکن خواهد بود. به همین دلیل با استفاده از یک شرط if و یک فیلتر مرتبه اول ساده و همچنین تنظیم دستی داده‌ها، اعداد بدست آمده از سنسورها را بهبود می‌بخشیم. در ادامه کدهای مربوط به اطلاحات روی داده‌های و نمونه از داده‌های بهبود داده‌شده آورده شده.

|  |
| --- |
| def **FirstOrder**(*x*,*dt*=1,*taw*=9):      a = *dt*/(*dt* + *taw*)      y = [0 *for* i *in* range(**len**(*x*))]      y[0] = *x*[0]  *for* i *in* range(**len**(*x*)-1):          y[i+1] = y[i] + a\*(*x*[i+1] - *x*[i])  *return* y |

کد شماره‌ی 2: فیلتر مرتبه اول ساده برای دمپ کردن تغییرات بزرگ

|  |
| --- |
| def **reject\_dist**(*x\_new*,*x\_old*):  *if* (**abs**(*x\_new*-*x\_old*)>40):  *return* *x\_old*,*x\_old*  *else*:  *return* *x\_old*,*x\_new* |

کد شماره‌ی 3: شرط if برای گرفتن تغییرات و جامپ‌های موحود در مقادیر سنسور



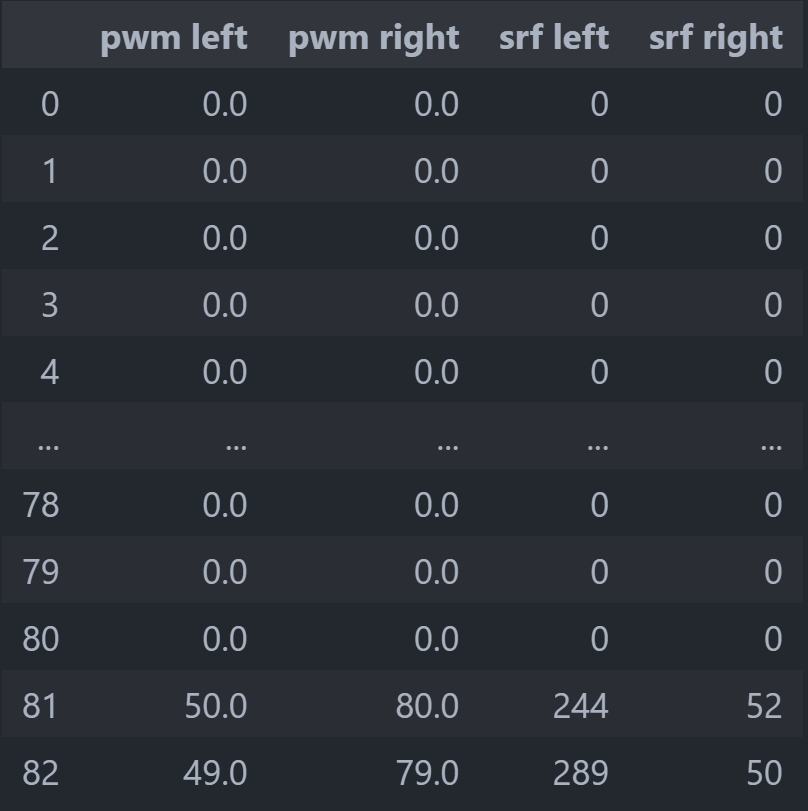
شکل 1: سمت چپ: داده‌های اصلی، سمت راست: داده‌های تغییر داده شده

## **وارد کردن داده‌ها به لیست و Data Frame**

برای خواند فایل‌ها به کمک کتابخانه‌ها globe و os فولدرهای موجود را شناسایی کرده و تمام فایل‌های .txt که با نام‌های pwm\_left، pwm\_right، right\_srf و left\_srf موجود دارند را می‌خوانیم. خروجی این دستور به صورت رشته (String) است که به کمک کتابخانه re (regular expression) اعداد آن را پیدا کرده و در لیست‌های جدید append می‌کنیم. توجه داریم که برای برابری طول داده‌های خوانده شده در هر حرکت، این لیست‌های بدست آمده را تا طول 120 (یکی بیشتر از طول بیشترین داده‌های حرکت) پد می‌کنیم. در نهایت لیست‌های دوبعدی بدست آمده را یک بعدی کرده و در یک Data Frame ذخیره می‌کنیم.

|  |
| --- |
| pwm\_left = list()  pwm\_right = list()  srf\_right = list()  srf\_left = list()  max\_length = 120  *for* FolderName *in* [name *for* name *in* os.**listdir**("./Data robot Modified") *if* os.path.**isdir**(os.path.**join**("./Data robot Modified", name))]:     path = f'./Data robot Modified/{FolderName}'  *for* filename *in* glob.**glob**(os.path.**join**(path, 'pwm\_left.txt')):  *with* **open**(os.path.**join**(os.**getcwd**(), filename), 'r') *as* f:           x = re.**findall**(r'-?\d+\.?\d\*', f.**read**())           x = [**eval**(x[i]) *for* i *in* range(**len**(x))]  *for* i *in* range(max\_length-**len**(x)):              x.**insert**(0,0)  *# add zero at the begining*           pwm\_left.**append**(x)    *for* filename *in* glob.**glob**(os.path.**join**(path, 'pwm\_right.txt')):  *with* **open**(os.path.**join**(os.**getcwd**(), filename), 'r') *as* f:           x = re.**findall**(r'-?\d+\.?\d\*', f.**read**())           x = [**eval**(x[i]) *for* i *in* range(**len**(x))]  *for* i *in* range(max\_length-**len**(x)):              x.**insert**(0,0)  *# add zero at the begining*           pwm\_right.**append**(x)    *for* filename *in* glob.**glob**(os.path.**join**(path, 'right\_srf.txt')):  *with* **open**(os.path.**join**(os.**getcwd**(), filename), 'r') *as* f:           x = re.**findall**(r'-?\d+\.?\d\*', f.**read**())           x = [**eval**(x[i]) *for* i *in* range(**len**(x))]  *for* i *in* range(max\_length-**len**(x)):              x.**insert**(0,0)  *# add zero at the begining*           srf\_right.**append**(x)    *for* filename *in* glob.**glob**(os.path.**join**(path, 'left\_srf.txt')):  *with* **open**(os.path.**join**(os.**getcwd**(), filename), 'r') *as* f:           x = re.**findall**(r'-?\d+\.?\d\*', f.**read**())           x = [**eval**(x[i]) *for* i *in* range(**len**(x))]  *for* i *in* range(max\_length-**len**(x)):              x.**insert**(0,0)  *# add zero at the begining*           srf\_left.**append**(x)  pwm\_left = [pwm\_left[i][j] *for* i *in* range(**len**(pwm\_left)) *for* j *in* range(**len**(pwm\_left[i]))]  pwm\_right = [pwm\_right[i][j] *for* i *in* range(**len**(pwm\_right)) *for* j *in* range(**len**(pwm\_right[i]))]  srf\_left = [srf\_left[i][j] *for* i *in* range(**len**(srf\_left)) *for* j *in* range(**len**(srf\_left[i]))]  srf\_right = [srf\_right[i][j] *for* i *in* range(**len**(srf\_right)) *for* j *in* range(**len**(srf\_right[i]))]  controller\_data = pd.DataFrame({'pwm left': np.**array**(pwm\_left).**reshape**(-1,),                                   'pwm right': np.**array**(pwm\_right).**reshape**(-1,),                                   'srf left': np.**array**(srf\_left).**reshape**(-1,),                                  'srf right': np.**array**(srf\_right).**reshape**(-1,)}) |

کد شماره‌ی 4: خوانن فایل‌های داده‌ها و ذخیره آن‌ها در لیست‌های جدید و Data Frame



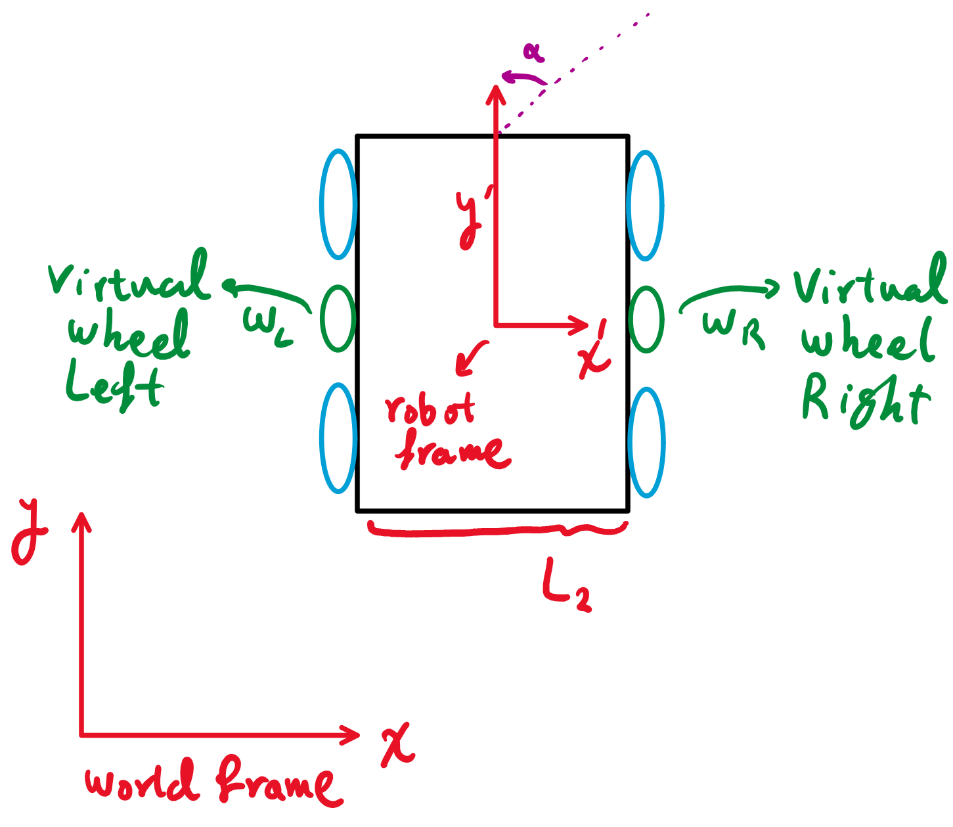
شکل 2: داده‌های موجود در Data Frame

## **معادلات سینماتیکی ربات**

برای آموزش هرچه بهتر شبکه، داده‌های باید نرمال‌سازی شوند. از آن جهت که داده‌های موجود در این موجموعه هم مثبت و هم منفی هستند، از نرمال‌سازی استاندارد (Standard Scaling) برای این امر استفاده می‌کنیم. برای این کار از کتابخانه scikit-learn کمک می‌گیریم. در کد شماره‌ی 7، دو تابع نرمال‌سازی استاندارد و MinMax برای نرمال‌سازی داده‌های یک Data Frame آورده شده. برای تست نرمال‌سازی MinMax نیز امتحان شد که نتایج مناسبی به همراه نداشت.

|  |
| --- |
| def **position**(*pwm\_right*, *pwm\_left*):      r = 0.02      L2 = 1      max\_speed = 100      Ts = 0.1      heading = 0      w = 0      x = [0]      y = [0]      v\_right = np.**array**([i\*max\_speed\*r/100. *for* i *in* *pwm\_right*])      v\_left = np.**array**([i\*max\_speed\*r/100. *for* i *in* *pwm\_left*])      v\_x = (v\_right + v\_left)/2      v\_y = (v\_right + v\_left)/2      w = v\_left - v\_right      w = [i/L2 *for* i *in* w]  *for* i *in* range(**len**(v\_x)-1):          v\_x[i] = v\_x[i]\*(np.sin(heading))          x.**append**(v\_x[i]\*Ts+x[i])          v\_y[i] = v\_y[i]\*(np.cos(heading))          y.**append**(v\_y[i]\*Ts+y[i])          heading = w[i]\*Ts+heading      plt.**plot**(x[0],y[0],'r\*')      plt.**plot**(x[1:],y[1:])      plt.**plot**(x[**len**(x)-1],y[**len**(x)-1],'g\*')  *return* x, y |

کد شماره‌ی 5: توابع معادلات سینماتیکی ربات برای تبدیل PWM به موقعیت x و y



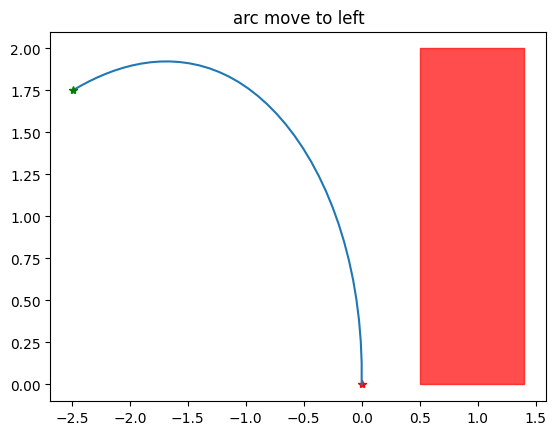
شکل 3: نمای ربات و چرخ‌های مجازی در نظر گرفته شده

## **رسم حرکت ربات و تشکیل دیتاست مورد نیاز برای شناسایی‌کننده**

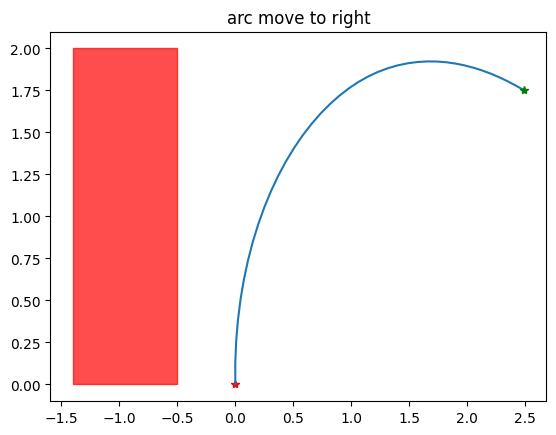
به کمک تابع تعریف شده در قسمت قبل و لیست‌های خوانده شده از فایل‌ها، مسیر‌های طی شده توسط ربات در فاز جمع‌آوری داده‌ را رسم می‌کنیم. به این ترتیب برای هرکدام از 11 حرکت نمودارهای زیر آورده شده که در آن‌ها ستاره قرمز نقطه شروع و ستاره سبز نقطه پایان حرکت است.

|  |
| --- |
| pos\_x = list()  pos\_y = list()  plt.**title**("arc move to left")  x,y = **position**(pwm\_right[:120], pwm\_left[:120])  pos\_x.**append**(x)  pos\_y.**append**(y)  plt.**fill\_between**(np.**arange**(0.5,1.5,0.1), 0, 2, *color*='red', *alpha*=0.7)  plt.**show**()  plt.**title**("arc move to right")  x,y = **position**(pwm\_right[120:240], pwm\_left[120:240])  pos\_x.**append**(x)  pos\_y.**append**(y)  plt.**fill\_between**(np.**arange**(-0.5,-1.5,-0.1), 0, 2, *color*='red', *alpha*=0.7)  plt.**show**()  **.**  **.**  **.**  **.**  pos\_x = np.**array**(**sum**(pos\_x, []))  pos\_y = np.**array**(**sum**(pos\_y, []))  identification\_data = pd.DataFrame({'pwm left': np.**array**(pwm\_left).**reshape**(-1,),                                  'pwm right': np.**array**(pwm\_right).**reshape**(-1,),                                  'x position': np.**array**(pos\_x).**reshape**(-1,),                                  'y position': np.**array**(pos\_y).**reshape**(-1,)}) |

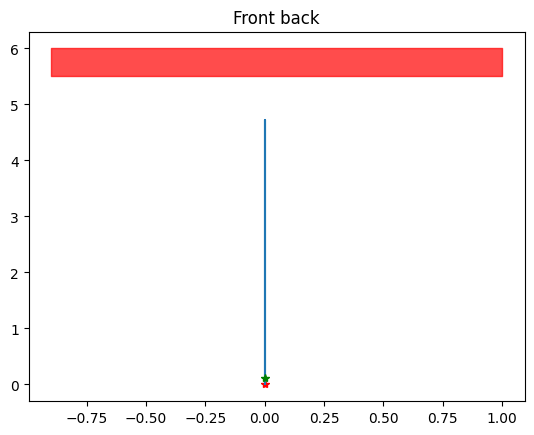
کد شماره‌ی 6: رسم مسر حرکت ربات و بدست آوردن موقعیت x و y آن به کمک تابع تعریف شده و تشکیل Data Frame از آن‌ها



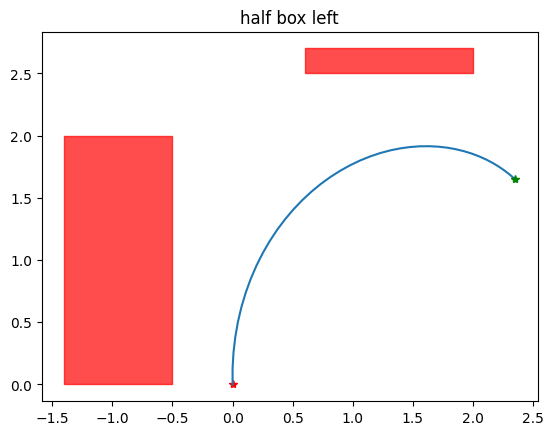
شکل 4: حرکت قوسی به سمت چپ



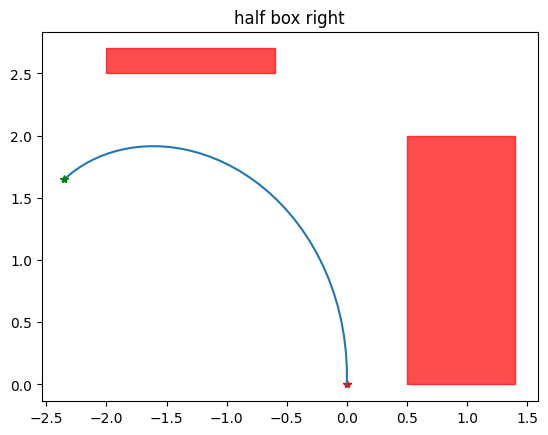
شکل 5: حرکت قوسی به سمت راست



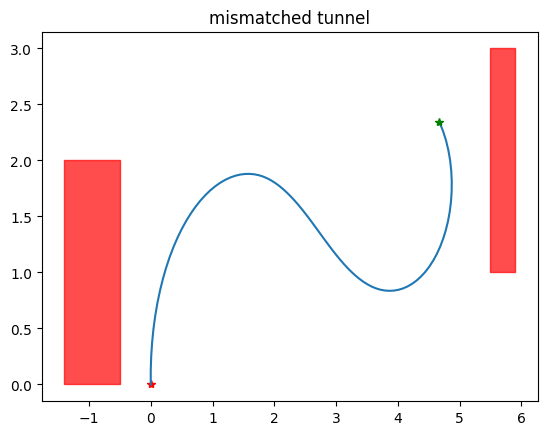
شکل 6: حرکت به جلو و برگشت به عقب



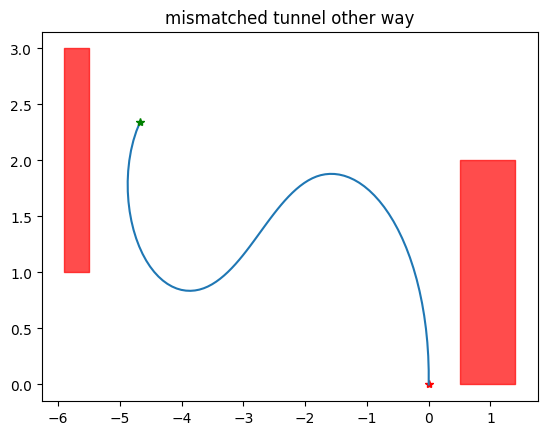
شکل 7: حرکت قوسی به چپ در یک نیمه جعبه



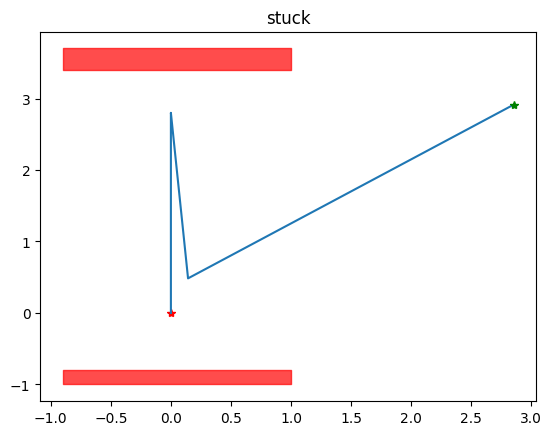
شکل 8: حرکت قوسی به راست در یک نیمه جعبه



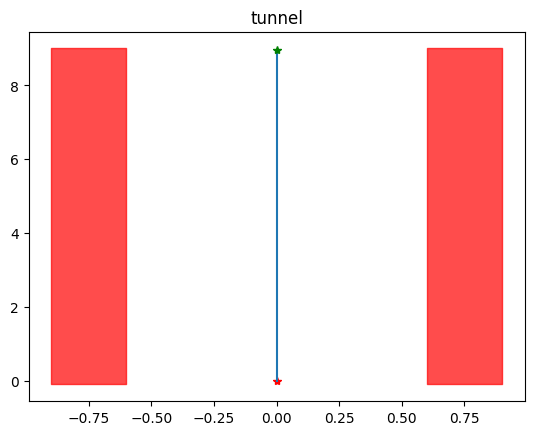
شکل 9: حرکت در یک تونل غیرهمسان



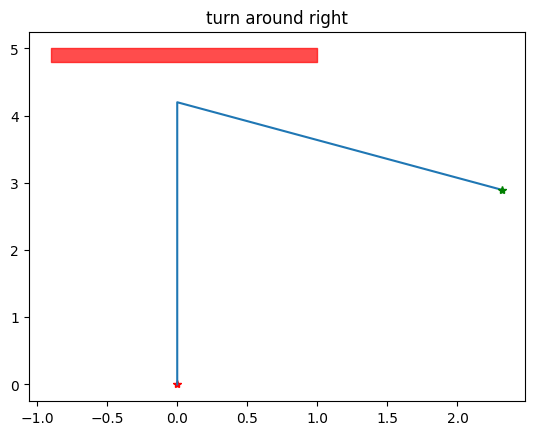
شکل 10: حرکت در یک تونل غیرهمسان از جهت دیگر



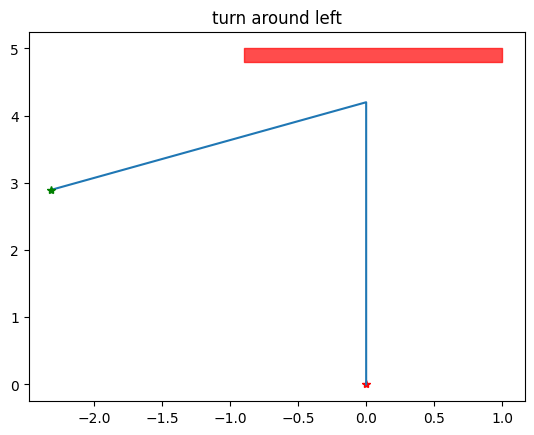
شکل 11: حرکت عقب‌گرد در گیر کردن بین دو دیوار



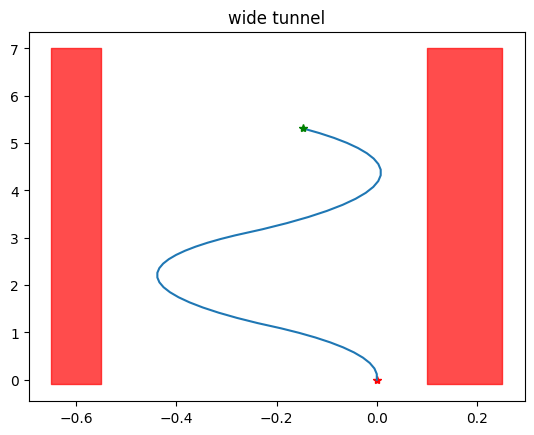
شکل 12: حرکت در تونل باریک و همسان

****

شکل 13: گردش به راست در برخورد با دیوار

****

شکل 14: گردش به چپ در برخورد با دیوار



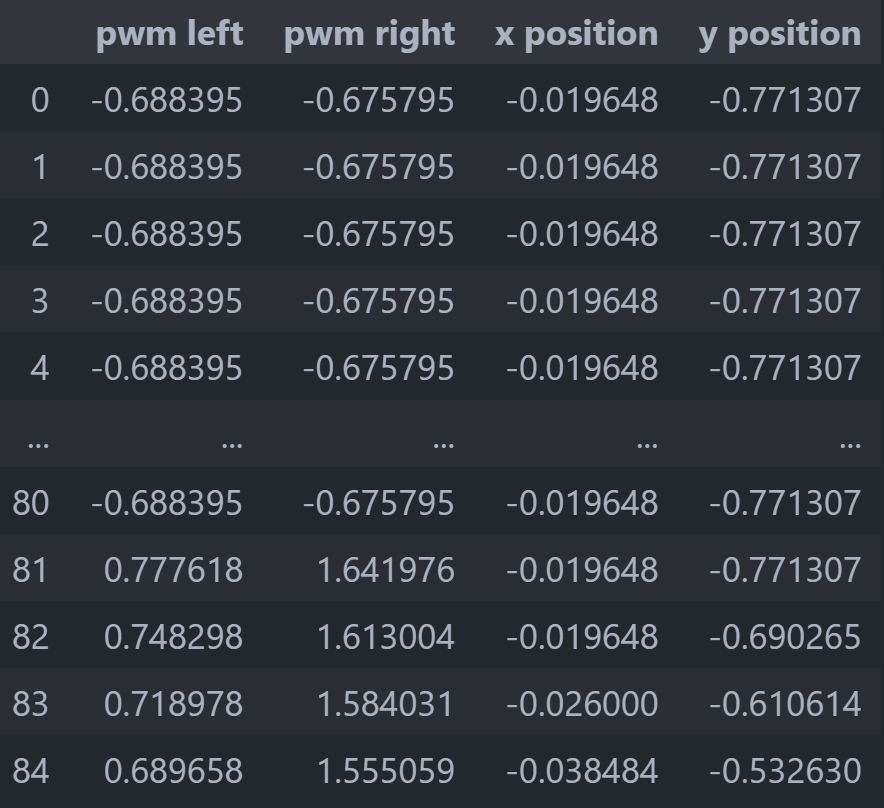
شکل 15: حرکت در تونل پهن و همسان

## **نرمال‌سازی**

برای آموزش هرچه بهتر شبکه، داده‌های باید نرمال‌سازی شوند. از آن جهت که داده‌های موجود در این موجموعه هم مثبت و هم منفی هستند، از نرمال‌سازی استاندارد (Standard Scaling) برای این امر استفاده می‌کنیم. برای این کار از کتابخانه scikit-learn کمک می‌گیریم. در کد شماره‌ی 7، دو تابع نرمال‌سازی استاندارد و MinMax برای نرمال‌سازی داده‌های یک Data Frame آورده شده. برای تست نرمال‌سازی MinMax نیز امتحان شد که نتایج مناسبی به همراه نداشت.

|  |
| --- |
| def **feature\_normalizer\_std**(*x*):      scaler = StandardScaler()  *if* **isinstance**(*x*,pd.Series):  *x* = scaler.**fit\_transform**(*x*.values.**reshape**((-1,1)))  *else*:  *for* i *in* *x*.columns:  *x*[i] = scaler.**fit\_transform**(*x*[i].values.reshape((-1,1)))  *return* *x*  def **feature\_normalizer\_minmax**(*x*):      scaler = MinMaxScaler()  *if* **isinstance**(*x*,pd.Series):  *x* = scaler.**fit\_transform**(*x*.values.**reshape**((-1,1)))  *else*:  *for* i *in* *x*.columns:  *x*[i] = scaler.**fit\_transform**(*x*[i].values.reshape((-1,1)))  *return* *x*  identification\_data = **feature\_normalizer\_std**(identification\_data)  identification\_data.**head**(85) |

کد شماره‌ی 7: توابع نرمال‌سازی داده‌های یک Data Frame



شکل 16: نمونه Data Frame نرمال شده

# **شناسایی**

در این بخش با داده‌های آماده شده به شناسایی مدل ربات به کمک شبکه عصبی می‌پردازیم.

## **قرار دادن داده‌ها در حالت سری زمانی**

در این پروژه، ماهیت داده‌های بدست آمده از ربات سری زمانی می‌باشد، به این ترتیب که داده‌های x و y هر لحظه به x و y مرحله قبل و همچنین PWMهای مرحله قبل وابسته است. به همین دلیل با استفاده از توابع تعریف شده در کد شماره‌ی 8 این داده‌ را به فرم سری‌های زمانی در می‌آوریم. تابع split\_sequences برای پنجره زنی روی داده‌ها به کار می‌رود، به این ترتیب که از دیتاست ورودی 4 ستون اول را تا سطر n\_steps برای مجموعه seq\_x انتخاب کرده و داده‌ی موجود در دو ستون آخر سطر n\_step را برای y\_seq. به این ترتیب سری که منجر به ایجاد y\_seq می‌شود را بدست آورده‌ایم. این سری در x و سری خروجی در y اضافه می‌شوند که آرایه این دو خروجی تابع می‌باشند.

در تابع time\_series\_MLP ورودی دیتاستی است که قصد ایحاد سری‌های زمانی از آن را داریم به همراه اسم ستون‌های مربوط به ورودی (4 ستون) و اسم ستون‌های خروجی (2 ستون) و همچنین تعداد زمان‌های موجود در هر نمونه از سری‌های زمانی. پس از جداسازی ستون‌های مورد نظر به کمک تابع hstack این ستون‌ها را کنار هم قرار داده و آراریه بدست آمده را به تابع split\_sequence می‌دهیم. توجه داریم که سری‌های انتخاب شده دارای 80 نمونه زمانی و 4 ورودی و 2 خروجی هستند. به همین دلیل خروجی x\_train\_seq یک آرایه سه بعدی است (آرایه‌ای از آرایه‌های دو بعدی) که باید به آرایه دو بعدی تبدیل شود. بعد دوم آرایه همان تعداد نمونه‌ها و بعد سوم آن همان تعداد attributeها (4) می‌باشد. خروجی این تابع داده‌های آموزش آماده و ابعاد شبکه است.

|  |
| --- |
| def **split\_sequences**(*sequences*, *n\_steps*):  *# split a multivariate sequence into samples*      x, y = list(), list()  *for* i *in* range(**len**(*sequences*)):  *# find the end of this pattern*          end\_ix = i + *n\_steps*  *# check if we are beyond the dataset*  *if* end\_ix > **len**(*sequences*)-1:  *break*  *# gather input and output parts of the pattern*          seq\_x, seq\_y = *sequences*[i:end\_ix, : -2], *sequences*[end\_ix, -2:]          x.**append**(seq\_x)          y.**append**(seq\_y)  *return* np.**array**(x), np.**array**(y)  def **time\_series\_MLP**(*data*, *inseq1*, *inseq2*, *inseq3*, *inseq4*, *outseq1*, *outseq2*, *n\_steps*):  *# define input sequence*  *# convert to [rows, columns] structure*      in\_seq1 = np.**array**(*data*[[*inseq1*]]).**reshape**(-1, 1)      in\_seq2 = np.**array**(*data*[[*inseq2*]]).**reshape**(-1, 1)      in\_seq3 = np.**array**(*data*[[*inseq3*]]).**reshape**(-1, 1)      in\_seq4 = np.**array**(*data*[[*inseq4*]]).**reshape**(-1, 1)      out\_seq1 = np.**array**(*data*[[*outseq1*]]).**reshape**(-1, 1)      out\_seq2 = np.**array**(*data*[[*outseq2*]]).**reshape**(-1, 1)  *# horizontally stack columns*      data\_set = np.**hstack**((in\_seq1, in\_seq2, in\_seq3, in\_seq4, out\_seq1, out\_seq2))      x\_train\_seq, y\_train\_seq = **split\_sequences**(data\_set, *n\_steps*)      n\_input = x\_train\_seq.shape[1] \* x\_train\_seq.shape[2]      n\_output = y\_train\_seq.shape[1]  *# flatten input*      x\_train\_seq = x\_train\_seq.**reshape**((x\_train\_seq.shape[0], n\_input))  *return* x\_train\_seq, y\_train\_seq, n\_input, n\_output  x\_train\_seq, y\_train\_seq, n\_input, n\_output = **time\_series\_MLP**(identification\_data, "pwm left", "pwm right", "x position", "y position", "x position", "y position", 80) |

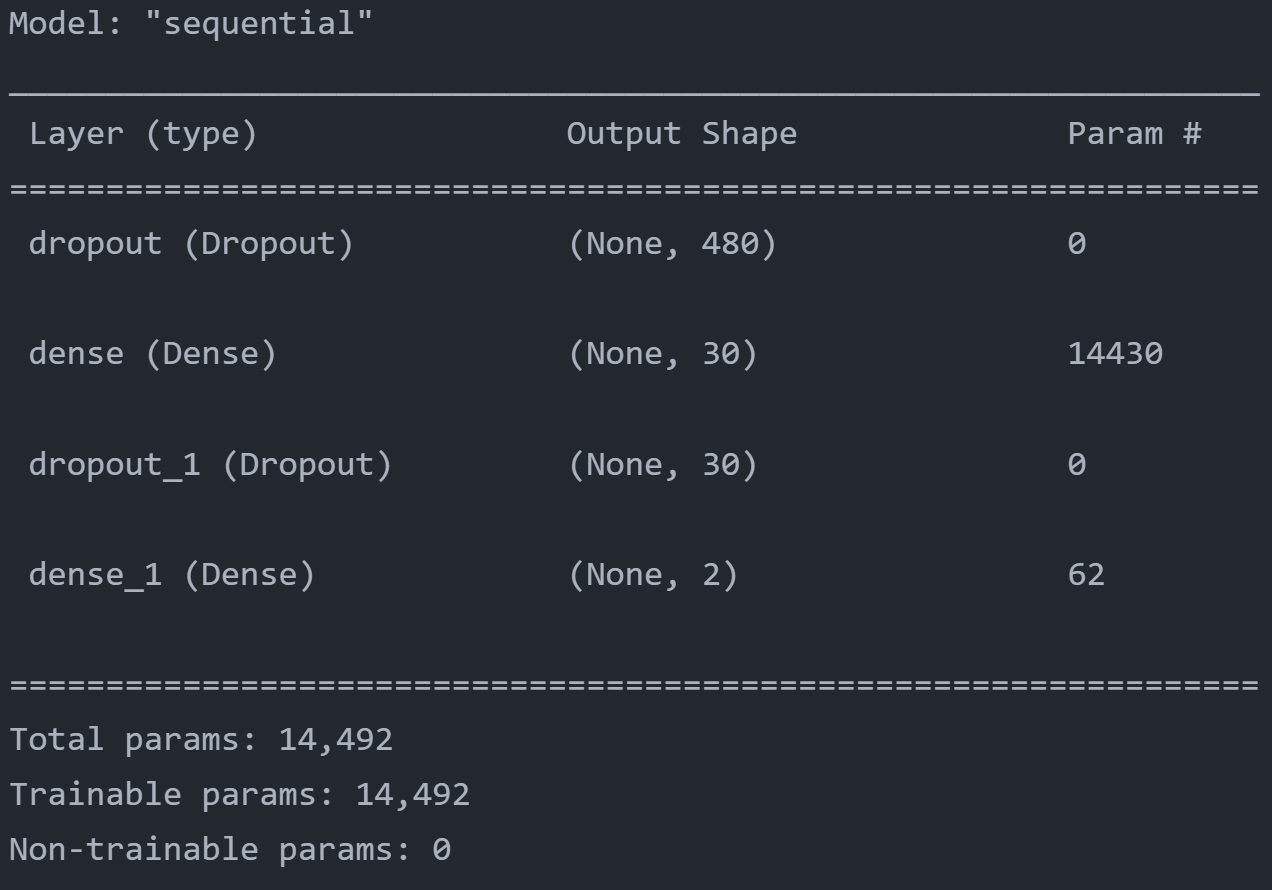
کد شماره‌ی 8: توابع نرمال‌سازی داده‌های یک Data Frame

## **مدل شبکه و آموزش**

با توجه به قضیه universal approximation، یک شبکه عصبی دولایه (1 لایه پنهان) برای تخمین هر تابعی می‌تواند به خوبی عمل کند. به کمک keras یک شبکه دولایه با تابع فعال‌ساز tanh (داده‌های مثبت و منفی داریم) برای لایه پنهان ایجاد می‌کنیم. در این شبکه از لایه‌های dropout استفاده کردیم که به طور رندوم در هر epoch تعداد از نورون‌ها را (که در آرگومان ورودی آن مشخص شده) خاموش می‌کند. این موضوع به generality شبکه کمک می‌کند. بعد ورودی و خروجی شبکه در تابع قبلی بدست آمده.

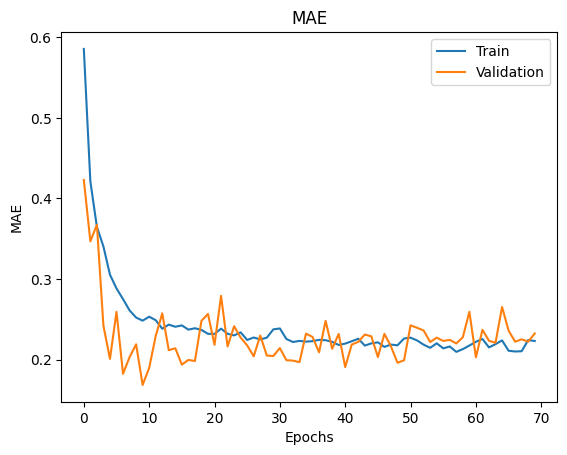
|  |
| --- |
| model\_NN = tf.keras.Sequential()  model\_NN.**add**(tf.keras.layers.**Input**(n\_input))  model\_NN.**add**(tf.keras.layers.Dropout(0.15))  model\_NN.**add**(tf.keras.layers.Dense(30,*activation*='tanh'))  model\_NN.**add**(tf.keras.layers.Dropout(0.1))  model\_NN.**add**(tf.keras.layers.Dense(n\_output))  model\_NN.**summary**() |

کد شماره‌ی 9: تعریف مدل شبکه در keras

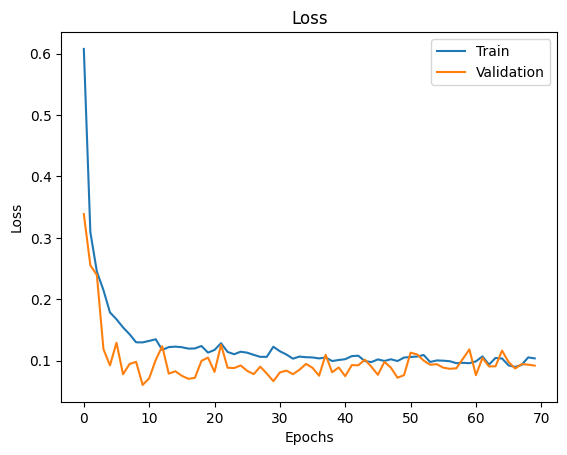


شکل 17: خلاصه مشخصات شبکه تشکیل داده شده

برای آموزش شبکه از Adam optimizer با ضریب یادگیری 001/0 استفاده می‌کنیم. توجه داریم که به علت کم بودن داده‌های سایز batchها را کوچک در نظر می‌گیریم (mini batch). چون عمل مورد نظر برای این شبکه regression است معیار آن را نیز mae قرار می‌دهیم. در ادامه نتایج بدست آمده از آموزش این شبکه آورده شده.



شکل 18: کاهش mae با پیشرفت epochها



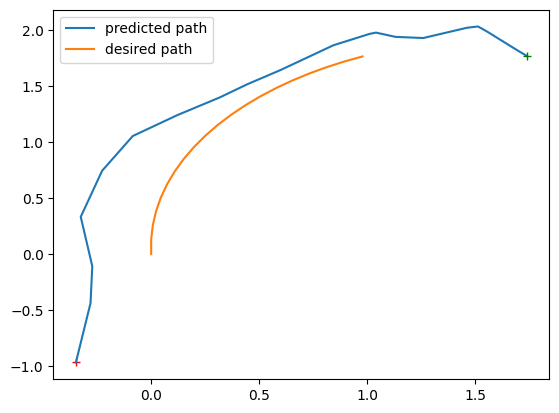
شکل 19: کاهش loss با پیشرفت epochها

## **تست شبکه**

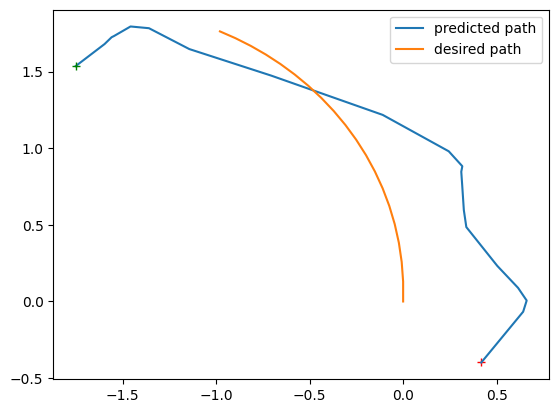
برای تست شبکه با ایجاد یک سری PWM و آماده‌سازی داده‌ها به روش مشابه، از متد predict استفاده کرده و داده‌های بدست آمده و مطلوب را را با تابع position رسم می‌کنیم. همچنین به کمک معیار mean\_squared\_error موجود در scikit-learn خطای خروجی بدست آمده با خروجی مطلوب را می‌سنجیم. همانطور که از نتایج فاز آموزش انتظار می‌رفت، نمودار خروجی با تقریب خوبی نمودار مطلوب را دنبال کرده و خطای بدست آمده با معیار تعریف شده تقریبا 95/0 است.

|  |
| --- |
| pwm\_right1 = np.**arange**(80,60,-1)  pwm\_left1 = np.**arange**(50,30,-1)  pwm\_right = np.**pad**(pwm\_right1, (max\_length-**len**(pwm\_right1),0), 'constant', *constant\_values*=(0,))  pwm\_left = np.**pad**(pwm\_left1, (max\_length-**len**(pwm\_left1),0), 'constant', *constant\_values*=(0,))  x,y= **position**(pwm\_right, pwm\_left)  test\_data = pd.DataFrame({'pwm\_left': np.**array**(pwm\_left).**reshape**(-1,),                            'pwm\_right': np.**array**(pwm\_right).**reshape**(-1,),                            'x position': np.**array**(x).**reshape**(-1,),                            'y position': np.**array**(y).**reshape**(-1,)})  test\_data = **feature\_normalizer\_std**(test\_data)  x\_test, y\_test, n\_input, n\_output = **time\_series\_MLP**(test\_data, "pwm\_left", "pwm\_right", "x position", "y position", "x position", "y position", 80)  prediction = model\_NN.**predict**(x\_test)  **print**(f"mean squared error of the test {np.sqrt(**mean\_squared\_error**(y\_test,prediction))}")  plt.**plot**(prediction[0,0],prediction[0,1],'r+')  plt.**plot**(prediction[:,0],prediction[:,1],*label*="predicted path")  plt.**plot**(prediction[-1:,0],prediction[-1:,1],'g+')  plt.**plot**(x,y,*label*="desired path")  plt.**legend**()  plt.**show**() |

کد شماره‌ی 10: تعریف مسیر تست و آماده‌سازی داده‌های آن. استفاده از متد predict برای بدست آوردن خروجی شبکه



شکل 20: تخمین قوس راست توسط شبکه



شکل 21: تخمین قوس چپ توسط شبکه

## **ذخیره سازی مدل و وزن‌ها**

در آخر برای استفاده مدل در موارد بعدی به کمک متد save مدل را ذخیره می‌کنیم. همچنین وزن‌های لایه‌ها را در یک فایل .txt ذخیره می‌کنیم. حجم مدل ذخیره شده، به دلیل سادگی بسیار ناچیز است.

|  |
| --- |
| wights\_NN = list()  *for* layer *in* model\_NN.layers:      wights\_NN.**append**(layer.**get\_weights**())  f = **open**("weights\_NN.txt","w")  f.**write**(str(wights\_NN))  f.**close**()  model\_NN.**save**(*filepath*='ident\_model.h5', *include\_optimizer*=True) |

کد شماره‌ی 11: ذخیره‌سازی مدل و وزن‌ها لایه‌ها

# **کنترلر**

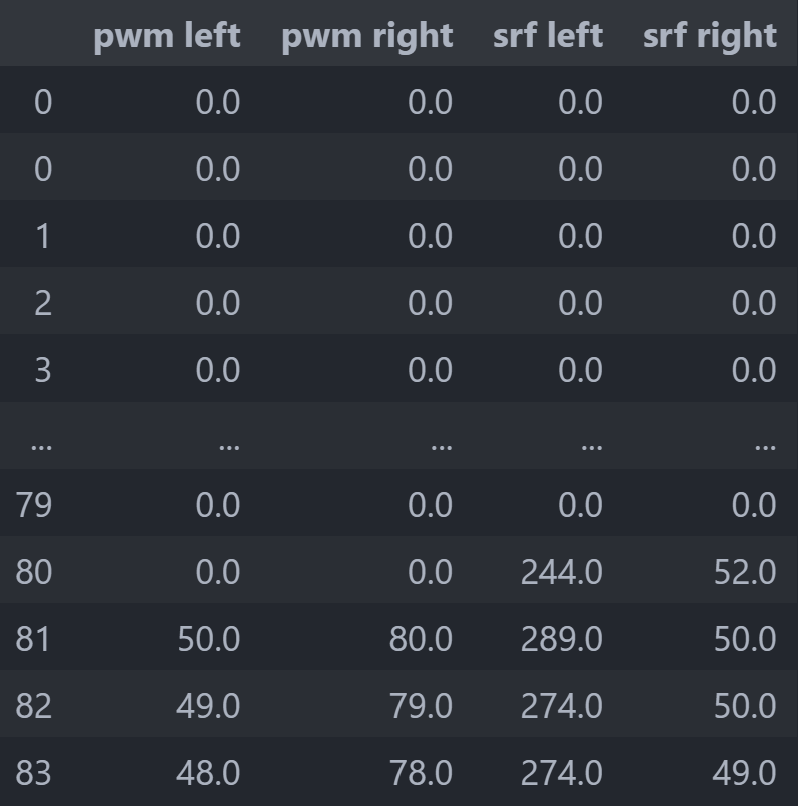
در این بخش با داده‌های آماده شده در بخش اول به طراحی کنترلر برای obstacale avoidance ربات می‌پردازیم.

## **قرار دادن داده‌ها در حالت سری زمانی**

همانطور که پیشتر گفته شد، داده‌های به دست آمده از تست عملی ربات نیاز به پیش پردازش دارند تا در حالت سری زمانی قرار بگیرند. توجه داریم که داده‌های قرار داده شده در لیست‌ها همگی مربوط به یک بازه نمونه‌برداری هستند و به همین دلیل باید داده‌های سنسورها را در دیتاست یک ردیف به بالا شیفت بدهیم. برای این کار با اضافه کردن یک سطر صفر و شیفت دادن دو ستون سنسورها به بالا و همچنین حذف سطر NaN بوجود آمده در انتها داده‌ها را برای پنجره زنی و قرار دادن در سری زمانی آماده می‌کنیم.

|  |
| --- |
| LSTM\_set = pd.DataFrame({      "pwm left": [0.0],      "pwm right": [0.0],      "srf left": [0.0],      "srf right": [0.0]},)  LSTM\_set = LSTM\_set.**append**(controller\_data)  LSTM\_set[['srf left','srf right']] = LSTM\_set[['srf left','srf right']].**shift**(-1)  LSTM\_set = LSTM\_set.**dropna**(*axis*=0)  *# LSTM\_set = feature\_normalizer\_std(LSTM\_set)*  LSTM\_set.**head**(85) |

کد شماره‌ی 12: جلو بردن داده‌های سنسور به اندازه یک بازه نمونه‌برداری



شکل 22: داده‌های شیفت داده شده

برای ایجاد سری‌ها به این صورت عمل می‌کنیم که هر time step یک بردار 4 المانی است که هر کدام از این المان‌ها مربوط به یکی از ورودی‌های شبکه (داده‌های سنسور و PWMها) می‌باشد. این بردارها را به لیست‌های دو بعدی تبدیل می‌کنیم که پس از تشکیل رشته‌ای از نمونه‌ها بتوان آن‌ها را به عنوان هر time step جدا کرد. در ادامه با جلو رفتن در هر سطر، داده‌های سطرهای قبلی باید به انتها اضافه شوند تا فرم کلی سری زمانی شکل بگیرد. برای این کار از دستور cumsum() در کتابخانه pandas استفاده می‌کنیم و داده‌های بدست آمده را در یک ستون جدید در دیتاست قرار می‌دهیم. همچنین چون خروجی ما دو بعدی است (دو PWM)، آن ها را در لیست قرار می‌دهیم.

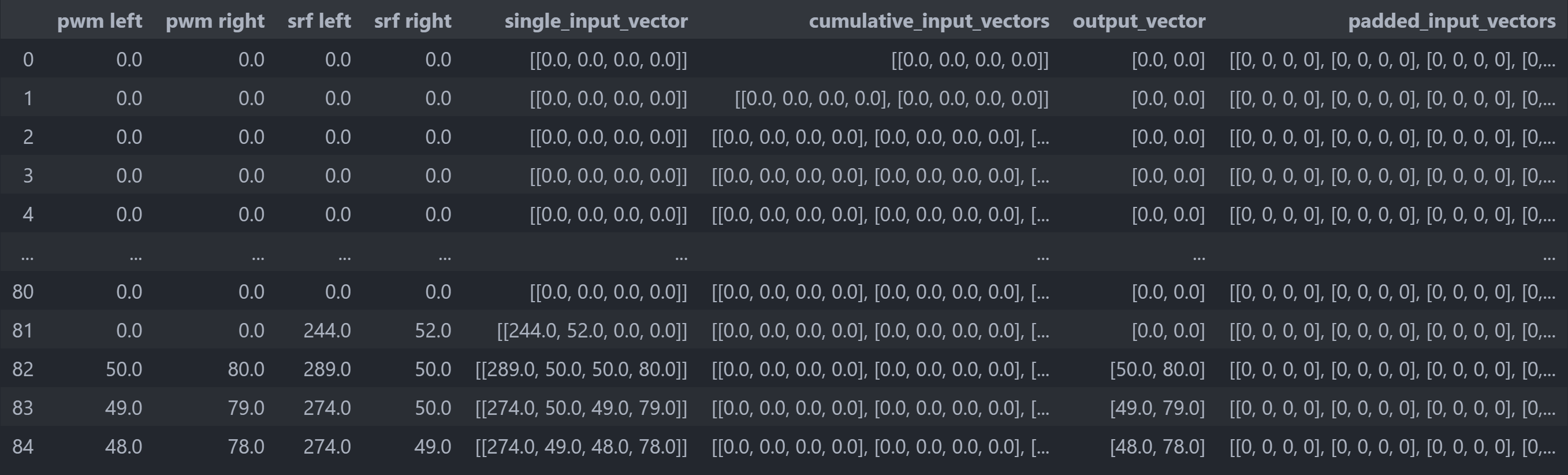
|  |
| --- |
| *# Put inputs into a single list*  LSTM\_set['single\_input\_vector'] = LSTM\_set[['srf left','srf right','pwm left','pwm right']].**apply**(tuple, *axis*=1).**apply**(list)  *# Double-encapsulate list so we can sum it in the next step and keep time steps as separate elements*  LSTM\_set['single\_input\_vector'] = LSTM\_set.single\_input\_vector.apply(lambda *x*: [list(*x*)])  *# Using .cumsum() to include previous row vectors in the current row list of vectors (creating time series)*  LSTM\_set['cumulative\_input\_vectors'] = LSTM\_set.single\_input\_vector.cumsum()  LSTM\_set['output\_vector'] = LSTM\_set[['pwm left','pwm right']].**apply**(tuple, *axis*=1).**apply**(list) |

کد شماره‌ی 13: تنظیم سری‌های زمانی با بردارهای 4 مولفه‌ای

لیست‌های بدست آمده در ستون cumulative\_input\_vectors هم طول نیستند و برای ورودی شبکه باید هم طول شوند. به همین دلیل ماکزیمم طول آن‌ها را پیدا می‌کنیم و به کمک تابع pad\_sequences کتابخانه pandas آن‌ها را تا همان طول ماکزیمم پد می‌کنیم و در یک ستون جدید ذخیره می‌کنیم.

|  |
| --- |
| max\_sequence\_length = LSTM\_set.cumulative\_input\_vectors.apply(**len**).max()  padded\_sequences = tf.keras.preprocessing.sequence.**pad\_sequences**(LSTM\_set.cumulative\_input\_vectors.tolist(), max\_sequence\_length).tolist()  LSTM\_set['padded\_input\_vectors'] = pd.Series(padded\_sequences).**apply**(np.**asarray**)  LSTM\_set.**head**(85) |

کد شماره‌ی 14: هم طول کردن سری‌های بدست آمده در مرحله قبل



شکل 23: دیتاست جدید

بعد از ایجاد دیتاست جدید، ستون پد شده را به عنوان x\_train\_seq قرار می‌دهیم. این ستون باید برای ورودی به LSTM به فرم 3 بعدی باشد به این ترتیب که ابعاد ورودی برابر تعداد attributeها، طول ورودی برابر time step های‌ موجود در هر نمونه و تعداد کلی نمونه‌ها برابر با تعداد سطرهای دیتاست است. همچنین بعد خروجی بسیار واضح و برابر با 2 است.

|  |
| --- |
| x\_train\_init = np.**asarray**(LSTM\_set.padded\_input\_vectors)  *# Use hstack to and reshape to make the inputs a 3d vector*  x\_train = np.**hstack**(x\_train\_init).reshape(**len**(LSTM\_set),max\_sequence\_length,4)  y\_train = np.**hstack**(np.**asarray**(LSTM\_set.output\_vector)).reshape(**len**(LSTM\_set),2)  **print**(x\_train\_init.shape)  **print**(x\_train.shape)  *# Input length is the length for one input sequence (i.e. the number of rows for your sample)*  *# Input dim is the number of dimensions in one input vector (i.e. number of input columns)*  *# Output dimensions is the shape of a single output vector*  input\_length = x\_train.shape[1]  input\_dim = x\_train.shape[2]  output\_dim = **len**(y\_train[0]) |

کد شماره‌ی 15: آماده‌سازی ابعاد داده‌های ورودی و خروجی شبکه

بعد از ایجاد دیتاست جدید، ستون پد شده را به عنوان x\_train\_seq قرار می‌دهیم. این ستون باید برای ورودی به LSTM به فرم 3 بعدی باشد به این ترتیب که ابعاد ورودی برابر تعداد attributeها، طول ورودی برابر time step های‌ موجود در هر نمونه و تعداد کلی نمونه‌ها برابر با تعداد سطرهای دیتاست است. همچنین بعد خروجی بسیار واضح و برابر با 2 است.

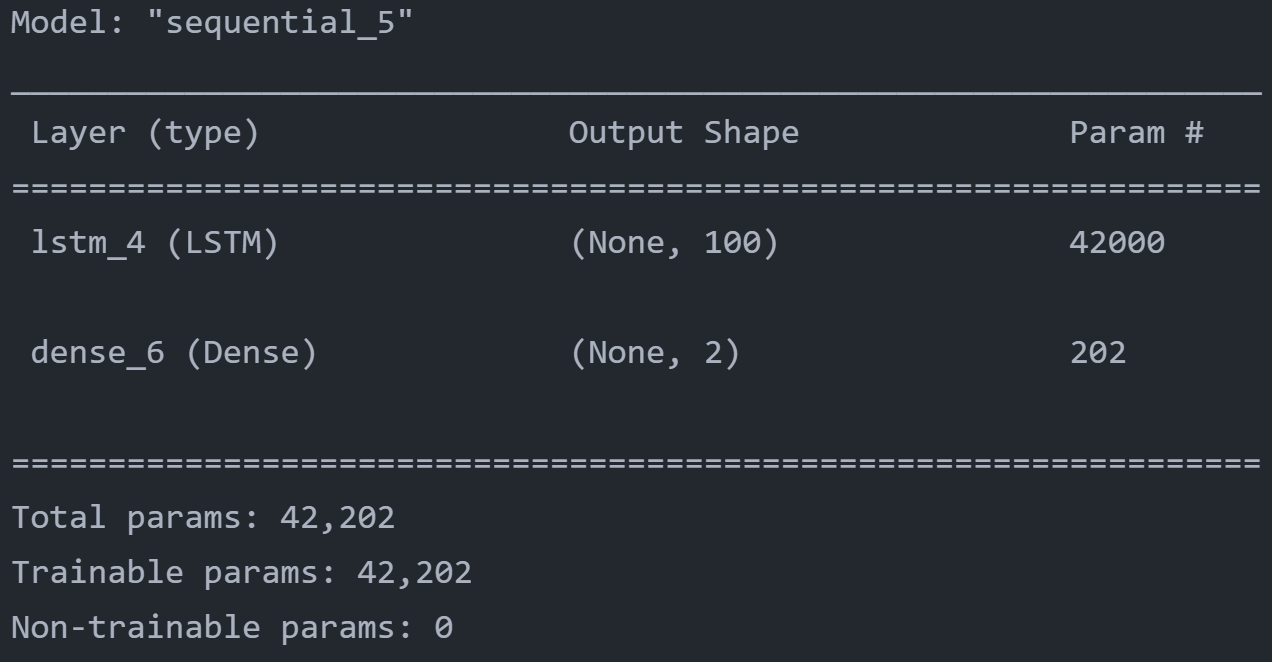
توجه داریم که در این دیتاست، نرمال‌سازی نباید انجام شود. زیرا داده‌ّای صفر اضافه شده به عنوان padding در نرمال‌سازی شده به معنای صفر نیستند و باعث خطای شبکه در تخمین می‌شوند (هرچند معیارهای آموزش و mse بر مطلوب بودن شبکه دلالت کنند).

## **مدل شبکه و آموزش**

با توجه به این موضوع که این کنترلر باید توسط بک برد raspberry pi 3 اجرا شود که توان محاسباتی آن به اندازه یک کامپیوتر مستقل نیست، ترجیح بر این است که ساختار شبکه تا حد ممکن ساده و حجم آن کم باشد تا سر هر زمان نمونه برداری داده‌های جدید برای کنترل ربات آماده باشد. هم همین دلیل فقط از یک لایه LSTM با 100 یونیت و تابع فعال‌ساز tanh (به دلیل مشابه قسمت قبل) استفاده می‌کنیم. خوبی عمل کند. به کمک keras یک شبکه دولایه با تابع فعال‌ساز tanh (داده‌های مثبت و منفی داریم) برای لایه پنهان ایجاد می‌کنیم. در این شبکه از لایه‌های dropout استفاده کردیم که به طور رندوم در هر epoch تعداد از نورون‌ها را (که در آرگومان ورودی آن مشخص شده) خاموش می‌کند. این موضوع به generality شبکه کمک می‌کند. بعد ورودی و خروجی شبکه در تابع قبلی بدست آمده.

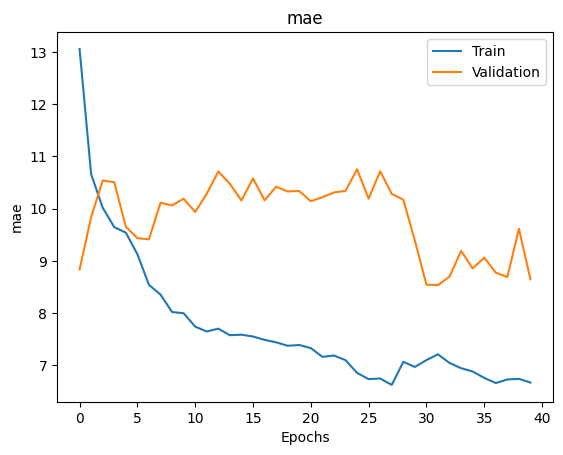
|  |
| --- |
| model\_control = tf.keras.Sequential()  model\_control.**add**(tf.keras.layers.LSTM(100, *input\_dim* = input\_dim, *input\_length* = input\_length))  model\_control.**add**(tf.keras.layers.Dense(output\_dim))  model\_control.**summary**() |

کد شماره‌ی 16: تعریف مدل شبکه کنترلر در keras

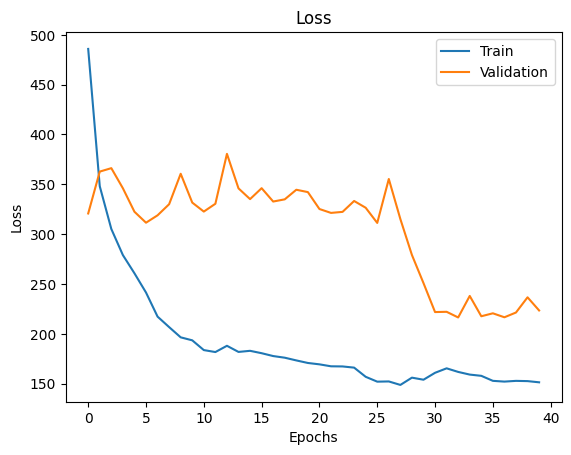


شکل 24: خلاصه مشخصات شبکه کنترلر تشکیل داده شده

برای آموزش شبکه از روش Stochstic Gradient Descend استفاده شده. در ادامه نتایج بدست آمده از آموزش این شبکه آورده شده.



شکل 25: کاهش mae با پیشرفت epochها و رد کردن مینیمم محلی



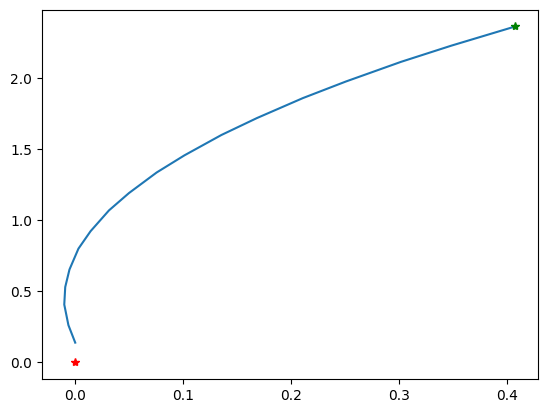
شکل 26: کاهش loss با پیشرفت epochها و رد کردن مینیمم محلی

## **تست شبکه**

برای تست شبکه با ایجاد یک سری PWM و آماده‌سازی داده‌ها به روش مشابه، از متد predict استفاده کرده و داده‌های بدست آمده و مطلوب را را با تابع position رسم می‌کنیم. همچنین به کمک معیار mean\_squared\_error موجود در scikit-learn خطای خروجی بدست آمده با خروجی مطلوب را می‌سنجیم. همانطور که از نتایج فاز آموزش انتظار می‌رفت، نمودار خروجی با تقریب خوبی نمودار مطلوب را دنبال کرده و خطای بدست آمده با معیار تعریف شده تقریبا 95/0 است.

|  |
| --- |
| srf\_left = np.**arange**(40,80,2)  srf\_right = np.random.randint(200,250,**len**(srf\_left))  pwm\_right = np.**array**([0])  pwm\_left = np.**array**([0])  predict = list()  test\_set = pd.DataFrame({"pwm left": pwm\_left.**reshape**(-1,),                          "pwm right": pwm\_right.**reshape**(-1,),                          "srf left": srf\_left[0].reshape(-1,),                          "srf right": srf\_right[0].reshape(-1,)})  *for* i *in* range(**len**(srf\_left)-1):      test\_set = pd.DataFrame({"pwm left": pwm\_left.**reshape**(-1,),                              "pwm right": pwm\_right.**reshape**(-1,),                              "srf left": srf\_left[i].reshape(-1,),                              "srf right": srf\_right[i].reshape(-1,)})      test\_set['single\_input\_vector'] = test\_set[['srf left','srf right','pwm left','pwm right']].**apply**(tuple, *axis*=1).**apply**(list)      test\_set['single\_input\_vector'] = test\_set.single\_input\_vector.apply(lambda *x*: [list(*x*)])      test\_set['cumulative\_input\_vectors'] = test\_set.single\_input\_vector.cumsum()      test\_set['output\_vector'] = test\_set[['pwm left','pwm right']].**apply**(tuple, *axis*=1).**apply**(list)      padded\_sequences = tf.keras.preprocessing.sequence.**pad\_sequences**(test\_set.cumulative\_input\_vectors.tolist(), max\_sequence\_length).tolist()      test\_set['padded\_input\_vectors'] = pd.Series(padded\_sequences).**apply**(np.**asarray**)      x\_test\_init = np.**asarray**(test\_set.padded\_input\_vectors)      x\_test = np.**hstack**(x\_test\_init).reshape(**len**(test\_set),max\_sequence\_length,4)      y\_test = np.**hstack**(np.**asarray**(test\_set.output\_vector)).reshape(**len**(test\_set),2)      prediction = model\_control.**predict**(x\_test)      pwm\_right = np.**array**(prediction[0,1])      pwm\_left = np.**array**(prediction[0,0])      predict.**append**(prediction.tolist())  predict = np.**array**(**sum**(predict, []))  **print**(predict)  x,y = **position**(predict[:,1], predict[:,0]) |

کد شماره‌ی 17: تعریف مسیر تست و آماده‌سازی داده‌های آن. استفاده از متد predict برای بدست آوردن خروجی شبکه

****

شکل 27: تخمین قوس راست توسط شبکه

## **ذخیره سازی مدل و وزن‌ها**

در آخر برای استفاده مدل در موارد بعدی به کمک متد save مدل را ذخیره می‌کنیم. همچنین وزن‌های لایه‌ها را در یک فایل .txt ذخیره می‌کنیم. حجم مدل ذخیره شده، به دلیل سادگی بسیار ناچیز است.

|  |
| --- |
| wights\_contoller = list()  *for* layer *in* model\_control.layers:      wights\_contoller.**append**(layer.**get\_weights**())  f = **open**("weights\_controller.txt","w")  f.**write**(str(wights\_contoller))  f.**close**()  model\_control.**save**(*filepath*='controller\_model.h5', *include\_optimizer*=True) |

کد شماره‌ی 18: ذخیره‌سازی مدل و وزن‌ها لایه‌ها